The Performance of the Concurrent Fault Simulation Algorithms in MOZART

Silvano GAI
CENS-CNR
Politecnico di Torino
10129 Torino, Italy

Pier Luca MONTESSORO
Dip. Automatica e Informatica
Politecnico di Torino
10129 Torino, Italy

Fabio SOMENZI
SGS-THOMSON Microelectronics
20041 Agrate Brianza, Italy

Abstract

MOZART is a concurrent fault simulator for large circuits described at the RT, functional, gate, and switch levels. Performance is gained by means of techniques aimed at the reduction of unnecessary activity. Two such techniques are levelized two-pass simulation, which minimizes the number of events and evaluations, and list event scheduling, which allows optimized processing of simultaneous (fraternal) events for concurrent machines. Moreover, efficient handling of abnormally large or active faulty machines can dramatically improve fault simulator performance. These and related issues are discussed in this paper: both analytical and experimental evidence is provided for the effectiveness of the solutions adopted in MOZART. A new performance metric is introduced for fault simulation. This is based on comparison with the serial algorithm and is more accurate than those used up till now.

1 Introduction

Fault simulation is the process of estimating the effectiveness of a sequence of test vectors by determining which faults in a given set will be detected by its application. The coverage, i.e. the percentage of faults detected, can be checked against a predetermined threshold: if the simulated faults are representative of the prevailing failure mechanisms, this threshold can then be related to the desired defect level. In addition, the list of undetected faults can help the designer improve the coverage of the test vectors, as well as remove logic redundancies appearing as undetectable faults. Its range of features makes fault simulation the ideal choice for test quality assessment, whenever it is applicable. However, the excessive amount of computer time required often hinders its use.

The paper discusses the solutions adopted in the development of MOZART and introduces analytical models and a new and more accurate metric of concurrent simulation performance, to comment on a significant set of experiments.

2 The MOZART system

MOZART is a concurrent multilevel fault simulator. It currently handles the switch, gate, functional, and RT levels.

The switch level model is an enhancement of Bryant's [1]: delays are handled and transistors are considered open when $V_{GS} = 0$. At the gate level primitives are scalar unidirectional elements with a variety of output options. At the Functional Level the user defines primitives for describing RAMs, ROMs and PLAs. An interface is provided to external modules that allows users to describe the function of blocks by a programming language. Hardware is described at the RT level by *guarded statements*, each of which is a set of assignments of expressions (operators) to registers or wires (carriers) conditioned by a guard. A *guard* is either a *level* or an *edge* sensitive expression.

MOZART descriptions are hierarchical: their building blocks are called *units*: instances of other units can be used in their *bodies* (but, recursion is not supported).

Delay times can be assigned to each operator and carrier, with the exception of transistors that switch as soon as the signals at their terminals change. In this way, intrinsic and output delays can be taken into account separately.

3 Multilevel Simulation

A possible model for Hardware Description Language interpretation is a network of operators and carriers whose evaluations are triggered by the occurrence of events. Indeed the MOZART implementation model closely follows this approach: the data structure representing a circuit is a network of *descriptors*, one for each operator and carrier. Event-driven selective trace is used to exploit circuit *latency*. Pending events are kept in a priority queue called a *time queue*. A two-pass procedure [2] is used as a general framework: two distinct phases are iterated for each simulation time:

- The first phase is called *propagation*, since it is essentially related to the propagation of activity via the fanout links. During this phase, all events present in the time queue for that simulation time are retrieved. The network topology is examined and fanout elements are selected for evaluation by placing them in appropriate *evaluation lists*.
- The second phase is called *evaluation* since selected elements are evaluated and events may thus be scheduled or canceled.

Iteration of the two passes stops when there are no more events in the time queue for the given time.

During propagation, if a fanout element belongs to the gate level, the new value being propagated is stored in its descriptor, and the element itself is linked in an evaluation list. If it is a transistor, the new state is immediately computed and any adjacent perturbed nodes are linked in an evaluation list. If it is a node or an RT element, no values are propagated, but the element is nevertheless linked in an evaluation list. The input (i.e. fanin) values will be considered only at the evaluation time.

Fault simulation of N faults requires simulating the behavior of the good machine and N faulty ones. In MOZART, faults are

stuck-at-0/stuck-at-1 on all wires and stuck-on, stuck-open on all transistors.

In Concurrent Fault Simulation, a Reference (R) machine and Concurrent (C) machines (one for each fault) are simulated simultaneously. The close structural and behavioral similarity between these machines means that a differential representation of information is optimal, and at any time only those parts of C-machines which differ from the R-machine are explicitly represented. The total state of a C-machine can be inferred by using the state of the R-machine for the implicit parts.

This differential representation of information requires integration of the data structure of the good machine simulator with a structure for the representation of C machines. In MOZART, this specific data structure is the *machine list*. Machine lists describe the state of all C machines present at the particular primitive element. One or more lists for each primitive element of the circuit link the R and C machine *descriptors*. Gates, transistors and nodes need only one machine list, whereas functional elements may require several.

C-machine descriptors exist only if (and as long as) their contents are different from the associated R-machine descriptors. C-machines have identification numbers (C-numbers) and all machine lists are sorted according to them; C-machine descriptors with identical C-numbers will appear on different lists, and are best viewed as the descendants of a single ancestor. An ancestor and its descendants constitute the explicit part of a C-machine. In concurrent fault simulation, the ancestor is the *fault source*, i.e. the fault description associated with the element in which the fault occurs, while the descendants are the *fault effects*, i.e. the erroneous behaviors caused by the fault on other elements of the network. The main difference between fault sources and fault effects is that the former are always explicit, whereas the latter are subjected to divergence and convergence operations.

Divergence is the operation of creating a new descriptor to represent information for a C machine that becomes different from the reference descriptor. Similarly, *convergence* is the annihilation of a concurrent descriptor as it becomes identical to its associated reference descriptor.

4 MOZART Fault Simulation Algorithm

Introduction of the C machines makes the simulation algorithm more complex. During propagation and element evaluation we must also consider the values of all the C machines differentially coded by the machine list. It is thus advantageous to use a single unified control mechanism for the coordinated traversal of an arbitrary number of machine lists. The major benefits are reduced programming complexity and high-speed execution. A suitable traversal mechanism is Multi-List Traversal (or MLT), described in [3]. This mechanism controls the entire traversal, including event processing for R and C machines, the handling of explicit as well as implicit C-descriptors, divergences and convergences, and various other tasks.

MLT traverses lists of items appearing in C-machine order, so that those for the same machine on different lists are reached and then processed simultaneously. When a C-descriptor is present on some lists only, the R-machine information (in the first item of every list) replaces the absent C-descriptor.

The lists to be traversed in coordination are either the input, output, and state lists of a single element or the output lists of interconnected elements. The former case is called *Evaluation MLT* and takes place during the second step to compute the new state of complex elements. The latter case is called *Propagation MLT* [4] and takes place during the first step to deliver new signals to fanout elements, which keep local copies of them (gates and switches).

The cost of concurrent simulation on large circuits containing a significant number of faults is proportional to the number of descriptors traversed. An efficient algorithm should traverse the minimum number of descriptors. *Fault dropping* is widely used for this purpose and is described is in section 6. Several additional techniques have been adopted in MOZART in order to improve performance. First, the number of events should be minimized, since every event may trigger a propagation and some evaluation MLTs. Minimizing the number of events thus reduces the number of traversals. For this purpose, MOZART uses the *Levelized Two Pass* algorithm (LTP) [2], which is based on the idea of ordering the propagation of events through the network. This can be obtained with levelizing.

To levelize a network means to define a partial order among network elements, so that, at a given time, the value of an element can depend only on the values of elements having a lower level. In the second step, element evaluation starts from those having the lowest level. With this method, an evaluation is not repeated as the result of variation of an element with a lower level. Levelizing is possible at all levels of abstraction, if loops of elements having zero delay are forbidden. How to levelize a network is described in [2].

Levelizing is more efficient if the number of zero-delay elements in the network is large. The presence of these elements is quite common in the early stages of a project (e.g., when there are no accurate timing data), or in the design of circuits by means of library models (e.g. semi-custom circuits). The timing characterization of these models is generally accomplished by associating the delays only with the peripheral elements of the cell, leaving the inner elements at zero delay.

In addition to greater efficiency (see Section 6), there are also several advantages regarding correctness. If two events occur at the same simulation time on the same list and machine, then during the evaluation MLT triggered by the first event the emitting lists have not reached their final and correct configuration. This causes element evaluation with inconsistent input and/or state values, which, for certain sequential primitives, may result in erroneous and irreversible state changes.

LTP minimizes the number of events. However, this does not prevent the scheduling of more events for different machines at a given simulation time on a list. Inefficiency would result from letting each of them trigger a propagation MLT. *List event* and *fraternal event processing* are used to group such MLTs into one MLT only.

4.1 List Events and Fraternal Event Processing

In concurrent fault simulation, an event is a 4-tuple (E, C, T, S), where E is the element affected by the activity, C is the C-number of the concurrent machine, T is the occurrence time, and S is the new state. Events with the same E and T values are said to be *fraternal*. Separate descriptors are conventionally placed in the time queue for each event. This, however, makes the coordinated processing of fraternal events difficult. In MOZART, therefore, one descriptor only serves a set of fraternal events. It represents a *list event* and indicates the presence of activity on a machine list. A list event is a 3-tuple (E, T, F), where E and T are the same as before and F is the number of fraternal events it corresponds to.

The items in the machine lists contain a flag showing if they are scheduled or quiescent and, if scheduled, the value of C, T and S, so that by traversing the list associated to the list event, the 4-tuple (E, C, S, T) can be reconstructed. This approach may seem cumbersome at first, but it has the major advantage of allowing all fraternal events to be processed in a single MLT. Since LTP

guarantees that a single list event will be processed at a given time on a given list, the number of machine lists traversed is minimized.

Fraternal Event Processing (FEP) also ensures that, if an R-event is due, the R-machine is processed first during MLT. This is important, because the R-machine represents the absent (implicit) C-machines on many of the lists being traversed and thus should be updated before any such C-machine is evaluated so as to avoid useless convergences and divergences.

List events, as described here, are similar to the composed events introduced in [5] and adopted in [6]. We do not, however, process list events before propagating the activity they entail, as proposed in [6]. By contrast, the way we do propagation MLT (an emitting list is traversed only once as such) allows us to avoid the creation of the so-called *phantom* gates, without resorting to preliminary processing of list events.

To estimate the benefits of FEP, let us consider the case in which the R-machine is quiescent. Processing an event requires the scanning of a machine list to reach a C-machine descriptor and therefore the traversal of a number of items. We compare the number of traversals in FEP with list events (N_t^{fep}) and in the strategy which deals with each fraternal event separately (N_t^s). Let L be the length of a machine list and $E(L)$ its expected value. Without FEP, the average positioning cost for a set of fraternal events is given by:

$$E(N_t^s) = (E(L)+1)\frac{E(F)}{2} \tag{1}$$

because for each event positioning starts from the beginning of the list. However, with FEP the average positioning cost is:

$$E(N_t^{fep}) = (E(L)+1)\frac{E(F)}{E(F)+1} \tag{2}$$

since positioning starts from the last processed fraternal event and the gain is:

$$\frac{E(N_t^s)}{E(N_t^{fep})} = \frac{E(F)+1}{2} \tag{3}$$

By similar reasoning, if the R-machine is active one obtains:

$$\frac{E(N_t^s)}{E(N_t^{fep})} = \frac{E(F)+1}{2}\left(1+\frac{1}{2E(L)}\right) \tag{4}$$

In both cases, the advantage of FEP increases linearly with F.

4.2 Trigger Inhibition

The number of traversals is minimized by the above two techniques. One can also optimize by minimizing the actions associated with the traversals. Traversal of one block to position on the next has a very low cost compared to traversal associated with evaluation. We particularly want to avoid evaluations that produce results already available [3].

Let E be the set of events that caused the MLT. A simple case is that it is useless to evaluate machine C_i if no events in E affect either C_i or the R-machine. More complex cases arise when events for the R-machine belong to E. Here, it is necessary to see if machine C_i is explicit and quiescent on all those lists where events for the R-machine occur. If this happens, *trigger inhibition* takes place and machine C_i should not be evaluated on the son lists since none of its inputs is changing. Two cases must be considered at this point. If machine C_i is already explicit on a son list, no particular action is required, whereas if it is implicit, a change in the R-machine must cause a divergence. Machine C_i must be diverged with the old state of the R-machine, which is therefore temporarily saved before starting MLT. A list for which the value of the R-machine has changed is called a *potential divergence list.* This attribute is tested when trigger inhibition is recognized on a C-machine to decide whether to diverge or not.

5 Fault detection and fault dropping

Fault detection is compared observation of the R and C machine values on a series of predetermined test points (generally the outputs) to see whether the test pattern has covered some of the simulated faults. This observation takes place in suitable temporal windows (linked to the tester timing), called *detection windows.* When a fault is detected on the outputs, the simulation of the corresponding C-machine is normally stopped. *Fault dropping* is the mechanism which suspends the simulation of the detected faults, so as to free memory and considerably reduce the number of traversals.

Fault detection and fault dropping are part of the more general issue of observation and statistics. Machine behaviors must be observed at test points and statistics such as size (number of descriptors) and number of events for each machine must be collected, so as to identify machines behaving very differently to the R-machine, and thus contradicting the assumption upon which the concurrent simulation is based. The simulation time of these machines may become very high (90 % and more) and dominate the entire simulation. Two types of these machines exist: hyperactive and hypertrophic machines.

The former generally concern faults which affect critical control signals, causing oscillations. These faults generally represent less than 5 % of the total. These machines are dropped, even if they have not yet been detected at test points, as the corresponding faults are catastrophic for circuit operation. An example of this kind of fault is given in Fig. 1. The s-a-1 fault on the control signal of latch L2 modifies the behavior of the circuit, allowing the oscillation of the relative C-machine when CK holds the value 1.

The second type of machines concern faults that inhibit network initialization, causing huge X machines. These machines can only be *potentially detected.* A typical example is the s-a-0 on the CK input of latch L1 in Fig. 1. It should be noted that this fault will be detected by any input pattern covering both s-a-0 and s-a-1 faults on the output of L1, although there is no equivalence between these faults.

A fault is marked as hyperactive and dropped when the number of events on the associated C-machine has been too high with respect to the number of events on the R-machine in a given interval of time.

In the same way a hypertrophic fault is dropped when the size of the related C-machine becomes comparable to that of the R-machine and the fault is already marked as potentially detected.

6 Experimental results

We shall now present data from several experiments to support the claim that the techniques described effectively improve the efficiency of multilevel fault simulation.

Table 1 collects general information on the circuits and input vectors used in the simulation test runs.

C6288a and C6288b are two versions of a 16 × 16 parallel multiplier belonging to the set of benchmark circuits proposed in [7]. In C6288a, 0-delay elements are used, whereas C6288b is a unit-delay version. Their comparison shows the potential advantages of LTP simulation. LSSD is a small circuit designed according to the well-known testability methodology [8]. It has

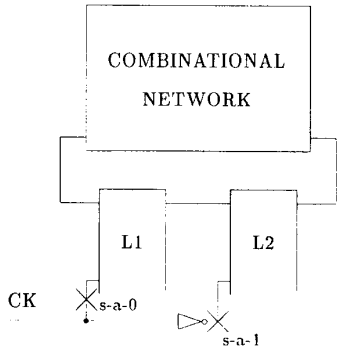

Figure 1: Hyperactive and hypertrophic faults

circuit	delays	type[a]	switch	gate primitives	RT	RAMs	patterns	faults
C6288a	no	C	—	2416	—	—	85	7744
C6288b	yes	C	—	2416	—	—	85	7744
GA	yes	S	—	4002	11	—	2250	9635
LSSD	yes	S	—	377	—	—	4386	939
BS	yes	S	—	12	235	2	252	224
MULT8a	no	C	—	—	15	—	56	64
MULT8b	no	C	—	362	—	—	56	1388
MULT8c	no	C	1190	651	—	—	56	2498

[a]C: combinational - S: sequential

Table 1: Summary of experiments

circuit	R	25%	50%	75%	100%
C6288a	6.33	17.51	37.47	59.51	83.99
C6288b	153.42	385.17	573.28	819.68	1057.39
GA	136.05	1109.62	2330.98	4571.99	5844.46
LSSD1	68.93	92.98	117.05	140.61	169.04
LSSD2	68.82	271.26	505.74	694.31	899.11
BS	231.01	261.13	289.41	316.68	395.27
MULT8a	0.37	0.45	0.54	0.63	0.72
MULT8b	0.98	1.71	2.36	3.12	3.85
MULT8c	41.80	311.52	696.81	931.24	1540.27

Table 2: Times (in seconds) as a function of the number of C-machines

been simulated with and without circuit initialization to show the impact of uninitialized faulty machines. GA is a gate array circuit, BS is a bit sliced processor, and MULT8a, MULT8b, and MULT8c are three successive refinements of a 8×8 multiplier.

Table 2 reports CPU times (measured on a DEC VAX 8650 running VMS) as a function of the numbers of C-machines simulated. Data refer to the simulation of the R-machine alone (column R) and of various percentages of all possible C-machines (after fault collapsing). Times for 25%, 50%, and 75% of the C-machines were averaged over several independent random samples of faults.

In what follows $T_{r\alpha}$ is the time for the simulation of the R-machine alone for circuit α and $T_{cf\alpha}$ that for the concurrent simulation of all faults. The ratio $\frac{T_{cf\alpha}}{T_{r\alpha}}$ is written K_α

Data from Table 2, normalized with respect to $T_{r\alpha}$ are plotted in Fig. 2, while Fig. 3 plots $U_\alpha(t)$, i.e. the fraction of undetected faults as a function of the fraction of applied test vectors for circuit α.

We interpret the plots in Fig. 2 with reference to Fig. 3. If we assume the immediate dropping of a fault upon detection, the normalized mean life of a C-machine for circuit α is given by

$$M_\alpha = \int_0^1 U_\alpha(t)dt \qquad (5)$$

In general, higher values of M_α lead to increased values of K_α, as one would obviously expect. This is the case for circuit GA, which has the highest value for both M_α and K_α. There are, however, circuits such as C6288a, C6288b, and LSSD2, which have significantly different K_α values in spite of their similar M_α values. First of all, the high value of K_{MULT8c} is due to the present absence of trigger inhibition at the switch level. To understand the difference between the two versions of C6288, one should consider that the impact of fault detection and dropping time, which is approximately the same in absolute terms, is greater in relative terms on the shorter of the two simulations. Indeed, in C6288a, the time for detection and dropping is comparable to that spent for simulation. Since the fault detection and dropping time is also negligible in LSSD2, it is clear that C6288a and LSSD2 have similar K_α values only accidentally. It is the discrepancy between C6288b and LSSD2 that deserves an explanation: it is, in fact, due to the presence in the latter of large uninitialized C-machines corresponding to potentially detected faults.

The impact of such faults on simulation speed is illustrated by Fig. 4. Both simulations of LSSD1 and LSSD2 refer to the same circuit and input vectors. The only difference is that latches in LSSD1 are initialized to known values before faults are injected,

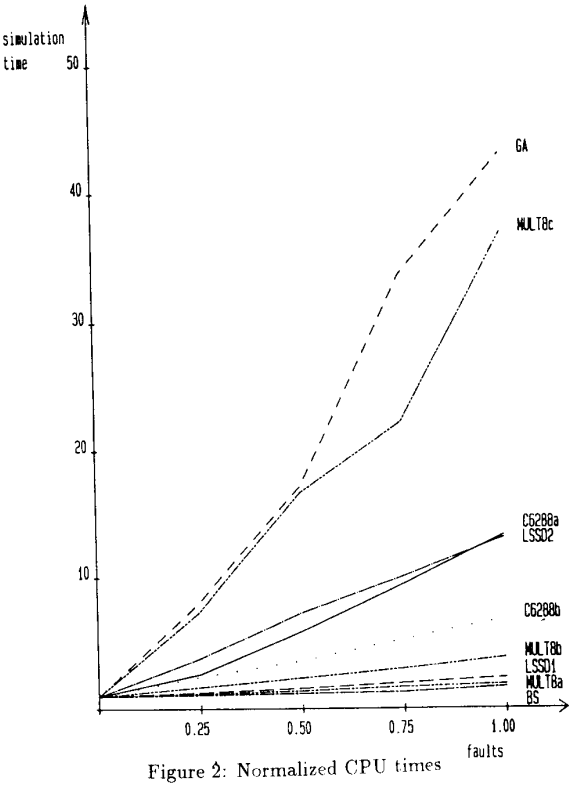

Figure 2: Normalized CPU times

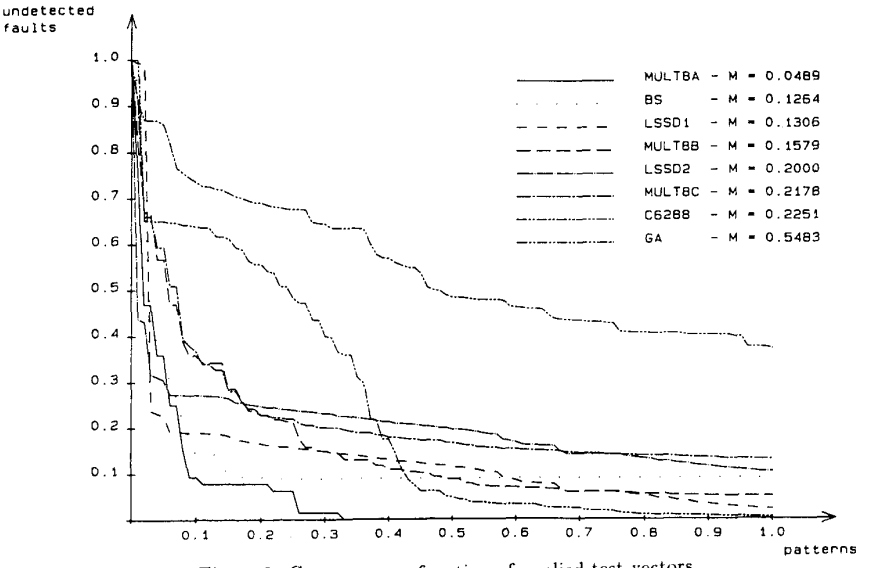

Figure 3: Coverage as a function of applied test vectors

whereas in LSSD2 they are not. As a consequence, large uninitialized C-machines develop in LSSD2, due to faults on the clock input lines of latches. These faults, on the other hand, cause solid detection if the C-machines have been initialized prior to fault injection. Their effects are amplified by the small size of the circuit, which prevents dropping of hypertrophic machines.

On circuits larger than LSSD, the speedup achieved by dropping hypertrophic and hyperactive machines is often quite significant and therefore difficult to quantify, since the mere presence of a few hyperactive faults can prevent a simulation run from ending in a reasonable time. For circuit GA, it has been established that dropping hypertrophic and hyperactive machines (2.2% of the total) accelerates simulation by a factor greater than 100.

M_α also allows us, via a comparison with serial fault simulation, to define a metric for fault simulation performance. Indeed, if $n_{f\alpha}$ is the total number of faults for circuit α, $T_{sf\alpha}$, then the

α	$n_{f\alpha}$	M_α	K_α	G_α	S_α
C6288a	7744	0.2251	13.27	131.4	631.1
C6288b	7744	0.2251	6.89	253.1	1314.8
GA	9635	0.5483	42.96	123.0	229.6
LSSD1	939	0.1306	2.45	50.4	647.6
LSSD2	939	0.2000	13.06	14.5	77.9
BS	224	0.1264	1.71	17.1	315.5
MULT8a	64	0.0488	1.95	2.1	67.4
MULT8b	1388	0.1579	3.93	56.0	437.7
MULT8c	2498	0.2178	36.85	14.8	69.7

Table 3: Serial vs. Concurrent Simulation

time for serial fault simulation (with fault dropping), is approximately given by

$$T_{sf\alpha} = (1 + n_{f\alpha} M_\alpha) T_{r\alpha} \tag{6}$$

The speed gain of concurrent over serial fault simulation is expressed by

$$G_\alpha = \frac{1 + n_{f\alpha} M_\alpha}{K_\alpha} \tag{7}$$

Table 3 reports data derived from those of Table 1 and 2. Besides G_α, it reports the average speed of simulation of a C-machine, relative to the speed of simulation of the R-machine as

$$S_\alpha = \frac{n_{f\alpha} T_{r\alpha}}{T_{cf\alpha} - T_{r\alpha}}, \tag{8}$$

S_α is used in [4,6] as an indicator of the performance of concurrent fault simulation and is shown here to allow comparison with previous work. We believe, however, that G_α is a more accurate indicator than S_α. S_α only accords concurrent fault simulation the benefits of fault dropping. In spite of their importance, these are neither specific to that approach, nor an integral part of it. If we consider the results in [4] relative to circuit "Mult 0-del" (it is C6288a with a longer test sequence), S_α is extremely high (8036), but the low value of G_α (59) shows that most of the simulation is carried out with very few faults active ($M_\alpha = 0.0148$).

Also, by considering only the time directly accountable to the simulation of the faulty machines, S_α tends to amplify the gains of concurrent fault simulation when they are already high (i.e., $T_{r\alpha}$ is close to $T_{cf\alpha}$), as in the case of C6288b and MULT8a.

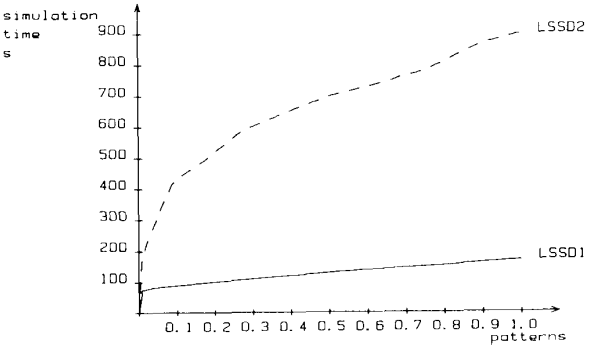

Figure 4: Simulation CPU times for experiments LSSD1 and LSSD2

We now consider the effects of Fraternal Event Processing and List Events. With reference to equations (1-3), Table 4 reports the values of $E(F)$ measured on circuits GA, C6288a, C6288b and MULT8b and the values of $\frac{E(NT_*)}{E(NT_{fep})}$.

circuit	$E(F)$	$\frac{E(NT_*)}{E(NT_{fep})}$
C6288a	6.73	3.87
C6288b	6.14	3.74
GA	3.4	2.20
MULT8b	2.96	1.98

Table 4: Efficiency of fraternal event processing

Equation 1 shows that, without FEP, the time spent for an event occurring on a C-machine (with no simultaneous event for the R-machine on the same list) is bound to increase roughly linearly with $E(L)$. Alternately, with FEP and List Events, it varies as $\frac{E(L)+1}{E(F)+1}$; while not constant, this grows at a much lower rate then $E(L)$.

As far as events for the R-machine are concerned, their costs are proportional to the length of machine lists. Finally, events for C-machines that occur at the same time as events for the R-machine on the same list require no additional overhead for collecting and traversing lists. Their cost is therefore independent of n and very low. If the number of C-machines is varied as in Fig. 2, then the number of events processed per second decreases as more faults are simulated. There is then an optimum value of faults that maximize simulation speed. However, with FEP this optimum value often falls near to or even beyond $n_{f\alpha}$. If this is not the case, dividing faults in lots and processing them in several passes saves simulation time.

A comparison of data for the two experiments on circuit C6288 (see Table 2) shows the efficiency of levelized simulation. The example is particularly favorable to LTP simulation for two reasons: all elements in C6288a have 0 delay and the logical depth of the circuit is very high. Both factors determine how far useless events can propagate before they are stopped by either primary outputs or elements with non-zero delays.

Lastly, we consider the advantages of multilevel simulation. A comparison of MULT8a, MULT8b, and MULT8c shows the reduction in CPU time obtained by resorting to more abstract descriptions of the circuit. However, one should also notice the reduced significance of fault coverage established at gate and RT levels with respect to that obtained with a faithful structural description.

7 Conclusions

In this paper, the principles of multilevel fault simulation have been discussed and the algorithms adopted in MOZART for their efficient implementation have been presented. The emphasis in those techniques is on the reduction of the number of traversed items in concurrent machine lists and the avoidance of unnecessary evaluations and events. The impact on simulation performance has been studied both analytically and experimentally. A new and more accurate metric of fault simulation performance has been introduced. The scattering of values reported in section 6 shows the need for a large set of significant benchmarks. Considerable work is still required in this direction to permit the meaningful comparison of programs and algorithms. However, the same results show that MOZART is adequate for the fault simulation of the custom and ASIC IC's for which it is currently employed.

It has been seen that hyperactive and hypertrophic faults adversely impact fault simulation speed. Their identification and removal is therefore mandatory. The way this is done in MOZART is reasonably efficient and can be improved in two ways: earlier detection of oscillating machines; non-simulation of faulty machines, such as the one discussed in section 5, to which the two following conditions apply:

- their simulation could not predict more than *possible* detection.
- their detection can be inferred from that of other non-equivalent faults.

Potential savings exceed 50% of the simulation time on sequential circuits.

References

[1] R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," IEEE Trans. on Computers, Vol. C-33, No. 2, Feb. 1984, pp. 160-177.

[2] S. Gai, F. Somenzi, M. Spalla, "Fast and Coherent Simulation with Zero Delay Elements," IEEE Transactions on CAD/ICAS, Vol. CAD-6, no. 1, January 1987, pp. 85-92.

[3] E. Ulrich, "Concurrent Simulation at the Switch, Gate and Register Levels," Proc. International Test Conference, Philadelphia (PA), November 1985.

[4] S.Gai, F.Somenzi, and E.Ulrich, "Advanced Techniques for Concurrent Multilevel Simulation," Proc. IEEE Int. Conf. on CAD, Santa Clara, CA, Nov. 1986, pp. 334-337.

[5] M. Abramovici, M.A. Breuer, K. Kumar, "Concurrent Fault Simulation and Functional Level Modeling," Proc. 14th Design Automation Conference, June 1977, pp. 128-137.

[6] C.Lo, H.N.Nham, and A.K.Bose, "Algorithms for an Advanced Fault Simulation System in MOTIS," IEEE Trans. on CAD/ICAS, Vol. CAD-6, March 1987, pp. 232-240.

[7] F. Brglez, H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran," special session on ATPG and fault simulation, Proc. 1985 IEEE Int. Symp. Circuits and Systems, Kyoto, Japan, June 5-7, 1985.

[8] E.B.Eichelberger and T.W.Williams, "A Logic Design Structure for LSI Testability," Proc. 14th Design Automation Conference, June 1977, pp. 462-468.