Comparing Different Fault Models Using VERIFY[1]

Volkmar Sieh, Oliver Tschäche, Frank Balbach
Institut für Mathematische Maschinen und Datenverarbeitung (IMMD) III,
Universität Erlangen-Nürnberg, Martensstr. 3, 91058 Erlangen, Germany
{sieh,tschaeche,balbach}@immd3.informatik.uni-erlangen.de

Abstract

*Fault injection is a widely used method for evaluating dependable systems. The intention of this paper is to compare typical fault models used for fault injection regarding their accuracy in predicting the reliability of the system. For this purpose, we set up an experiment by injecting faults in a VHDL model of the DP32-processor at gate-level, pin-level and at register-transfer level. The experiment has been performed by using the simulation based fault injector VERIFY (**V**HDL-based **E**valuation of **R**eliability by **I**njecting **F**aults efficientl**Y**). The differences in the predicted mean failure rate of the system will be shown and a comparison of system's behavior after fault injection will be presented. The results demonstrate that faults have to be injected at least at gate-level in order to represent the correct timing behavior of the digital system after the fault has been injected and help the designer of a dependable system to find the weak components.*

1. Introduction

More and more aspects of our everyday life are controlled by digital systems, where some of the services they provide are indispensable or even safety-critical. It is, therefore, necessary to evaluate all aspects of dependability of these systems. A common method to perform this evaluation is the injection of faults during runtime. For this purpose, several mechanisms for fault injection have been developed which are based on different fault models (see Section 2). (The widely used approach of injecting only permanent stuck-at faults for test-pattern generation will not be discussed in this paper.) Until now it has been an open question to what extend the results of fault injection can be compared when different fault models are used. This paper presents a comparison of several fault models by injecting faults at gate-level, pin-level and by flipping bits in internal registers.

The detailed knowledge of system's reaction over the time after injecting the fault is essential for the designer of a safety-critical digital system in order to identify and improve weak components. In addition to the comparison of the basic

1. This work is supported by the Deutsche Forschungsgemeinschaft as part of SFB 182 and Project number Da365/2-1

parameters of dependability, e.g. the mean failure rate of the system or the probability of a fault leading to a failure, we will take a closer look at the behavior of the system after injecting faults of the three fault models. For this purpose, we instrumented a set of fault injection experiments using the simulation based fault injector **VERIFY (V**HDL-based **E**valuation of **R**eliability by **I**njecting **F**aults efficientl**Y)**, which is based on an extension of the widely used hardware description language VHDL.

The rest of the paper is organized as follows: The following Section 2 gives an overview about related research. In Section 3 we present the basic concepts of describing faults and performing an experiment with VERIFY. The fault model and the processor model of the experimental study using a VHDL-model of the DP-32 processor, is presented in Section 4. The results of the simulation experiments and the comparison of three different fault models is presented in Section 5. A summary of the results is given in Section 6.

2. Related Work

The impacts of transient, permanent and intermittent faults have been investigated by several researchers at several abstraction levels of system- and fault-description. It has been shown that more than 85% of computer failures are due to transient problems [16][27]. Therefore, lots of effort have been made to investigate the impact of transient faults to the system. Several approaches towards this goal can be distinguished.

Injecting faults at the physical level has been done by either stressing the hardware with environmental parameters or by modification of the pin-level values. The first method has been used by Karlsson and Gunneflo by inducing transient soft errors with heavy-ion radiation to several processors [11][20]. This method has also been applied and compared with fault injection by electromagnetic interference to validate the MARS-system [19]. The second approach to inject faults at the physical level is the use of pin-level fault injectors, where the values of temporary patterns of several pins of an IC are under control of external devices which determine the time and duration of injection [1][22]. Whereas the latter method enables reproducibility of the results due to the ability to control all parameters (i.e. location, time and duration of a given fault), inducing soft errors corresponds more to the real physical nature of the faults. In addition to this, the current trend of integrating more and more components on-chip makes it difficult for pin-level fault injection to cover the internal faults adequately. Both approaches for the injection at the physical level tend to have a high overhead in hardware and are only feasible after the system has already been produced at least in the prototype version. It is therefore not possible to evaluate the system's reliability during early design phases.

Software implemented fault injection (SWIFI) also needs the physical hardware to inject faults. Several research groups have developed powerful tools to inject faults by software. Segall et. al. made one of the early approaches with a tool called FIAT [3][25], where the task's memory image can be corrupted during runtime. FERRARI, which was developed by Kanawati et al. [17] uses the ptrace function of UNIX to allow transient fault injection by corruption of a process's

memory image and by insertion of software trap instructions. Kao et al. proposed a tool named FINE which is able to inject faults by using a software monitor to trace the control flow [18]. Several other examples for software implemented fault injection into distributed systems are DOCTOR by Han et al. [13], Xception by Carreira et al. [5] and EFA by Echtle et al. [9]. As already mentioned, all of the approaches given above need access to at least a prototype of the hardware for which the effects of faults have to be examined. Another drawback of SWIFI is the fact, that they cover only a very small subset of all possible processor faults, which makes it difficult to evaluate the reliability of the system.

The third group of tools which have been developed for fault injections covers the simulation based approach. The major advantage of simulations based fault injection is the observability of all components, which have been modelled. Therefore, this approach enables the evaluation of dependability because it covers almost all levels of abstraction. The other benefit of simulation based fault injection is the ability to obtain the values for reliability already in the design phase of the system because the physical hardware is not needed for this method. One of the first approaches of run-time injection has been presented by Czeck and Siewiorek [8] who examined error propagation by injecting faults in a VERILOG-model of an IBM RT PC. They injected almost 19000 transient gate-level faults to understand fault manifestation and error propagation. In order to avoid massive overhead in simulation time, they chose about 10 locations for transient stuck-line fault with a duration of 1 machine cycle. Although this approach provided valuable results for understanding fault manifestations, it can not be used to determine reliability parameters like mean time to failure. Choi et al. [4] injected transient faults in a model of a jet-engine controller by using the mixed-mode simulator SPLICE. Another mixed-mode fault simulation approach has been presented by Cha [6], where transient gate-level faults have been injected by using a combination of a timing fault simulator (TIFAS) and zero-delay parallel fault simulator TPROOVES to speed up the simulation time. REACT [7], DEPEND [10] and SIMPAR [14] are tools which allow building up and evaluating models of dependable computing systems at system-level. Because of the enormous overhead of the simulation engines used, these tools are unmanageable for evaluating models at gate-level.

After standardization the hardware description language VHDL [28] became more and more attractive and is nowadays widely used to develop digital systems, Rimén et al. [23] proposed a general approach to fault injection in VHDL models. They identified two different categories of fault injection techniques in combination with VHDL: modification of the VHDL-model and the use of built-in commands of the VHDL simulator. The first category can be again subdivided in a *saboteur*-based technique, where components for fault injection are added and in a *mutant*-based technique. For the latter approach, regular components will be exchanged by so called mutants, which behave identically to the original except for the time of fault injection. The second category of fault injection techniques is the manipulation of variables and signals of the model during simulation time by using built-in command of the simulator. The research group developed a tool named MEFISTO, which covers the fault injection techniques of both categories [15][24]. The drawback of using mutants as described by the research group is a

huge amount of overhead for system evaluation as these mutants are static and the model has to be recompiled for each of the experiments.

Johnson et al. developed ADEPT [21], which is an approach for evaluating dependability aspects of a system using VHDL and Colored Petri Nets. By combining the approaches they facilitate an analytical and simulation based evaluation of system's parameter. Due to this combination, the method implemented with ADEPT fits very well for evaluation at system-level but analyzing a processor at gate-level with hundreds of thousands gates seems to be impossible with this approach.

An approach for combining software implemented fault injection with the simulation based approach has been presented by Sieh et al. [26]. The status of a MC88100 RISC processor has been transferred to a VHDL model of the processor using the ptrace function. After fault injection in the VHDL model, the status has been written back to the physical processor. Like in FERRARI the drawback of this approach is limited accessibility to the status of the processor using the ptrace function.

Rimén et al. [24] compared the behavior of a system after injecting faults at pin-level with the system's behavior after flipping bits in internal registers and latches.

Our approach extends this work by comparing these fault models with the gate-level fault model which is closer to the hardware.

3. Fault injection using VERIFY

VHDL has been established during the last decade as one of the most important hardware description languages for integrated digital circuits. An increasing number of manufacturers use this language for their system's design because the development of digital circuits is supported starting from high-level descriptions down to the generation of net-lists of the gate-level components. Simulation is used at all levels of abstraction in order to support the implementation of the digital system. Therefore, simulation is an essential feature for checking the functional and temporal behavior already during the early design phases. So far, it has not been foreseen in VHDL to directly support the checking of the reliability parameters of the system during the design phase.

The development of dependable systems does not only require the validation of temporal and functional behavior but also the ability to validate the dependability features. For this purpose we came to the conclusion that the type of faults, their mean time between occurrence and their duration should be an integral part of the description of each behavioral component of the model. In order to demonstrate the feasibility of this approach, we developed the VERIFY tool. It allows the unified description of the fault free behavior as well as the component's behavior after one of the faults correlated with this component has been activated. The natural way in VHDL of exchanging information with a component is by signals. Therefore, we use the concept of signals to describe the faults correlated to a component and at the same time have an interface for the simulator to activate the fault for a given time. Each of the possible faults known for the component can be described by a separated signal. Then, the behavioral description of the component has to be extended by the actions correlated to the faults.

This approach enables the manufacturers which provide the design libraries, i.e.

the AND-gates, OR-gates and other basic behavior-described components, to express their knowledge of fault occurrences in these components. By extending the models by their fault behavior, a simulator can be used in an early design phase to evaluate the reliability of dependable systems, to investigate the manifestation of faults and to compare several design alternatives regarding the overhead and benefit of fault tolerance mechanisms. It should be noted that the parameters of the faults, i.e. frequency and duration, can easily be adjusted according to the environment the dependable system will be used (e.g. space-mission systems, controllers for nuclear power plants, etc.), by exchanging the design library modules.

After the faults, their parameters and their behavior have been described for the behavioral components of the VHDL-model, the simulator can inject the faults during simulation time. There are two basic alternatives to make these fault injection signals (FIS) and their parameters visible for the simulator: including signals in the entity declaration or keeping the FIS transparent to other components. In the first alternative the FIS would be declared as VHDL-ports of the component's entity. This would require to make the FIS of all behavioral components of a digital circuit visible to the testbed, which is the highest level in the description hierarchy. For each of the FIS, there would have to be a "path" through all levels of hierarchy to its behavioral component. Several problems would arise with this kind of implementation:

- The ability of describing a complex model hierarchically is an integrated part of the philosophy of modelling with VHDL. The principle of encapsulating component-specific data (and the fault parameter fulfill these criteria) would be violated. If the behavior of a component in case of faults would have to be adjusted to a new environment, the modeler would be eventually forced to redesign all structural components containing this behavioral component.
- As the fault parameters would also have to be visible outside the correlated component, they would have to be described at the highest level of hierarchy, i.e. the testbed in order to determine the time and duration of injection. The description of faults would therefore be distributed over several components.

To avoid these problems, we chose the second alternative, i.e. keeping the complete fault description of one component transparent to all other components. For this purpose, we introduce a new signal type. In our approach, the FIS are described like internal signals with the extension of two additional parameters: the mean time between occurrence of the fault and its mean duration. The encapsulation of component-specific data is therefore ensured and there is no need to change the description of structural components if the behavior or the parameters of faults in internal entities are changed. Figure 1 gives an example of a VHDL description of a NOT-gate which has been extended by its fault description.

The bold typed lines show the standard VHDL description of the gate and the normal weighted lines show one possibility of extending the description of the NOT-gate with its fault behavior and the corresponding parameters. As it can be seen, the entity declaration and, therefore, the interface with other components need not be modified. For this demonstration, we used the widely accepted stuck-at fault model but it should be noted that any other fault behavior can also be described using this technique. As it can be seen from the behavioral description

```
ENTITY not_gate IS
     PORT (    input:          IN        bit;
               output:         OUT       bit);
END not_gate;


ARCHITECTURE behaviour OF not_gate IS
     SIGNAL i_sa0: BOOLEAN INTERVAL 10000 h DURATION 5 ns;
     SIGNAL i_sa1: BOOLEAN INTERVAL 15000 h DURATION 5 ns;
     SIGNAL o_sa0: BOOLEAN INTERVAL 20000 h DURATION 5 ns;
     SIGNAL o_sa1: BOOLEAN INTERVAL 30000 h DURATION 5 ns;
BEGIN
     PROCESS (input, i_sa0, i_sa1, o_sa0, o_sa1)
     BEGIN
          IF i_sa0 OR o_sa1 THEN
               output <= '1';
          ELSIF i_sa1 OR o_sa0 THEN
               output <= '0';
          ELSE
               output <= NOT input AFTER 10 ns;
          END IF;
     END PROCESS;
END behaviour;
```

Figure 1 Example code of a NOT-gate

the signal `i_sa0` models the case, where a stuck-at-0 fault occurs at the input of the NOT-gate. The mean time between the occurrence of this modelled fault is given as 10000 hours and its mean duration is 5 ns.

For the VERIFY-tool we developed a compiler which is able to handle the described extensions to VHDL and a simulator for running the fault injection experiments. When the compiler translates the VHDL-source code, the FIS are extracted automatically and are supplied for the simulator which is linked to the executable. Using this technique, the simulator has access to all fault parameters. Figure 2 gives an overview of the different phases and modules of VERIFY.

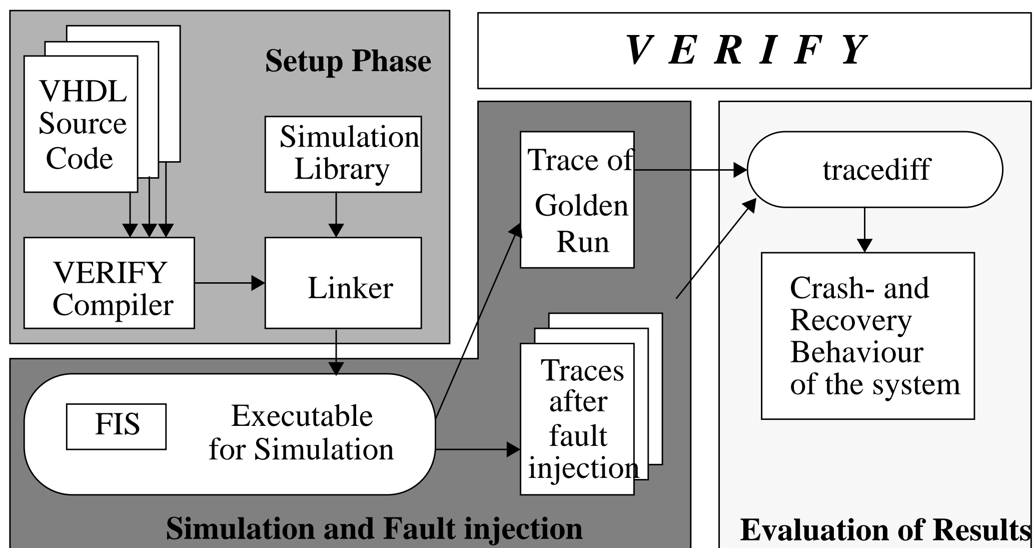

Figure 2 Overview of the VERIFY fault injection tool

The new technique implemented by VERIFY enables a completely automated evaluation of system dependability features. For starting an experiment, the user specifies the simulation time and the number of faults k which should be injected. Each experiment consists of one fault free run (golden run) and k runs where for each run exactly one fault is injected.

Once the experiment has been started, the simulator injects the required number of faults without any user interaction. Each time a fault has to be injected, the simulator determines automatically the type, location, time of occurrence and duration of the fault according to the fault descriptions in the model. The probability of a fault described in the model for being injected is inverse proportional to its mean time between occurrence. The dependability of the complete system can be evaluated by injecting several thousands of faults within a simulation time which is representative for the service the system has to provide.

During the fault injection experiments a trace of all signal values is logged for the golden run and for the time after a fault has been injected. In order to speed up the simulation time, the experiments are carried out by a technique called *multi-threaded fault injection* described by Güthoff and Sieh in [12]. In addition to this, our design goal of efficiency for the simulation has been reached by avoiding the generation of any additional events during the time the fault is not activated by the simulator. The traces produced by the simulator are evaluated by a tool called tracediff which enables the determination of the propagation of errors and of the probability of system crash, recovery and the mean recovery time.

4. Experimental study

In order to compare the fault models, we set up an experiment using a VHDL-model of the DP32-processor. In the following two subsections, we give an summary of the basic characteristics of the DP32-processor and present the fault models for which fault injection experiments have been performed. A detailed description about the DP32-processor model can be found in [2].

4.1 The DP32 processor

The DP32 is a simple 32-bit RISC processor which has been chosen to be able to compare our results with the fault injection experiments performed by Jenn et. al. using the MEFISTO tool [15]. The VHDL model of the processor presented in [2] is at RTL-level and has been modified so that the Synopsys™ synthesis tool automatically generates a gate-level VHDL model.

The A/D bus is used to fetch instructions and to transfer data between registers and memory. The actions of the DP32 are controlled by a single finite state machine (FSM) in the control block of Figure 3. The instruction set of the DP32 includes load/store, flow control, logic and arithmetic instructions. The model used for our experiments does not support instructions for multiplication and division. The register file of the original model supports 256 registers, our model is reduced to 8 registers. Further minor changes make the model fit to be processed by the Synopsys™ synthesis tool.

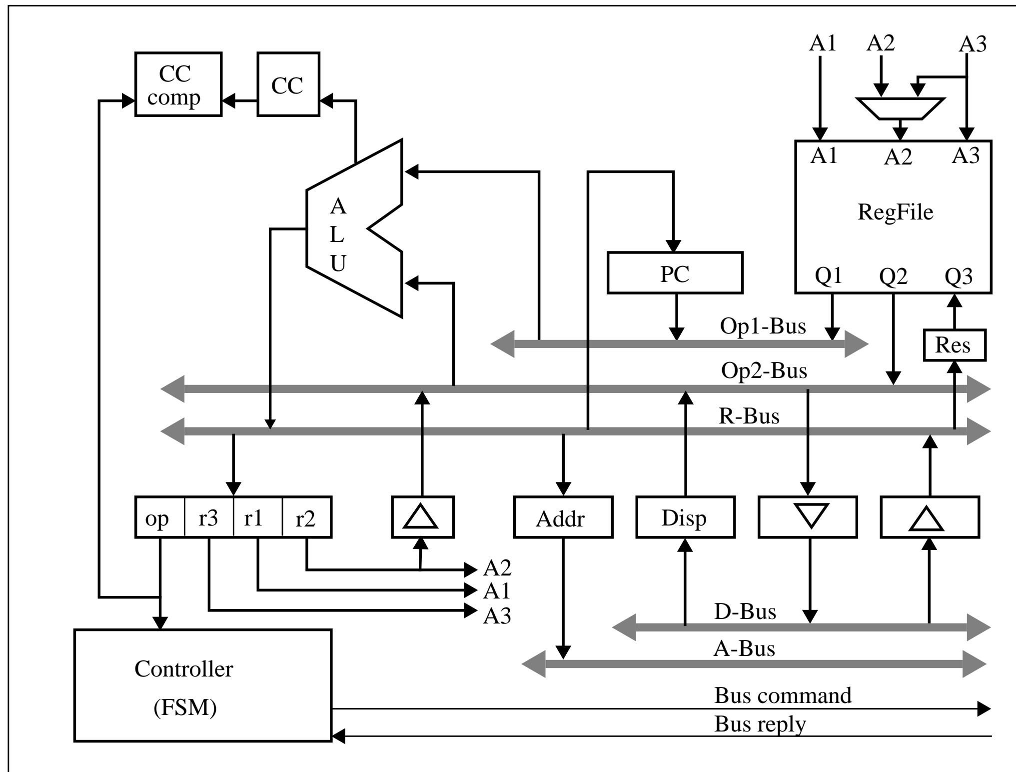

Figure 3 Structure of the DP32 processor

The fault injection experiments are performed with two different gate-level models of the DP32. The models are automatically generated by the synthesis tool using different cell libraries. The library for the simple model includes only the fundamental gates: an *AND* and *OR* gate with two inputs, *Inverter*, *Tristate Driver*, *D-flipflop* and *D-latch*. Due to this simple library the generated gate-level model of the DP32 includes 32,4% more gates than the advanced gate-level model generated with an advanced library (3710 gates vs. 2801 gates). The advanced library is a superset of the simple library. The advanced library includes the additional cells for *NAND* gates, *NOR* gates, *XOR* gates, *Multiplexer* and *Full Adder*. The logic gates are available in versions with 2, 3 and 4 bit inputs.

During the simulation of both gate-level models the test program of Figure 4 is processed. Firstly, it resets register r0 to zero. Then, it increments r2 starting at 0 until r2 is equal to 10, where it restarts at r2 equal to zero. The simulation of one cycle (counting from 0 to 10) needs 6 µs at 20 ns clock cycle length.

```
            initr0
start:      addq(r2, r0, 0)       ! r2 := 0
loop:       sta(r2, counter)      ! counter := r2
            addq(r2, r2, 1)       ! increment r2
            subq(r1, r2, 10)      ! if r2 = 10 then
            brzq(start)           ! restart
            braq(loop)            ! else next loop
counter:    data(0)
```

Figure 4 Test program

4.2 Fault models

For our experiments we chose three different fault models: the well known stuck-at-x fault model at gate-level, bit-flips in registers of the register-transfer level and stuck-at-x at pin-level of the processor.

- The stuck-at-x fault model at gate-level is widely used in conjunction with test pattern generation. We extended the customary approach of allowing stuck-at-0 or stuck-at-1 only at output signals of the components by adding the same possibility of faults for input signals. If the output of a gate drives one signal which will be used as an input for several other gates, allowing faults only at the output of the gate would always affect all components connected with the affected signal. An example of this fault model has already been presented in Figure 1, which shows the extension of a NOT gate.
- For the second fault model we chose the single bit-flip model in the internal registers and latches of the DP32 processor. This model is used by nearly all tools which are based on the approach of software implemented fault injection (see Section 2). For the experiments the faults have been injected uniformly distributed over all bits of internal registers including the bits of the finite state machines of the processor.
- The third fault model we chose for our experiments is the injection of faults at the pin-level of the DP32 processor. Like in the gate-level stuck-at fault model, which is the most detailed one, we chose single stuck-at-x faults uniformly distributed over all pins. For this purpose we had to extend the testbed for our experiments with a socket which connects the DP32 processor with the main memory via the address- and data-bus and its control lines (see Figure 5). The location of the faults to inject is restricted to the socket and, therefore, to the pins of the processor. With this fault model, all signals going to or coming from the processor can be corrupted.

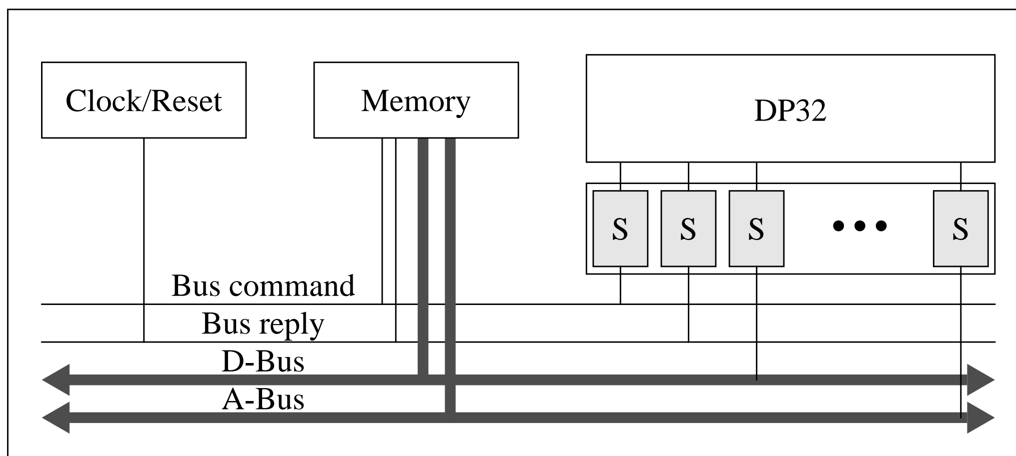

Figure 5 Testbed for pin-level fault injection

In all experiments the same mean frequency of occurrence and mean duration time is used. It would easily be possible to adjust these values if more realistic

rates or duration times are known.

It should be noted, that the general approach of the VERIFY tool allows an even more detailed fault model than the gate-level stuck-at-x used for our experiments.

5. Evaluation of results

Four experiments have been performed in order to evaluate the influence of different parameters on the results of fault injection. As already mentioned, our major goal has been the comparison of the fault models presented in the previous section. In addition to this, we intent to show for the most detailed fault-model (gate-level) the influence of a different implementation of the DP32 processor in hardware on the behavior of the system after fault injection.

5.1 Gate-level and fault models

We therefore performed four experiments which are described in the following:

- In the first experiment (called "*simple*") the simple gate-level model of the DP32 was used (see section 4.1) and only internal stuck-at faults were injected at the input and output ports of each gate.
- The evaluation of the influence of different hardware realizations on the behavior of the system after fault injection were measured by comparing *simple* with the second experiment called "*advanced*". For this purpose we used the advanced gate-level model described in section 4.1. The fault model which has been used was the same as in the *simple* experiment, i.e. stuck-at faults at every input and output of every gate inside the processor.
- For the third experiment (called "*bit-flip*") the advanced gate-level model of the DP32 was used in conjunction with bit-flip faults at all internal registers. The advanced gate-level model of the processor has been favored over the simple because of a shorter simulation time due to a fewer number of gates (see section 4.1).
- In the last experiment ("*pin-level*") stuck-at faults were injected into the socket of the processor. As no faults have been injected inside the DP32, any level of description of the DP32 could be used as a model of the CPU.

Every experiment consisted of 1000 different test runs, where for each run one single fault has been injected. The behavior of the system was observed for $2\mu s$ after injecting the fault. The mean duration of the fault was assumed to be 20ns. The time of the occurrence of the fault was chosen according to the strategy described in Section 3. All experiments have in common that no faults have been injected into the clock, reset and the RAM components of the testbed.

5.2 Extracted information

As we use simulation based fault injection for all four experiments, it is possible to observe all signals and all states of the system under test (processor, clock/reset, RAM) by automatically recording all signals and state changes in a trace file during simulation time. In the following a simple classification scheme is shown for evaluating the results of the four experiments.

After executing the golden run and all test runs of one experiment the trace files of the test runs were compared against the golden run. Each comparison may show one of the following results.

- The first possibility is that the injected fault crashes the system. The processor gets into an erroneous state and is not able to recover within the observation time. The fault in the internal component or at one of the input/output pins of the processor leads to a failure of the system.
- Another class of result is defined by the fact, that all incorrect signals return to correct values during the observation time on condition that the system under test changes its behavior after fault injection. In this case, the system was able to mask the fault or to recover from the injected fault.
- As a third possibility the comparison may show that the golden run and the test run did not differ although a fault has been injected. The fault was no “real fault” (e.g. stuck-at-0 fault when the signal was already ‘0’) or was masked by the next gate immediately (e.g. stuck-at-0/1 at one input of an *AND*-gate where the other input is ‘0’). These results are omitted from the statistics shown below.

5.3 Results

The diagrams below show how the system reacts to the faults injected. The graph $G(t)$ indicates the percentage of injected faults which caused faulty states in the system at a given time t after the fault has been deactivated. The function $1 - G(t)$ is the recovery time distribution ($p(T_r < t)$ with recovery time T_r). In addition to the recovery time distribution, the diagrams show also the probability that the system is not able to recover from an injected fault within the observation interval T. This probability is $G(T)$.

Each diagram on the right side is a magnification of the first two nanoseconds of the graph shown on the left side.

Figure 6 gives an overview of all four experiments.

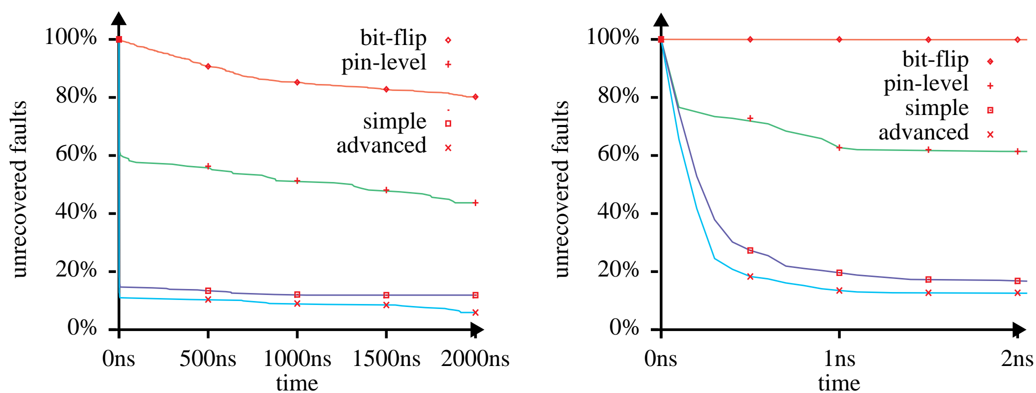

Figure 6 Experiment overview

It can be seen that the reaction of the system heavily depends on the type of the injected fault and on the used gate-level model. E.g.: faulty states induced by internal stuck-at faults (“simple”, “advanced”) and external stuck-at faults (“pin-level”) disappear much faster than erroneous states caused by bit-flips (“bit-flip”).

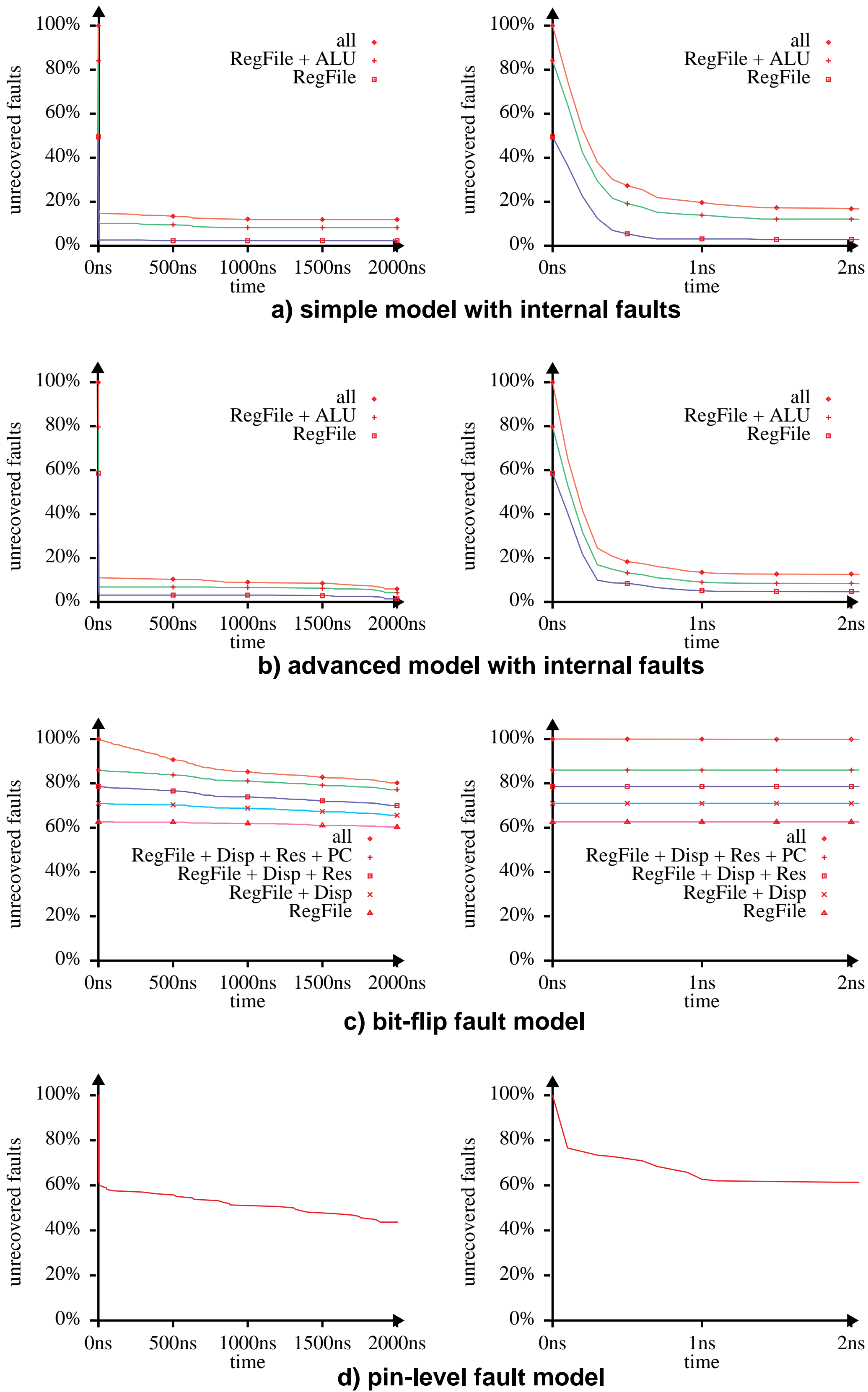

Figure 7 Detailed Experiment Analysis

In the diagrams of Figure 7a-7d the results of the four experiments are presented in greater detail by showing the percentage of unrecovered faults for the several components, in which faults have been injected. The function $G(t)$ is separated into functions $G_{c_i}(t)$ for each component c_i of the modelled system. It holds

$$G(t) = \sum_{c_i} G_{c_i}(t).$$

For each diagram the components are sorted ($c_o \geq c_1 \geq \ldots \geq c_n$) so that

$$G_{c_o}(0) \geq G_{c_1}(0) \geq \ldots \geq G_{c_n}(0).$$

The functions are stacked by order to be visible within the diagrams. So each graph called “$c_0 + c_1 + \ldots + c_i$“ (e.g. “RegFile + Disp + Res”) represents the function

$$\sum_{k=0}^{i} G_{c_k}(t).$$

Graph “all” represents $G(t)$. Only the most important components are given.

It can be seen that due to the fact that most gates (and most bits) are located in the register file, most of all stuck-at (and bit-flips) occurred in this component. Faults within the ALU can only be injected with the gate-level stuck-at fault model. The results show that the recovery time distribution function depends on the component where the fault was injected. Some components (e.g. the ALU) recover more slowly whereas other components (like the register file) are likely to be fault free more early after fault injection (e.g. Figure 7a, “ALU” and “RegFile”).

Figure 8 compares the percentage of faults which may occur within a component and the percentage of faults within a component which lead to a failure of the system, i.e. for which the system did not recover from.

As one can see about 60% of all faults occurred within the register file but only

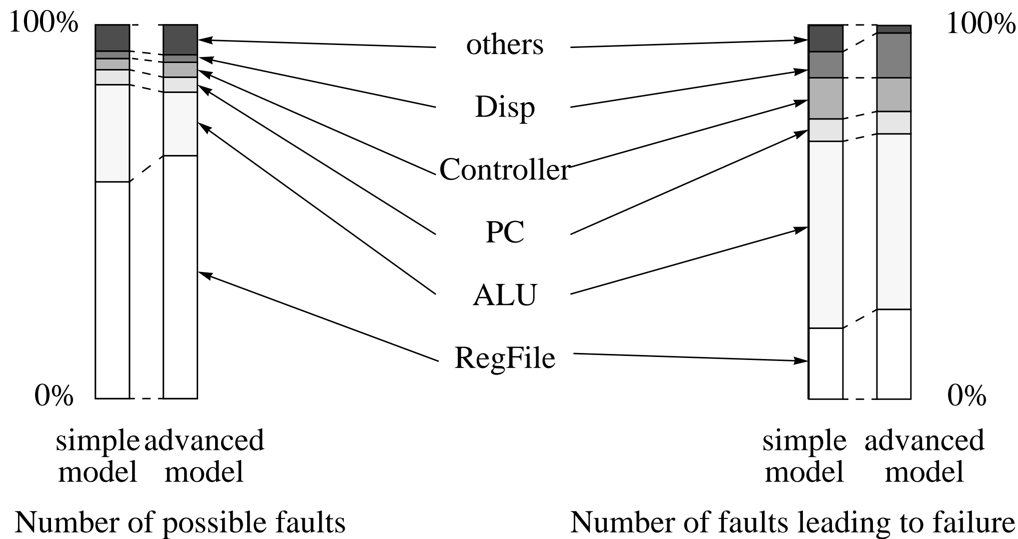

Figure 8 Comparison fault and failure propabilities

about 20% of the system failures are caused by faults within this component. On the other hand, only about 25% of all faults are likely to occur inside of the ALU but they cause more than 50% of all failures of the system under investigation. A system designer can take these results and improve or exchange the components where faults tend to lead to system crashes.

When injecting a fault f at different times it is possible to get an estimation of the probability $p(f)$ of this fault to lead to a failure of the system. To calculate the failure rate μ of the system it is necessary to know the rate of occurrence ($\lambda(f_1)$, $\lambda(f_2)$ to $\lambda(f_N)$) of all N types of faults.

$$\mu = \sum_{f=f_1}^{f_N} \lambda(f)p(f)$$

The arithmetic mean of all occurrence rates of faults is

$$\bar{\lambda} = \frac{1}{N}\sum_{f=f_1}^{f_N} \lambda(f)$$

If $\lambda(f_i) = const$ for all $i \in \{1\ldots N\}$, the arithmetic mean of all probabilities of a fault to lead to system failure is

$$\bar{p} = \frac{1}{N}\sum_{f=f_1}^{f_N} p(f) = \frac{\mu}{N\bar{\lambda}}$$

In our experiments we used the same rate of occurrence for all faults ($\lambda(f) = 0.01\,\text{year}^{-1}$ for all f). The number of different faults N varies because of the different gate-level and fault models. N, $\bar{p}$ and μ are calculated by VERIFY and are shown in Table 1. Column "gate-level model" and "fault model" indicate the different kind of models used.

The number of faults and the probability of a fault to lead to system failure depend heavily on the type of experiment performed. The number of faults varies between 23000 (simple gate-level model with stuck-at faults) and 140 (pin-level fault injection experiment):. The "bit-flip" experiment gives the highest probability

Table 1: Experiment summary

gate-level model	fault model	N	$\bar{p}$	$\mu\ [year^{-1}]$
simple	stuck-at	23000	4.5%	10.3
advanced	stuck-at	19000	2.0%	3.8
advanced	bit-flip	420	79.0%	3.3
advanced	pin-level	140	6.8%	0.097

of a fault to become the cause of a system failure (79%). The very same hardware model used with a different fault model ("stuck-at") shows only 2% as the probability for a fault to cause a system failure. Even when using the very same behavior model (compiled with slightly different cell libraries) and using the very same

fault model ("stuck-at") the fault injection experiments may show different results ($N = 23000$, $\bar{p} = 4.5\%$ versus $N = 19000$, $\bar{p} = 2\%$).

So when measuring $\bar{p}$ using fault injection this number has to be multiplied by N and $\bar{\lambda}$ to get an approximation of μ. But even when using a scaling factor it is possible that two systems with the same failure rate μ behave different in the presence of faults because of different shapes of the recovery time distribution function.

6. Conclusion

This papers presents the results of several evaluations of reliability of a computer system. The evaluations are done by simulation based fault injection using VERIFY. We experimented with different fault models and different gate-level implementations of one system.

It has been shown that the experiments are sensitive to changes of the fault model and, even, to very slight changes of the model of the system (replacing the simple cell library by the advanced). If evaluating changes in the hardware, one fault model could indicate an improvement of reliability while another fault model indicates a decrease in reliability. So, valuing improvements which should increase the fault tolerance of a system should be done very carefully.

It has been shown that the faults should be described at least at gate-level in order to get meaningful results from fault injection experiments. The overhead for the designer of the system is very low, because in general he only has to deal with a cell-library of gate-level components which will be delivered by the manufacturer providing the library.

Most results of fault injection experiments presented so far only dealt with the probability of a system to recover from a specific fault within a given observation interval. As it can be seen from the diagrams presented this probability is a function of the length of the observation time and of the type of fault injected.

In addition, the recovery time distribution of a system is an important variable classifying the reliability of systems, especially of real time systems. If it is only evaluated whether the system runs into a faulty or correct state after the fault has been injected important information to improve components in a fault tolerant way will be missed. Since most tools only evaluate at the end of a test program whether the function is fulfilled or not, but do not have any information about the impact of faults during the observation interval, they miss to get even a tendency of the shape of the recovery time distribution.

Finally, the paper shows that the results of an evaluation equally depend on the probability of a fault crashing the system and on its rate. So, if the probability of system crashes for each fault is evaluated, then, evaluating the reliability of a system in simulation based fault injection experiments has to consider the rate of each fault. If there is e.g. a fault which crashes the system with a probability of 99% but this fault rarely happens, then, the evaluation of this probability is useless. Therefore, a tool evaluating the reliability of systems must support an interface to set the rate of the injected faults.

7. References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, D. Powell: "Fault injection for dependability validation: a methodology and some applications". *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, February 1990, pp. 166-182

[2] P. Ashenden: “The VHDL-cookbook”, University of Adelaide, South Australia, Technical Report 1990

[3] J. Barton, E. Czeck, Z. Segall, D. Siewiorek: "Fault Injection Experiments using FIAT". *IEEE Transaction on Computers,* Vol. 39, No. 4, April 1990, pp. 575-582.

[4] G. Choi, R. Iyer, V. Carreno: "Simulated fault injection: A methodology to evaluate fault tolerant microprocessor architectures". *IEEE Trans. Reliability*, Vol. 39, No. 4, pp. 486-490, October 1990.

[5] J. Carreira, H. Madeira, J. G. Silva: "Xception: Software Fault Injection and Monitoring in Processor Functional Units" *Preprints of the DCCA-5, Working Conference on Dependable Computing for Critical Applications*, Urbana Champaign , USA, Beckman Institute, September 27-29, 1995, pp. 135-149.

[6] H. Cha, E. Rudnick, G. Choi, J. Patel, R. Iyer "A Fast and Accurate Gate-Level Transient Fault Simulation Environment", *Proc. 23rd Symp. on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993, pp. 310-319.

[7] J. A. Clark, D. K. Pradhan, “REACT: An Integrated Tool for the Design of Dependable Computer Systems”, in “Foundations of Dependable Computing, Models and Frameworks for Dpendable Systems”, G. M. Koob, C. G. Lau (ed.), Kluwer, pp. 169-192, 1994.

[8] E. Czeck, D. Siewiorek "Effects of Transient Gate-Level Faults on Program Behavior", *Proc. 20th Symp. on Fault Tolerant Computing (FTCS-20)*, Newcastle Upon Tyne, June 1990, pp. 236-243.

[9] K. Echtle, M. Leu "The EFA Fault Injector for Fault Tolerant Distributred System Testing", *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Amherst (MA), USA, pp. 28-35, 1992.

[10] K. K. Goswami, R. K. Iyer, “DEPEND: A Design Environment for Prediction and Evaluation of System Dependability”, *Proc. 9th Digital Avionics Systems Conference*, October 1990.

[11] U. Gunneflo, J. Karlsson, and J. Torin: "Evaluation of error detection schemes using fault injection by heavy-ion radiation.": *Proc. 19th Symp. on Fault-Tolerant Computing (FTCS-19)*, Chicago, Illinois, 21-23, June 1989, pp 340-347.

[12] J. Güthoff, V. Sieh: “Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method”, *Proc. 25th Symp. on Fault-Tolerant Computing (FTCS-25)*, Pasadena, California, June 1995, pp. 196-206.

[13] S. Han, H. A. Rosenberg, K. G. Shin: „*DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment*“, Technical Report Univ. of Michigan, December 1993

[14] A. Hein, K. K. Goswami, “Combined Performance and Dependability Evaluation with Conjoint Simulation”, *Proc. of 7th European Simulation Symposium*, Erlangen-Nuremberg, Oct. 26-28, 1995, pp. 365-369.

[15] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson: "Fault Injection into VHDL Models: The MEFISTO Tool". *Proc. 24th Symp. on Fault Tolerant Computing, (FTCS-24)*, IEEE, Austin, Texas, USA, pp. 66-75, 1994

[16] R.H. Iyer, D. Rosetti: “A measurement based model for workload-dependance of CPU-errors”, *IEEE Transactions on Computers*, vol C-35, 1986 Jun, pp 511-519

[17] G. A. Kanawati, N. A. Kanawati, J. A. Abraham: "FERRARI: A Tool for the Validation of System Dependability Properties", *Proc. 22th Symp. on Fault-Tolerant Computing (FTCS-22)*, Boston, Massachusetts, July 8-10, 1992, pp. 336-344.

[18] W. Kao, R. K. Iyer, D. Tang: "FINE: A Fault Injection and Monitor Environment for Tracing the UNIX System Behavior under Faults", *IEEE Trans. on Soft. Eng.*, Vol. 19, No. 11, Nov. 1993, pp. 1105-1118.

[19] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, J. Reisinger: "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", *Preprints of Fifth International Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, Urbana-Champaign, Illinois, USA, September 27-29, 1995, pp. 150-161.

[20] J. Karlsson, U. Gunneflo, J. Torin: "Use of Heavy-Ion Radiation from Californium-252 for Fault Injection Experiments" in *Dependable Computing for Critical Applications*, A. Avizienis, J.-C. Laprie (eds.), in series "Dependable Computing and Fault-Tolerant Systems", Vol. 4, Springer-Verlag Wien-New York, 1991, pp. 197-212.

[21] S. Kumar, R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, “ADEPT: A Unified System Level Modeling Design Environment”, Proceedings of the 1st Annual RASSP Conference, Arlington, Virginia, pp. 114-123, August 15-18, 1994.

[22] H. Madeira, M. Rela, J. G. Silva: "RIFLE: A General Purpose Pin-Level Fault Injector", *Proc. First European Dependable Computing Conference (EDCC-1)*, Berlin, Germany, Springer Verlag, October 4-6, 1994, pp. 199-216.

[23] M. Rimén, J. Ohlsson, J. Karlsson, E. Jenn, J. Arlat: "Design Guidelines of a VHDL-based Simulation Tool for the Validation of Fault Tolerance", *Proc. 1st ESPRIT Basic Research Project PDCS-2 Open Workshop*, LAAS-CNRS, Toulouse, France, September 1993, pp. 461-483.

[24] M. Rimén, J. Ohlsson, J. Torin: "On Microprocessor Error Behavior Modeling". *Proc. IEEE 24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, Austin, Texas, USA, 1994, pp. 76-85.

[25] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, T. Lin: "FIAT — Fault Injection Based Automated Testing Environment", *Proc. 18th Symp.on Fault-Tolerant Computing (FTCS-18)*, Tokyo, Japan, IEEE CS Press, pp. 102-107, June 1988.

[26] V. Sieh, A. Pataricza, B. Sallay, W. Hohl, J. Hönig, B. Benyó, “Fault Injection Based Validation of Fault-Tolerant Multiprocessors”, *Proc. µP’94, 8th Symposium on Microcomputer and Microprocessor Applications*, TU Budapest, 1994, pp. 85-94.

[27] D. Siewiorek, R. Swarz: “The Theory and Practice of Reliable Systems Design”, 1982; Digital Equipment Corporation.

[28] "*IEEE Standard VHDL Language Reference Manual*", ANSI/IEEE Std 1076-1993, IEEE Inc., 1993