V. Sieh, O. Tschäche, F. Balbach

VHDL-based Fault Injection with VERIFY

Internal Report No.: 5/96

VHDL-based Fault Injection with VERIFY[1]

Volkmar Sieh, Oliver Tschäche, Frank Balbach

Institut für Mathematische Maschinen und Datenverarbeitung (IMMD) III,
Universität Erlangen-Nürnberg, Martensstr. 3, 91058 Erlangen, Germany
email: {sieh,ortschae,balbach}@informatik.uni-erlangen.de

Abstract **-** This paper describes a new methodology to inject transient and permanent faults in digital systems. For this purpose, the simulation based fault injector VERIFY (**V**HDL-based **E**valuation of **R**eliability by **I**njecting **F**aults efficientl**Y**) has been developed, which allows fault injection at several abstraction levels of a digital system. The combined approach of injection and analysis of the results enables the system's engineer to evaluate the reliability of the system as well as the coverage of fault tolerance mechanisms applied to the system. The approach is applied to the DP32-processor, where faults are injected at pin-level, by flipping bits in internal registers and gate-level. The results of this comparison shows, that the time to recover from a fault and the total number of faults which lead to a recovery differ significantly according to the type of fault injection. Whereas in the first 2 µs after fault injection using the bit-flip fault model more than 79% of all faults injected were still present in the system, only 4.5% remained in the processor when using a stuck-at fault model at gate-level.

Keywords **-** Fault injection, VHDL, dependable systems, transient faults, experimental analysis, system validation, fault/error models

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, e.g. aircraft controllers. It is therefore necessary to evaluate all aspects of dependability of these digital systems. Mechanisms for fault tolerance developed for those systems have to be validated as well as characteristic dependable values have to be determined. As this evaluation has to be done in an early development phase or even in the design phase, field data regarding the reliability are not available at this time of the system's life cycle. To overcome this problem, fault injection gained an increasing interest during the past decade. This paper presents a new method to describe the faults/errors in a VHDL-model of the system, inject the faults according to their frequency during simulation time and to evaluate the system's behavior in case of the faults. For this purpose, we developed a tool called **VERIFY (V**HDL-based **E**valuation of **R**eliability by **I**njecting **F**aults efficientl**Y)** which includes an extended VHDL-compiler, a simulator with automated fault injection and several evaluation tools to analyze the results.

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

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 [13][24]. Therefore, lots of effort has 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 have 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 soft errors with heavy-ion radiation to several processors [9][16]. This method has also been applied and compared with fault injection by electromagnetic interference to validate the MARS-system [14]. 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][19]. Whereas the latter method enables reproducibility of the results by the ability to control all parameters, i.e. location, time and duration of a given fault, the inducing of 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 already 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][23], where the task's memory image can be corrupted during run-time. FERRARI, which was developed by Kanawati [18] 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 proposed a tool named FINE which is able to inject faults by using a software monitor to trace the control flow [17]. In addition to this, several examples for software implemented fault injection into distributed systems are DOCTOR by Han [11], Xception by Carreira [5] and EFA by Echtle [8]. 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 an subset of an unknown size 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 evaluation of dependable systems requires fault injection during run-time. Therefore, the widely used approach of injecting only permanent stuck-at faults for test-pattern generation will not be discussed here in the paper. The major advantage of simulations based fault injection is the observability of all components, which have been modeled. 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 [7] 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 a 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 and Iyer injected transient faults in a model of a jet-engine controller by using the mixed-mode simulator SPLICE [4]. 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.

After the standardized hardware description language VHDL became more and more attractive and is nowadays widely used to develop digital systems, Rimen and Arlat proposed a general approach to fault injection in VHDL models [21]. 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 like 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 MEPHISTO, which covers the fault injection techniques of both categories [12][22]. 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.

The work presented in this paper introduces a new technique of fault description, injection and evaluation of the fault manifestation. It is based on dynamic mutants, where the fault description is an integrated part of the behavioral description of the components. For this purpose, the language of VHDL has been extended in order to be able to describe the type, rate of occurrence and mean duration time of the faults. A compiler and a simulator, which is able to inject the faults according to this description have been developed.

The rest of the paper is organized as follows: In section 2 we present the idea the new fault injection method and a description of the VERIFY tool will be given. The fault model and the processor model of an experimental study (DP32-processor) will be presented in section 3. The results of the simulation experiments which compares fault injection at three different levels of abstraction, i.e. gate-level, pin-level and RTL-level, will be presented in section 4. Section 5 summarizes the paper.

2. Fault Injection using VERIFY

VHDL has been established during the last decade to one of the most important hardware description languages for integrated digital circuits. An increasing number of manufacturers use these language for their system's design because the development of the 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 mean time between faults 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, which allows the integrated 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 the use of signals. Therefore, we used the concept of signals to be able 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 therefore be described by a separated signal. The behavioral description of the component has then to be extended by the actions correlated to the faults.

This approach enables the hardware 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, 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: include the 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. The exchange of a behavioral components due to adjustments of the fault behavior would eventually force the modeler to redesign all structural components containing the exchanged 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 de-

scription of one component transparent to all other components. For this purpose, we introduced a new signal type to the VHDL-syntax. In our approach, the FIS are described like internal signals but have the extension of two additional parameters: the mean time of the 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 higher leveled components due to an exchange of their internal entities. 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 de-

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


ARCHITECTURE behaviour OF not_gate IS
     SIGNAL    i_stuck_at_0:       BOOLEAN INTERVAL 10000 h DURATION 5 ns;
     SIGNAL    i_stuck_at_1:       BOOLEAN INTERVAL 15000 h DURATION 5 ns;
     SIGNAL    o_stuck_at_0:       BOOLEAN INTERVAL 20000 h DURATION 5 ns;
     SIGNAL    o_stuck_at_1:       BOOLEAN INTERVAL 30000 h DURATION 5 ns;
BEGIN
          PROCESS (input) BEGIN
                    IF i_stuck_at_0 OR o_stuck_at_1 THEN
                              output <= '1';
                    ELSIF i_stuck_at_1 OR o_stuck_at_0 THEN
                              output <= '0';
                    ELSE
                              output <= NOT input AFTER 10 ns;
                    END IF;
          END PROCESS;
END behaviour;
```

Figure 1 Example Code of a NOT-Gate

scription of the gate and the normal weighted lines one possibility of extending the description of the NOT-gate with its fault behavior and the corresponding parameters. As it can be seen by this example, the entity declaration and therefore the interface with other components need not be modified for describing the faults. 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 the signal named *i_stuck_at_0* denotes the case, where a stuck-at-0 fault occurs at the input of the NOT-gate. The mean time between the occurrence of this fault type 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. In the current implementation, the compiler translates the VHDL-source code to a program in the C-programming language. The FIS are extracted automatically by the compiler and supplied for the simulator which is linked to the executable. Using this technique, the simulator has access to all fault parameters described for the system. Figure 2 gives an overview of the different phases and modules of VERIFY.

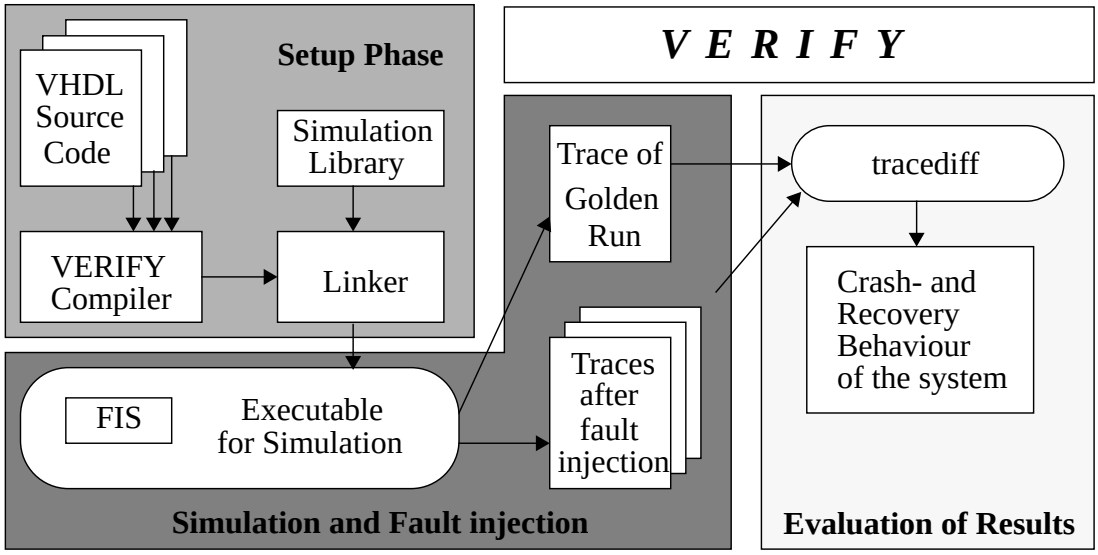


Figure 2 Overview of the VERIFY Fault Injection Tool

During the experiments of fault injection a trace of all signal values will be logged for the fault free run (golden run) and for the time after a fault has been injected. In order to speed up the simulation time, the experiments will be carried out by a technique called *multi-threaded fault injection* described by Güthoff and Sieh in [10]. In addition to this, our design goal of efficiency for the simulation has been reached by avoiding the generation of any additional events faults 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 evaluation of the propagation of errors and the evaluates the probability of system crash, recovery and the mean time needed for recovery.

The new technique implemented by VERIFY enables a completely automated evaluation of system dependability features. After the designer of the system has compiled the source code of the system's model, the execution of all fault-injection experiments is performed without any

need of interaction with the user of the tool. Once the experiments have been started, the simulator injects independently the required number of faults. The time and the location for injecting the next fault will be determined automatically by the simulator according to the weighting of the described fault intervals. This approach ensures that if a fault occurs twice as often as another one, it will be injected twice as often. Due to this reason, only the relative frequency of faults compared with all possible faults is needed to determine the time and location of the next injection. The dependability of the complete system can therefore be evaluated by injecting several thousands of faults within a simulation time which is representative for the service it has to provide.

3. Experimental Study

In order to prove the usability of the approach, it has been applied to the VHDL-model of a processor, derived from the DP32 in [2]. In the following two subsections, we give an overview about the basic characteristics of the DP32 processor and present several fault models for which fault injection experiments have been performed. A detailed description about the processor model can be found in [2].

3.1 The DP32 Processor

The DP32 is a simple 32-bit processor which has been chosen to be able to compare the results with the fault injection experiments performed by Jenn using the MEFISTO tool [12]. The following sections are related to gate level models of the DP32. Since the models shown in [2] are at a more abstract level the structural VHDL model is modified so that the Synopsys synthesis tool automatically generates a gate level VHDL model.

The DP32 is a simple 32 bit RISC processor. 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.

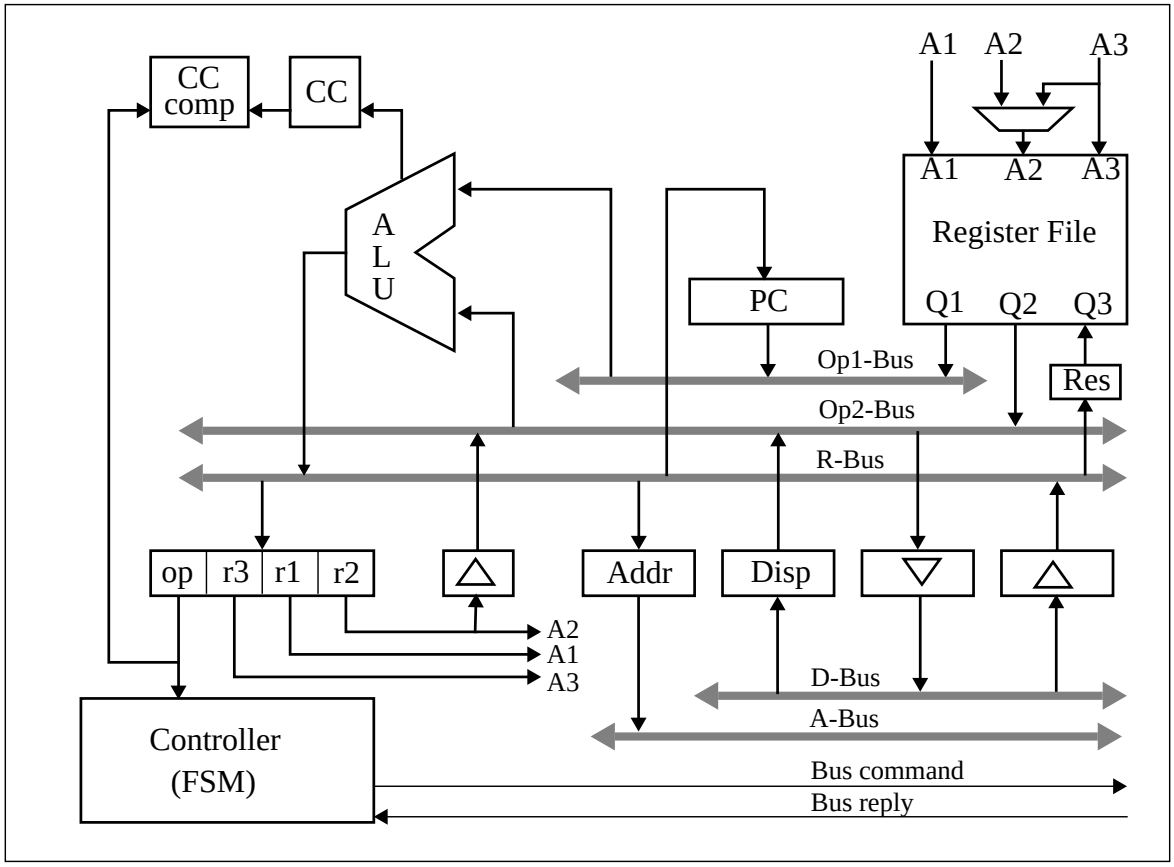


Figure 3 Structure of the DP32 Processor

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

3.2 Fault Models

As already mentioned in the previous sections, simulation enables control of almost all levels of abstraction of the modeled system. Therefore, the generation of a gate-level model of the DP32 allows the comparison of different fault injection methods. Although, this has already been done by other research groups [22], this is the first time using the same integrated tool for a comparison which shows the wide range of possible uses of VERIFY. For our experiments we chose three different fault models, i.e. 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 an input of a gate stucks at a given level (0 or 1), the gate will behave as if the signal driving this input contains this value. This extension has been chosen due to the fact, that faults (e.g. alpha-particles hitting the processor) can not only affect the output driver for a signal, but also may be located at the input of components. 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. Only the possibility of allowing additional stuck-at faults at the inputs of the components enables the injection of local faults. An example of this fault model has already been presented in Figure 1, where which shows the extension of a NOT gate.
- For the second fault model we chose the single bit-flip model in the internal registers of the DP32 processor. This model is used by nearly all tools which are based on the approach of software implemented fault injection (see survey in section 1). For the experiments the faults

have been injected uniformly distributed over all bits of internal registers including the ones of the FSMs of the processor.

- The third fault model we chose for our experiments has been 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 injected faults to inject is here restricted to the socket and, therefore, to the pins of the processor. With this fault model, all information going to or coming from the processor can be corrupted.

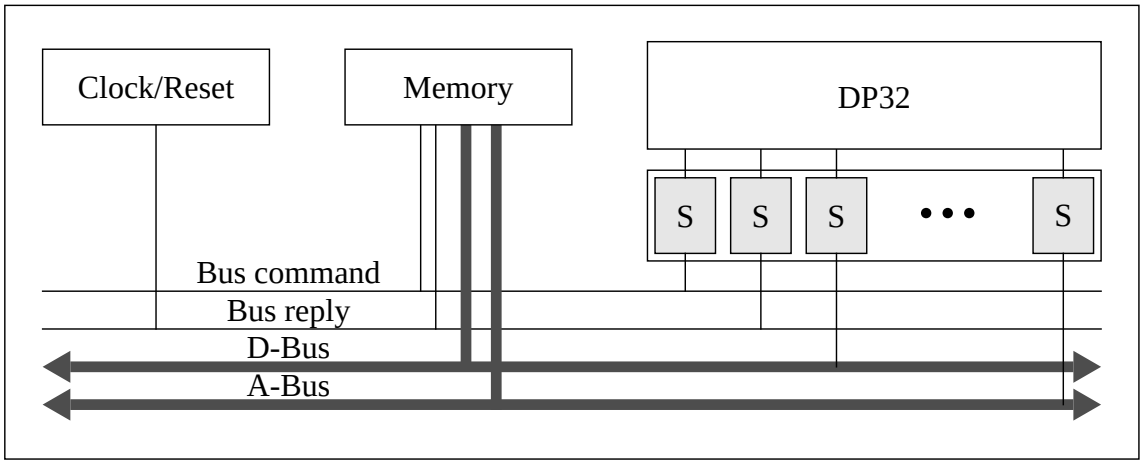


Figure 5 Testbed for Pin-Level Fault Injection

In order to abstract from the hardware technology used for an actual implementation of the processor, we used for all FIS modeled in the three experiments the same mean frequency for their occurrence, i.e. a rate of one fault in 100 years. This allows the comparison of the mean system failure rates for the three models.

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 as the reference model for our experiments. It is possible to describe a precise timing behavior of the fault as well as taking into account physical features of the used hardware technology if the information is available.

4. Evaluation of Results

Four experiments have been performed in order to evaluate the influence of different parameters for the results of fault injection. As already mentioned, our major goal for the experiments 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. We therefore performed four different experiments which will be described in the following:

- In the first experiment (called "*simple*") the simple gate level model of the DP32 was used (see section 3.1) 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 were measured by comparing *simple* with the second experiment called "*advanced*". For this purpose we used the advanced gate-level model described in section 3.1. 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 kinds of internal registers. The advanced gate-level model of the processor has been favored to the simple because of a shorter simulation time due to a fewer number of gates (see section 3.1).
- 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 duration of the fault was exponentially distributed with a mean value of 20ns. The time of the occurrence of the fault was chosen according to the strategy described in section 2. All experiments have in common that no faults have been injected into the clock, reset and the RAM components of the testbed.

As we use simulation based fault injection for all four experiments, it is possible to observe all signals and all states of the complete system under test (processor, clock/reset, RAM) by automatically recording all events 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. Every comparison may show one of the following re-

sults. 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 whole system. Another class of result is defined by the fact, that all internal states 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 completely 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.

The diagrams below show how the system reacts to the fault injection. The graph indicates the rate of injected faults which caused faulty states in the system at a given time after the fault has been deactivated. The 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. 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. Faulty states induced by internal and external (pin-level) stuck-at faults disappear faster than erroneous states caused by bit-flips.

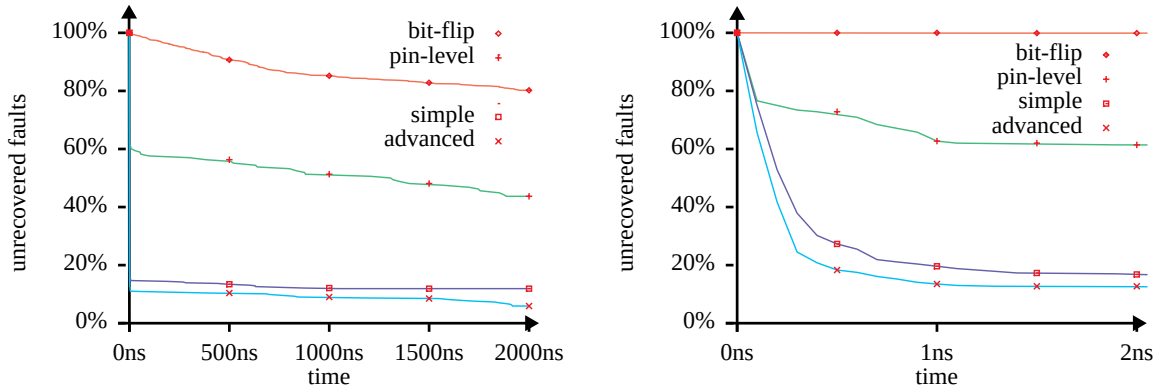
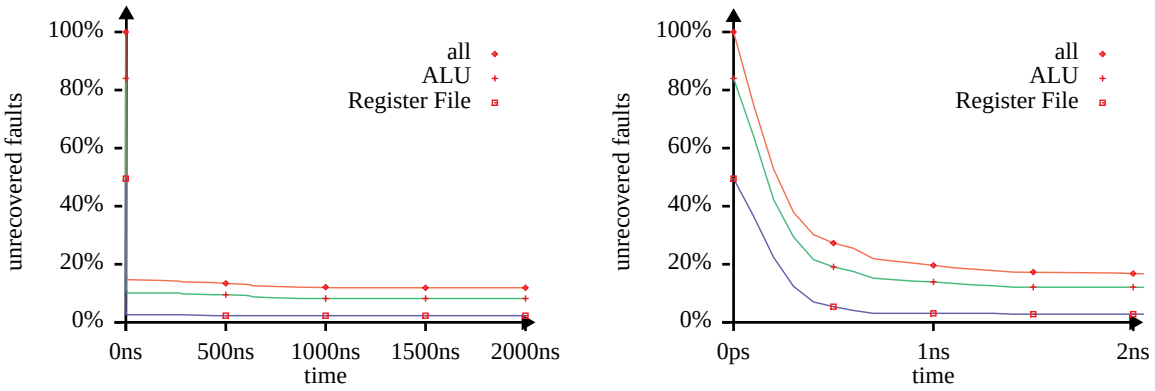
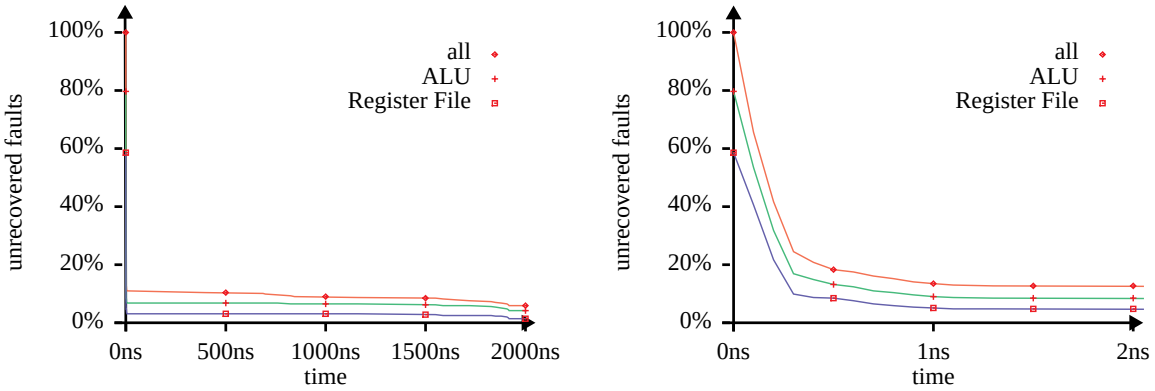


Figure 6 Experiment Overview

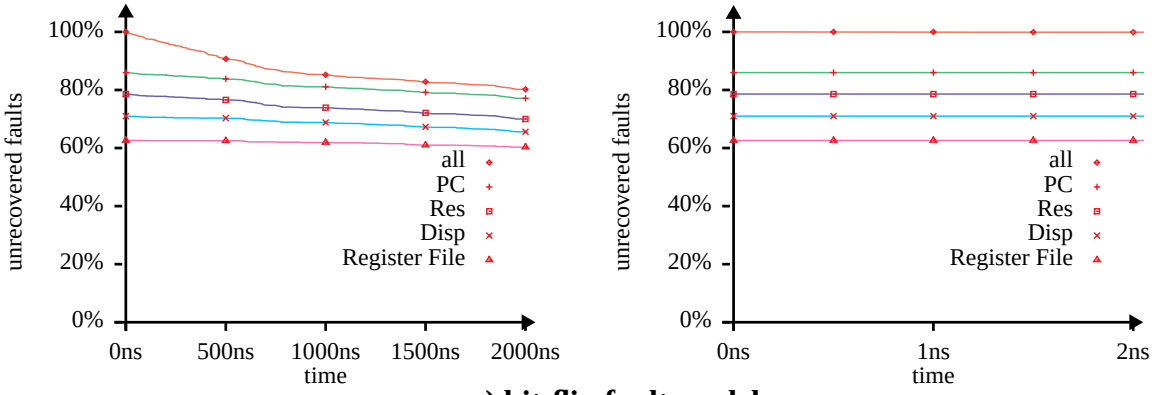
In the next diagrams (Figure 7 (a-d)) the results of the four experiments are shown in detail by showing the percentage of unrecovered faults for the several components, in which faults have been injected. It can be seen that due to the fact that most gates (and most bits) are located in



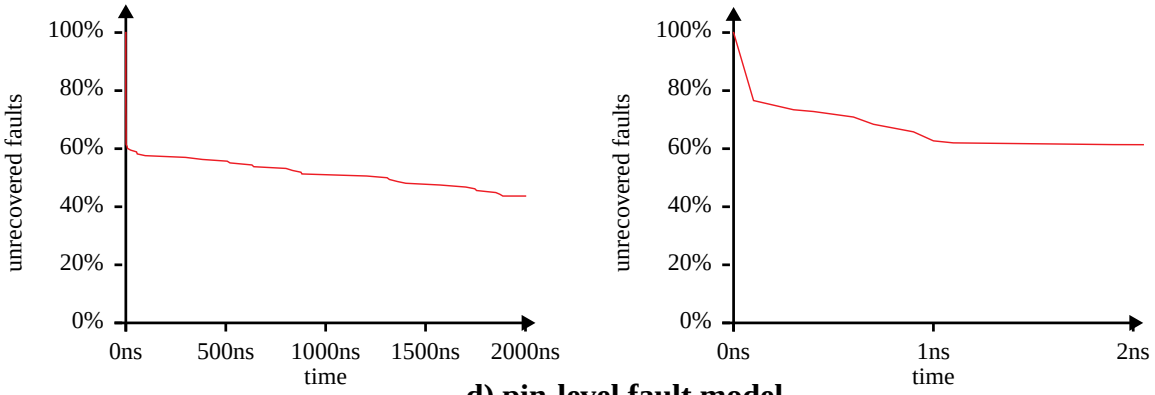
a) simple model with internal faults



b) advanced model with internal faults



c) bit-flip fault model



d) pin-level fault model

Figure 7 Detailed Experiment Analysis

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. In addition to the mean recovery time, the diagrams show also the percentage of injected faults, for which the system was not able to recover during the observation time.

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. As one can see more than 80% of the faults occurred within the register file and the ALU but only about 70% of the failures are caused by faults within these components. On the other side only less than 5% of all faults are likely to occur inside of the controller but they cause more than 10% of all failures of the system under test. A system designer can take these results and tune or exchange the components where faults tend to lead to system crashes.

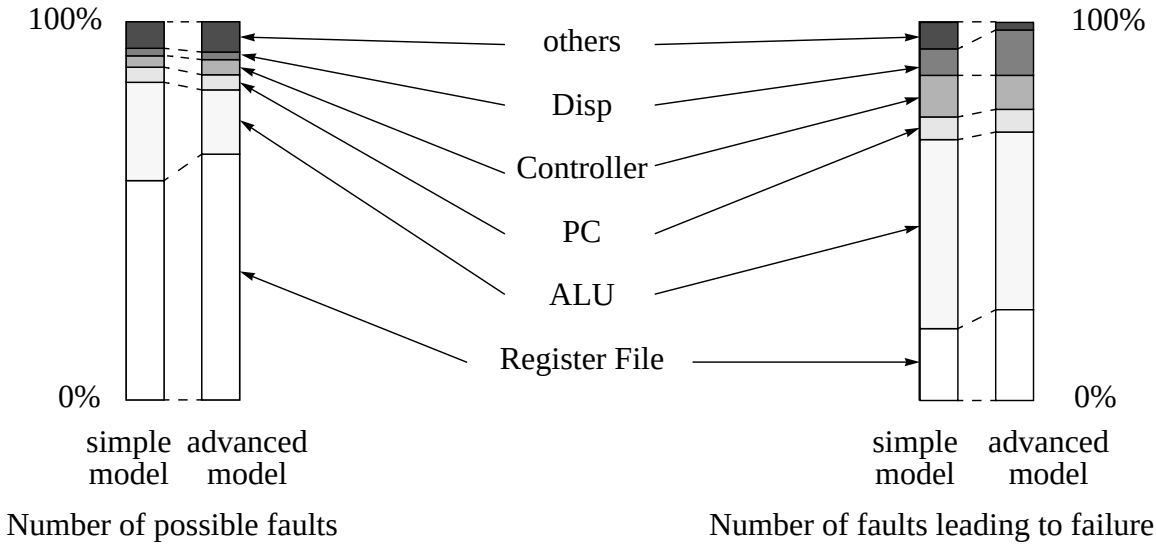


Figure 8 Comparison Fault and Failure Propabilities

Table 1 presents the summary of all four experiments. The columns “model” and “fault model” describe the experiment performed. Column “fault rate” contains the fault rate of the complete system assuming 1 fault per 100 years in our experiments. The probability of a fault to lead to a failure of the whole system (shown in the fourth column) heavily depends on the type of fault injection. It varies between 79.3% (bit-flips) and 2.0%. (internal stuck-at faults). The fifth column shows the effective failure rate of the system.

Table 1: Experiment Summary

model	fault model	mean system fault rate [$year^{-1}$]	probability of fault to lead to failure	mean system failure rate [$year^{-1}$]
“simple”	stuck-at	230	4.5%	10.3
“advanced”	stuck-at	190	2.0%	3.8
“advanced”	bit-flip	4.2	79.0%	3.3
“advanced”	pin-level	1.4	6.8%	0.097

5. Conclusion

The VERIFY tool presented in this paper allows the evaluation of the dependability features even in the design phase of a digital system. The designer of the system has only to develop the behavior description of the system. After an automatic generation of the gate-level model using a synthesis tool, the dependability of the system can be automatically evaluated using VERIFY. No additional design phase for this evaluation has to be performed.

The results of fault injection experiments heavily depend on gate-level model of the system and fault model used. Therefore, when using fault injection for the validation of the reliability of a system the model which corresponds to the actual implementation of the hardware must be used. Otherwise the results which are obtained may differ significantly from the actual hardware.

Our future research will focus on the extension of the fault injection method for mixed analogue/digital hardware description languages. This will help to extend the approach presented in this paper to embedded systems, where typically both digital and analogue components have to be developed.

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] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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

[13] 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

[14] 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.

[15] J. Karlsson, U. Gunneflo, P. Lidén, J. Torin: "Two Fault Injection Techniques for Test of Fault Handling Mechanisms", *Proc. International Test Conference,* Nashville (TN), USA, IEEE CS Press, Los Alamitos (Calif.), USA, Nov. 1991, pp. 140-149.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] G. Miremadi, J. Torin: "Effects of Physical Injection of Transient Faults on Control Flow and Evaluation of Some Software-Implemented Error Detection Techniques" in F. Cristian, G. Le Lann, T. Lunt (eds.), "*Dependable Computing for Critical Applications*" in series "Dependable Computing and Fault-Tolerant Systems", Vol. 9, Springer-Verlag Wien-New York, 1995, pp. 435-457.

[21] 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.

[22] 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.

[23] 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.

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

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