Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method*

Jens Güthoff and Volkmar Sieh

Department of Computer Science III
University of Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
email: {jsguetho,vrsieh}@immd3.informatik.uni-erlangen.de

Abstract

Fault/error injection has emerged as a valuable means for evaluating the dependability of a system. In particular, software-based techniques, which can be described as software-implemented and simulation-based techniques, have become very popular because of the relative simplicity of injecting faults. After discussing the advantages and drawbacks of these techniques, two approaches are introduced which try to overcome crucial problems when using software-based fault injection techniques. The first one improves the accuracy of software-implemented fault injection experiments. The second one offers detailed insights into the system dynamics in the presence of faults. With this knowledge, the number of fault injections, a major concern in simulation-based fault injection, can be significantly reduced. These approaches can be joined together offering accuracy of fault injection results as well as transparency of the system dynamics in the presence of faults. A case study, in which the de facto dependability properties of a standard component, a Motorola MC88100 RISC processor, are evaluated shows that.

1 Introduction

Digital systems have been entrusted with increasingly more critical responsibilities, requiring high dependability. Therefore, several fault tolerance mechanisms are integrated into a system. A major step towards the development of fault tolerant computing systems is the validation of the dependability properties of such a system. In the last decade, fault injection has become a popular technique for experimentally determining the dependability parameters of a system, such as detection latency or fault coverage.

In this paper, two novel approaches for performing fault injection are presented. The first one improves the accuracy of fault injection results by joining software-implemented and simulation-based fault injection techniques. The second one provides detailed insights into the internal dynamic behavior of the system under investigation (SUI). Disclosing the system dynamics is a prerequisite to investigate the effects of faults/errors in detail and can be also utilized to determine fault injection targets in time and location. With detailed knowledge of the internal dynamic behavior, the number of fault simulation runs can be significantly decreased, since injection is reduced to the important fault scenarios. Both approaches can be joined together offering accuracy of fault injection results as well as transparency of the system dynamics in the presence of faults. After discussing the two novel approaches, a case study is presented in which the dependability of a Motorola MC88100 RISC processor processing a Dhrystone-Benchmark is evaluated using these methods. Results obtained from the software-based and simulation-based fault injection experiments are also discussed here. Also examined is whether the simulation model is an accurate representation of the MC88100, i.e., whether the model is valid.

2 Related Research

Several researchers and organizations have utilized fault injection for system validation, a technique used to accelerate the occurrence of faults, errors, or failures [10]. A coarse classification of the fault injection methods is the division into (i) hardware-based (physical) and (ii) software-based fault injection methods. Furthermore, the software-based techniques can be separated into (a) software-implemented fault injection, where data is altered and/or timing of an application is influenced by software while running on real hardware, and (b) simulation-based fault injection, where the whole system behavior is modeled and imitated using simulation.

Various tools have utilized fault injection to evaluate system dependability applying the different techniques

* This work is supported by the Deutsche Forschungsgemeinschaft as part of SFB 182 and project no. II D 6-Da 141/7-1.

This work will be published in the Proceedings of the 25th Annual International Symposium on Fault Tolerant Computing, Pasadena (CA), USA, June 1995.

mentioned. A survey about current state of the art tools is given in [10]. In further, we are going to concentrate on software-based fault injection methods which have become very popular because of the relative simplicity of injecting faults.

FIAT [22] is a tool that injects faults in a distributed system task by corrupting the task memory image using special software. Unfortunately, the method used cannot inject transient errors into the instruction execution process. Furthermore, the time resolution of this software-implemented FIAT is in the order of seconds, which may skew the results. FERRARI [12], another software-implemented tool, uses dynamic corruption of the process control structure that in turn alters the execution state of the target program. FERRARI is capable to inject various transient and permanent faults/errors, such as address line faults, data line faults, and faults in the condition code flags. However, the fault locations in both hardware and software are limited to locations accessible to machine instructions. In section 3, we present a method which overcome this problem by combining software-implemented and simulation-based fault injection techniques discussed next.

In [11] the MEFISTO tool is presented which supports a system analyst in experimentally validating the system dependability. Here the simulation-based fault injection technique is chosen, offering the advantage of perfect controllability over where and when a fault/error is injected. Since users can model specific behaviors, simulation-based tools are not limited to any predefined set of fault models or component types. An erroneous system behavior can be forced by using mutants modeling a behavior different to the fault-free model-component, or saboteurs altering signals and variables. Faults can be also injected by using the command language of the underlying simulation engine. An important question when using simulation-based techniques is whether the simulation model in an accurate representation of the system, i.e., whether the model is valid. If a model is not valid, then any conclusions derived from the model will be of doubtful value. DEPEND [7,8] has shown in several case studies that the simulation-based fault injection approach is applicable. For example, the accuracy of the simulation model was validated by comparing the results of the simulations with measurements obtained from the fault injection experiments conducted on a Tandem Integrity S2 machine [10]. However, there are two major problems when using simulation-based fault injection: the effort and time required (i) to develop a simulation model and (ii) to perform simulation.

The first issue can be managed using modular decomposition and modular composability, which is supported by most of the current simulation tools. The second issue is a more serious problem in simulation-based fault injection. In most of the fault injection tools, time and location for fault activation is determined at random. Due to its consequently stochastic nature, **many** fault injections have to be performed in order to get credible results about system dependability and to evaluate the efficiency of built-in error detection and handling mechanisms. In order to save time in fault injection based dependability evaluation, there exist three solutions in principle: (i) workload-based fault injection [7, 12] (ii) sample-spread-based fault injection [19, 23], or (iii) introducing error behavioral functions [20]. However, in the referenced approaches the fault injection experiment itself is still a statistical one. In section 4, we present a method which is capable to deterministically determine fault injection targets based on the system‘s operational profile.

The contribution of this work is a method which:

1. Combines software-implemented and simulation-based fault injection improving the accuracy of results in contrast to current pure software-implemented methods.
2. Provides detailed insights into the system dynamics in the presence of faults/errors.
3. Performs operational profile-based fault injection in order to reduce the number of fault injections significantly.

3 Mixed-Mode Fault Injection

Software-implemented fault injection provides a cheap means to modify the hardware/software state of the SUI under software control, thus forcing an erroneous system behavior. One major drawback of software-implemented fault injection is the inability to inject faults to locations not accessible to software. Approximately 1/3 of the errors produced in logic-level fault injection cannot be emulated through the software approach [10]. Second, it must be made sure that the additional software necessary for performing fault injection experiments, such as fault injection, experiment control, or data collector and analyzer modules, does not skew fault injection results. Recent publications [2, 5, 7, 8, 11, 16, 20, 21] already pointed out the significant impact of software on system dependability.

In this situation, simulation-based fault injection can be applied offering perfect controllability over the SUI. *Mixed-Mode Fault Injection* is an approach in which software-implemented and simulation-based fault injection techniques are combined. Those components of a SUI into which a fault should be injected are simulated using a model of that component. This enables not only to alter data and/or timing of an application, but makes it possible to

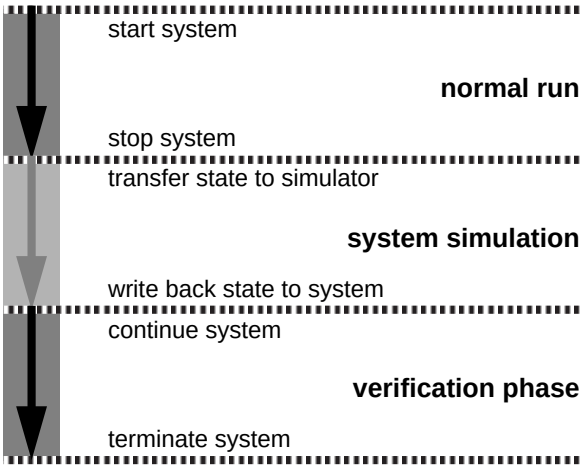

Figure 1: Functioning of the Fault Injector.

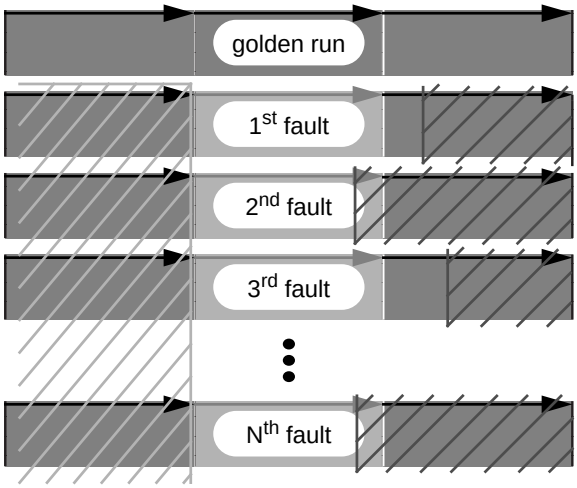

Figure 2: Multi-Threaded Fault Injection Scheme.

inject any kind of fault into a SUI. In section 5, this method is used to examine the effects of single microinstruction omissions.

The general functioning of the fault injector using the software-based approach is presented in figure 1. After starting, the SUI is working as usual. This phase of the fault injection process, denoted as normal run, is interrupted, and the system state is transferred to the simulator. Now, data can be altered and/or timing of the system can be influenced by software, i.e. the software fault injector which simulates a fault. At the end of the system simulation phase, the modified system state is written back and subsequently normal operation proceeds. Similarly to [4, 5,12], a golden run serves as the reference necessary to determine faulty states. The comparison between fault injection run and golden run is done in the verification phase.

Since nearly all computer systems support multi-tasking and/or checkpointing, they are able to save and restore the contents of all processor, coprocessor and contoller registers and pipelines. So in most cases it is possible to get a consistent state of the whole system, which can be transferred to a simulator. It is not necessary to transfer the contents of all the registers and memory cells to the simulator but only those which are used by the simulator. Because only few registers and memory cells can be altered within the (normally) short fault injection phase, only a small part of the system state has to be copied.

This methodology provides realistic measures of fault effects and fault behavior (propagation, latency) since most of the time the real target system is used during a fault injection experiment. However, there are several requirements for applying this method. Since software has a significant impact on system dependability [8, 16], an application must **not** be modified for fault injection. Consequently, the SUI must incorporate an interface which allows access to the internal state of another process in order to enable monitoring and altering of the application state without modifying the application itself. Such an interface makes use of the interrupt and exception handling mechanisms typically present in a SUI. For this reason, an application under investigation must not directly modify pointers to interrupt/exception handlers.

An acceleration of fault injection based dependability evaluation is achieved by applying a *multi-threaded fault injection scheme*. An experiment starts with performing a golden run. But instead of invoking a fault injection experiment from the very initial state for each type of fault considered, the reference states within the golden run are frozen. For each type of fault this state is restored and fault simulation proceeds just as invoked from the initial state. Therefore, the time from starting an application until fault injection time can be left out as shown in figure 2 by the hatched area on the left hand-side. Moreover, the simulation phase and verification phase are aborted as soon as the injected fault is detected or masked. This is represented by the hatched area on the right hand-side, denoting also an area which can be certainly left out. In [2] it was experimentally found that 99% of the injected faults are likely to become active within the next 200 instructions. With this method, the time-expense for fault simulation can be significantly decreased without making concession to the precision of results, which can happen using error injection mechanisms as described in [20].

4 Operational-Profile-Based Fault Injection

As stated in section 2, it is a major concern of simulation-based fault injection to reduce the number of fault injections. For example, the fault injector, a fundamental object in DEPEND [7], provides a workload-based

injection scheme that varies the fault arrival rate based on a specific workload. *Fault Expansion* [23] aims at reducing the number of fault injections. A fault is randomly selected and injected into a simulation model. After the outcome of the injection is determined, the equivalence class (EC) to which the fault belongs is identified. All faults in the same EC have common characteristics. After updating the size of the EC, all faults in that EC are removed from the population for subsequent random drawings. However, in both cases the fault injection experiment itself is still a statistical one which in turn requires many fault injection runs dependent on the confidence level chosen. In [16] it was reported that a substantial number of faults does not affect the program results. The **manual** inspection of some of these faults has shown several situations where the program result is correct in spite of fault injection:

- Faults whose errors are neutralized by the next instructions
- Faults affecting the execution of instructions that do not contribute to the benchmark results
- Faults whose errors are tolerated by the semantic of the benchmark under execution.

Uncovering such situations should be part of the evaluation process and should be supported **automatically** by appropriate evaluation tools.

In the simulation-based approach the golden run still plays a more important role than in the mixed-mode one. It not only provides reference states for multi-threaded fault injection, but also helps to reduce the number of fault simulation runs by selecting fault injection targets from *application viewpoint*. Many of the situations listed above

Inst.No	Instruction	read	write
64	`or r10,r0,0x28`	r0	r10
65	`st r10,r3,0xc`	r3, r10	—
:	:	:	:
143	`addu r10,r31,0x4c`	r31	r10

r10 is initialized

Information maintained by r10 is used for processing
(here the target address for a store instruction)

r10 is initialized with a new value
(old information is obsolete from application viewpoint)

Figure 3: Extraction of the Instruction Trace.

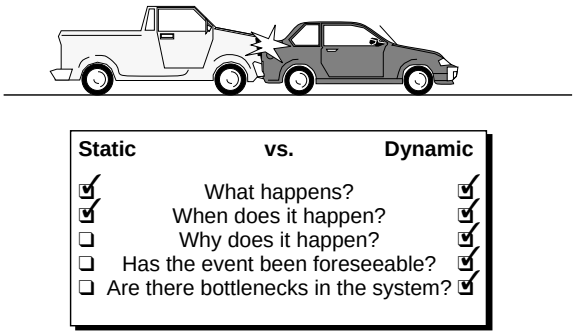

Figure 4: Static vs. Dynamic Representation of Simulation Results.

can already be identified through a detailed analysis of the golden run. The analysis should yield scenarios where the probability that an injected fault really affects program execution is extremely high. In other words, scenarios where it is certain that a fault injection does not affect program execution are neglected. Consider the instruction trace of the Dhrystone benchmark, depicted in figure 3, used in our case study. The information stored in r10 is “born“ at instruction number 64. A prior injected fault would be masked by this instruction. Until reaching the store instruction (here fortunately the next instruction), this information is sensitive on any alteration of register 10. After the re-initialization of register 10 (Inst.-No. 143), we can be sure that a fault injected into register 10 between instruction number 66 and 142 has no effect since the information maintained by this register is obsolete from application viewpoint.

The utilization of components which maintain information necessary for application processing is a measure of the probability that an injected fault manifests itself as an error, denoted as error probability. For this reason, the first step in the evaluation process is to determine the utilization of such components before performing any fault injections. The extent of fault injection is characterized by the error probabilities[1] of these components leading to *statistical fault injection*. In section 5 it will be seen that this method can drastically reduce the number of fault injections.

In the simulation-based approach, time and location for fault/error injection are determined by a detailed analysis of the golden run. Discrete-event simulation and event-driven monitoring are joined together, leading to a powerful methodology to analyze system dependability. The known advantages of simulated fault injection [11] are enhanced with event-driven monitoring, a technique employed successfully in performance evaluation of real

1. It is emphasized that the error probability of a permanent fault equals 1.

systems, which reveals the system behavior represented by events [17]. Since the system behavior is already abstracted to events during the modeling phase, monitoring of modeled system components is rather a first step in interpreting simulation data. During evaluation of a simulation run, the monitored events serve as a basis for identification of interesting fault injection scenarios.

In order to facilitate the identification of interesting fault injection scenarios, the internal dynamic behavior of a SUI must be made visible to the outside. Static representation like textual reports or static graphical analysis are not sufficient to make the system behavior visible. Additional dynamic representation can provide valuable insights into system dynamics over time (see figure 4). Interactions between system components are easily seen, the extent of an error is observable, and bottlenecks can be analyzed as they develop.

The representation of a system under investigation is based on two simple concepts [9]. (i) All process structures can be represented by hierarchical, annotated, directed graphs. These capture both control flow and data flow information. (ii) All executions of process structures can be described as traversals of these graphs by appropriately defined transactions. Interaction entities are used to enable exchange of information between components. Possible interaction entities are tokens, messages etc.

- Errors are always becoming active through interaction.
- Failures are distinguished in timing and value failures (failure domain viewpoint [13, 14]).
- The duration of such a failure can be temporary or permanent.
- Only errors of type value failure can be propagated through a computing systems, while errors of type timing failure reside in the component where the timing failure becomes active. An exception are those timing failures which cause a value failure.

Table 1: Assumptions about Error Activation, Error Propagation, and Failure Modes.

Based on the assumption listed in table 1 and using this modeling concept, simulation can be utilized for introducing "useful surrealism". In contrast to real systems, the simulation can maintain information about **when**, **where**, **why**, and **what** event occurs within the modeled system, which is not available in real systems and which can be utilized for system evaluation. In our approach, the current states of both interaction entities and model components are enriched with information about their error states and can be one of `ok`, `faulty known`, `faulty unknown`, or `dead`. These states reflect the actual state of the system and must be distinguished from system's viewpoint about its current state. Assume that an interaction entity is corrupted due to fault injection but the corruption is not already detected by the system. The actual state of the system is `faulty unknown` while from system's viewpoint the current state is `ok`. Introducing this kind of "useful surrealism" — the actual error state is, in fact, unknown from the system's viewpoint — enables efficient fault injection based dependability evaluation, because fault injection and fault behavior in time and space are made visible to the outside.

During simulation many event-traces are recorded in order to get detailed information about the internal dynamic behavior of the system. Unlike software-implemented fault injection techniques in which it must be sure that the additional software does not skew results [10], this does not invoke any problems using simulation. The modeler decides which functions contribute to system's behavior and which are used for other purposes, e.g. monitoring. The mass of event-traces stored in a trace file represents the data basis for a posteriori sophisticated data processing. Statistical evaluation and information retrieval tools produce strategic information out of uninterpreted raw data. With this strategic information, potential bottlenecks can be found and interesting fault injection scenarios are identified. Visualization offers the ability to see the data in graphical format and allows system analysts to process large amounts of information quickly. For example, animation of the event-traces enables a system analyst to observe immediately whether components are affected by an injected fault/error, which components are infected and, by stepping through the trace, where the error is propagated and how fast it is propagated. This is achieved by using different colors (see table 2) related to the error states.

Color	Error State	Reason
green	`ok`	—
red	`faulty unknown`	Corruption of the internal state (component & interaction entity)
blue	`faulty known`	Time or value failure detected due to successfully performed recovery
black	`dead`	Breakdown of a component, lost interaction entity

Table 2: Color Map.

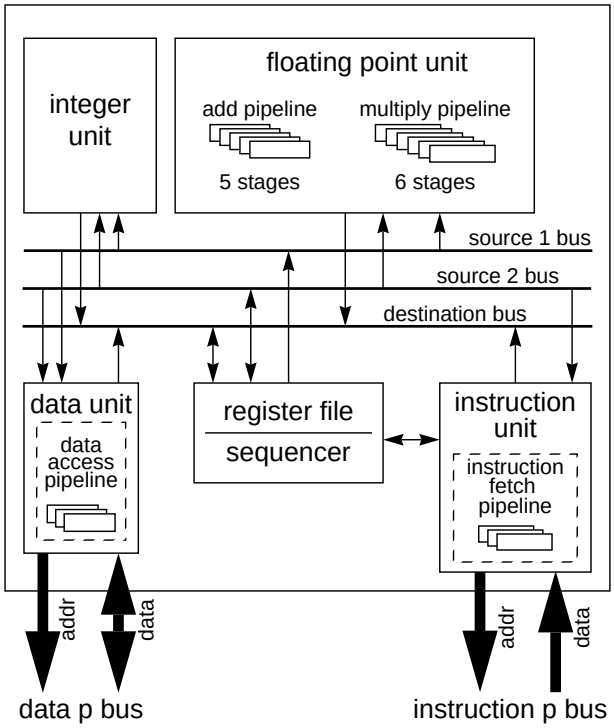

Figure 5: MC88100 Block Diagram.

5 A Case Study: The MC88100 RISC Processor

In this section, both fault injection methods are applied in dependability evaluation of a Motorola MC88100 RISC processor during the execution of a Dhrystone benchmark. The case study (i) shows how the number of fault injections can be reduced, (ii) investigates the de facto dependability properties of this processor and (iii) examines fault effect and fault behavior dependent on the type of fault. Another objective of the case study is to show that the simulation-based fault injection technique originally designed for dependability evaluation at system level is applicable even at lower levels of abstraction. The results obtained from the simulation-based experiment are comparable to the machine-close experiment. The next paragraphs discusses how the simulation model is built and how the fault injection is performed in both experiments.

5.1 Simulation Model

The Motorola MC88100 is a fully pipelined RISC processor with a Havard architecture (see figure 5). It contains four execution units which operates independently and concurrently. The integer unit and the floating-point unit execute all data manipulation instructions. Data memory accesses are performed by the data unit, and instruction prefetches are performed by the instruction unit [18]. There are 32 general-purpose registers, each 32 bits wide. These registers contain instruction operands and results, and provide address and bit-field information.

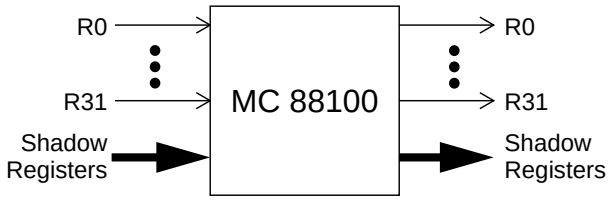

Figure 6: Simulation Model for software-based Fault Injection.

The mixed-mode approach simply considers the MC88100 as a black box with shadow registers and the 32 general-purpose registers as its in- and outputs (see figure 6). The shadow registers maintain information for exception recovery. Since the fault injector is invoked by forcing an exception handling, these registers represent the current state of the application under test. This information is used to simulate the instruction decode phase. In case of a fault injection, data and address information of an instruction can be altered or a whole microinstruction omitted.

For the simulation-based approach this simulation model is not sufficient. In a pure simulation-based approach we have to model all relevant parts of a SUI. Fortunately, the floating point unit, which is responsible for integer multiplication and division, need only perform these instructions. The implementation of all floating point instructions is completely left out since a Dhrystone benchmark doesn‘t use floating point operations. However, the remaining parts of the MC88100 represented in figure 5 must be considered. In order to process a real Dhrystone benchmark, the model must incorporate initialized memory. In addition, memory has to be provided for stack operations.The resulting simulation model is picked up in figure 7. The arbiter is responsible for the selection of the right memory module. Which module has to serve a memory access is determined by the data address. The double lines of representation of the MC88100 identify a high level component that is further unfoldable. The inner structure corresponds to the block diagram depicted in figure 5.

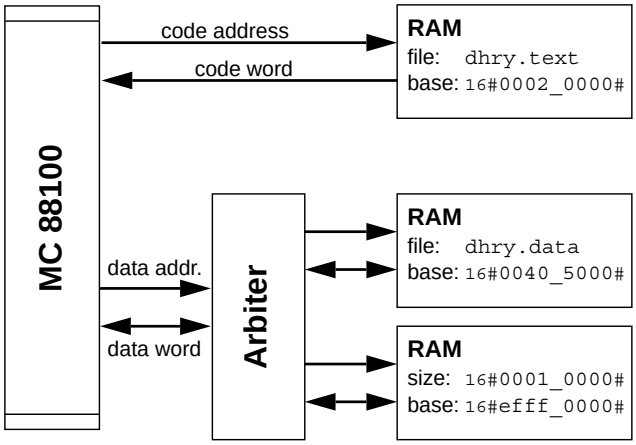

Figure 7: Simulation Model for Simulation-Based Fault Injection.

5.2 Fault Model

The fault model used in the case study is based on the functional CPU fault model elaborated by Brahme and Abraham [3]. It was validated for numerous processor types by estimating the fault coverage of the test generation algorithms showing an excellent coverage of typically higher than 90% [2].

In our experiments, faults in the *data storage* and *data transfe*r faults are mapped on bit errors in the register file, i.e. the contents of one of the 32 general-purpose register is altered. This is allowed due to the load/store architecture of the MC88100. The implementation of a *register decoding* fault can be generally taken out of the description in table 3. An instruction is decomposed into a sequence of consecutively executed *microinstructions*, each of them is regarded as a set of concurrently executed *microorders*. In this fault model, either microinstructions or microorders can be selected incorrectly instead of whole instructions. Due to the wide spectrum of *data manipulation* faults we do not deal with this class. Furthermore, we concentrate on single transient faults and the faulty area is limited to the processor. In other words, components external to the MC88100 are considered fault free. We assume that the occurrence of a fault is:

1. uniformly distributed in time from application start until termination.

2. uniformly distributed in location:
   (a) over the register file
   (b) over microorder and microinstructions

Function	Fault
data storage	bit errors
data transfer	bit errors in destination registers
data manipulation	function dependent
register decoding	inactive: READ constant WRITE no effect wrong: READ wrong source WRITE wrong destination additional: READ or/and WRITE both written
microinstruction	inactive wrong additional
microorder	inactive wrong additional

Table 3: Brahme-Abraham Fault Model.

5.3 Experiment Realization

The target system in the mixed-mode fault injection experiment is the Motorola MC88100 microprocessor, used in MEMSY [6], a multiprocessing system developed at the University of Erlangen, running on the UNIX operating system. The application processed by the system is a Dhrystone benchmark translated into machine-code by using the Greenhill C-compiler. It is important to name the specific compiler because experiments, not discussed here, have shown that the results of fault injection experiments on one target system using the same application described in a high level language vary on the compiler used. As in FERRARI [12], the `ptrace(2)` interface of UNIX operating system originally designed to enable program debugging is used for performing fault injection. The fault injector simulates the execution of the next instruction and injects custom faults as described in the fault injector configuration file compliant to the fault model.

In the simulation-based fault injection experiment we first have to built a model of the SUI. We use VHDL for the description of the simulation model for reasons pointed in [11, 15]. Despite the modeling capabilities of VHDL the simulation-based experiment differs from the mixed-mode one in the point that the UNIX operating system is not modeled due to model complexity. In order to process the application anyway, the application is reduced to the raw Dhrystone benchmark. In other words, the start-up code,

exception handling, and termination sequence are left out completely. Apart from that the experiment is performed using the same assumptions as the mixed-mode approach, but using the application viewpoint for selecting fault injection targets (recall section 4). Since in [11] a very good summery of the fault injection capabilities in simulation models is given, this topic is not discussed here.

5.4 Results

In the mixed-mode fault injection experiment, 5000 faults/errors were injected into the system. Like many others fault injection tools, time and location for injecting an error are determined at random related to the fault model introduced in subsection 5.2. Using the multi-threaded fault injection method, the whole fault injection process needs 42 minutes for completion.

As stated in section 4, the simulation-based approach follows another strategy in determining time and location for injection errors. The evaluation process starts with measuring the register utilization. Table 4 shows the register utilization measured within the simulation-based fault injection experiment. We can conclude that more than 40% of the faults injected in the register file during the execution of the raw Dhrystone benchmark cannot manifest themselves as errors certainly. With regard to the fault model represented in this section, the probability that a fault manifests itself as an error is less than 18% in all.

Register	Mean Utilization
r0, r31	0.979
r1, r14	0.538
r3, r4	0.403
r5, r12,r13	0.309
r2, r8-r11, r23-r25	0.117
r6, r7	0.016
r15-r22, r26-r30	0.0

Table 4: Register Utilization Classes.

This means that for a fault injection experiment where time and location for a fault injection experiment are uniformly distributed over the application run-time and the register-file respectively more than 82% of the injected faults have no effect on the error outcome. It should be emphasized that this result doesn‘t evaluate the dependability properties of a system but shows the significant impact of software on system dependability.

If we consider the estimated error probabilities in figure 8, it is promising to first inject faults in register r1,

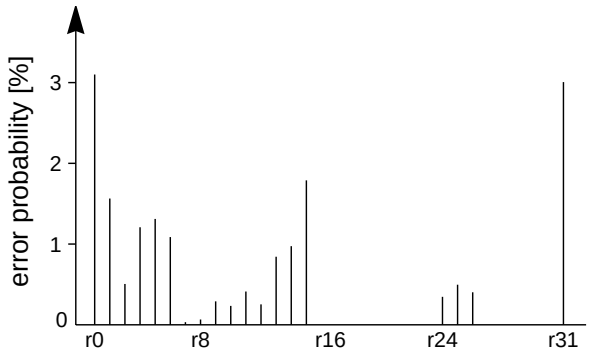

Figure 8: Estimated Error Probabilities.

r14, and r31, because faults injected in these registers are more likely to manifest themselves as errors. Register r0 is taken from the relevant set because this register denoted as constant zero in [18] can only be read. A transient fault concerning this register can only manifest itself as a data transfer error. Because of this technological speciality only the data transfer phase may be computed to the register utilization of r0. In this case, less than 8% of the overall register accesses refer to register r0 and so it must be removed from the most relevant set. The next relevant set comprise register r3, r4 and so on. While it may be straightforward to inject faults into the most utilized registers, it is important that a statistically significant percentage of fault injections which unlikely manifest themselves as errors are included in the fault injection experiment. Using this strategy in performing fault injection experiments combined with the visualization capability offered within the simulation-based approach, we are able to forecast the most likely faulty behavior of a SUI.

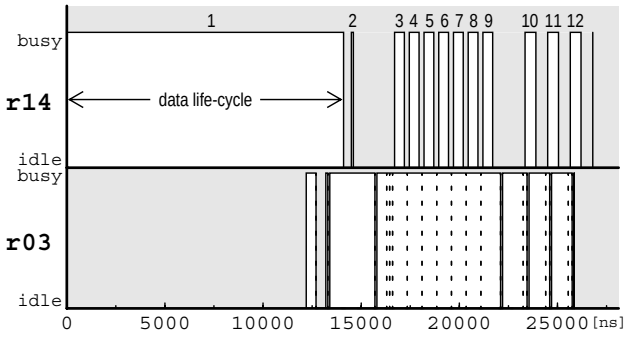

Figure 9: Fault Injection Timing.

The number of fault injections into a specific system component depends on the relevance of that component for application processing. In case of the single-bit error assumption affecting the processor‘s register file, the extent of fault injections into a specific register is proportional to its utilization — utilization as described in section 4. A

further reduction is achieved through a sophisticated *fault injection timing*. In figure 9 the register utilization over time of register r14, which is part of the most relevant set, and register r3, which is part of the 2nd order relevant set, are depicted exemplary. Here, a representation from application viewpoint is selected whereby the utilization is divided into several data life-cycles. A data life-cycle starts with the initialization of a register through a write access followed by any number of read accesses (represented by dotted lines in figure 9). The last read access before a write access determines the end of a data life-cycle. Under the single-bit error assumption, faults need to be injected only within data life-cycles or rather before each read access. For this reason, there exists only 12 moments for injecting a single-bit error into register r14 and 19 moments for injecting a single-bit error into register r3. For the whole register file a total of 220 out of 21600 possible moments for injecting a single-bit error were identified which really affect program execution. Conventional fault injection schemes would need more than 12000 fault injection experiments in order to hit all these fault injection moments with a likelihood of 90 percent. Using the single-bit fault model, a total of 6176 different faults/errors can be injected which certainly affect program execution. Each result obtained by any of the 6176 possible fault injection experiments must be weighted with the corresponding relative data life-cycle length. 1000 fault injection experiments took about 60 minutes on a SPARC10 (SunOS 4.1.3) using Model Technology‘s V-System VHDL-Simulator.

Number of observations	32
Mean of independent variable	18.41
Mean of dependent variable	17.92
Standard dev. of ind. variable	26.90
Standard dev. of dep. variable	26.66
Correlation coefficient	0.9996
Regression coefficient(slope)	0.9909
Standard error of coefficient	0.0048
Regression constant(intercept)	-0.3219
Standard error of constant	0.1548

Table 5: Regression Analysis.

However, the results obtained from the simulation-based approach are worthless if the simulation model doesn‘t approximately imitate the real behavior of the SUI. The validation of the simulation model used in the simulation-based approach shows an excellent agreement with the mixed-mode one concerning the register utilization and the Microinstruction omission while processing the raw Dhrystone benchmark. In table 5 the regression analysis of the register utilization is presented whereby the register utilization of the mixed-mode approach is considered an independent variable. Because of the excellent agreement we were able to examine the dependability properties of the MC88100 processing the raw Dhrystone benchmark by using the visualization capabilities offered within the simulation-based approach. It was observed that some of the single bit-errors in data/address registers are masked due to the delayed branching capability of the MC88100. If the processor reaches an instruction directing the next sequential instruction to be suppressed, the next sequential instruction already present in the instruction pipeline is marked invalid. Furthermore, it was observed that faults in register r1 are likely to manifest themselves as control flow errors. This happens because the return address of a subroutine call is stored here.

Register	dhry 0	dhry 100	dhry 500
r1	0.278	0.396	0.410
r2-r30	0.237	0.468	0.496

Table 6: Register Utilization.

For a single omission of a microinstruction it was observed that these errors produce erroneous results rather than control flow errors. Since errors in application data are not detectable by the built-in error detection mechanisms of the MC88100 the coverage is much worse than for single bit faults in data or address registers. Actually omissions of microinstructions manifest themselves more likely as errors which cause a system failure. Beside this, the error latency is much shorter than for single bit faults. It is very likely that the system already fails a few instructions after fault injection.

Fault type	without effect	detected	erroneous result
1-Bit-Fault in Data/Address Register	0.78	0.12	0.10
Omission of one Microinstruction	0.56	0.10	0.34

Table 7: Fault Effects.

Using the software fault injector we can investigate the dependability properties of the MC88100 running under real conditions. In table 6 the register utilization for different numbers of invocations of the Dhrystone

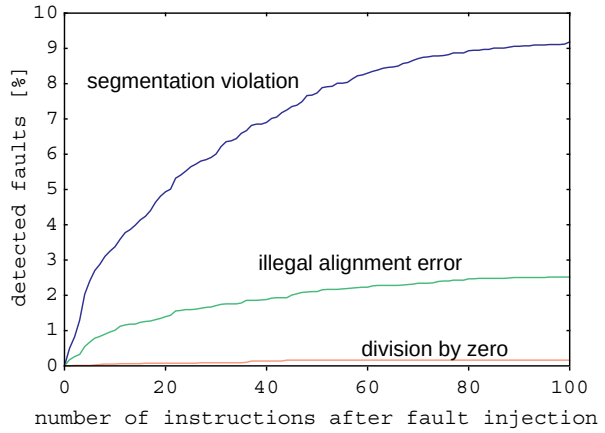

Figure 10: Detection Latency for Single Bit Faults in Data/Address Register.

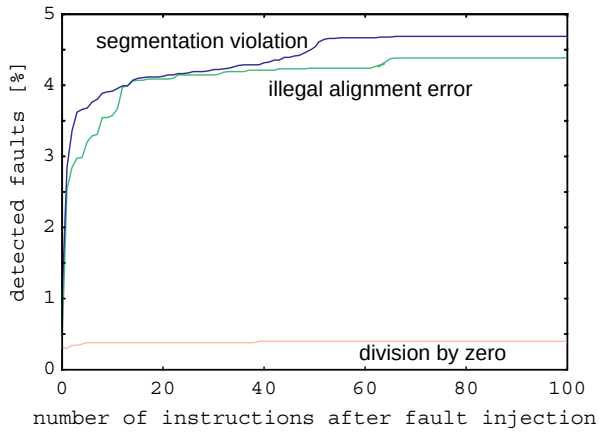

Figure 11: Detection Latency for Single Microinstruction Omission.

benchmark are listed (0, 100, and 500 times). As we can see in table 6, the start-up code usual to a binary produced by a compiler has a significant impact on the register utilization. We can conclude that some of the registers not used during the execution of the raw Dhrystone benchmark maintain information used for appropriate application termination. The fault effects summarized in table 7 underpin the guess that a dependency exists between register utilization and fault manifestation The results presented refer to the fault injection experiment dhry 500 where the Dhrystone benchmark is called 500 times. For this experiment, we estimate that less than 46% of the injected faults manifest themselves as errors. In both cases, single bit faults in data/ address register and single microinstruction omission, this boundary is in fact not violated.

Figure 10 and 11 show the different detection latency distributions dependent on the fault type in time expressed in number of instructions after fault injection. The resulting system failures can be separated into:

- *Segmentation violation*: The processor tried to access memory cells outside his address space;
- *Bus error*: The processor tried to access a word in memory with illegal word alignment;
- *Division by zero*: The processor tried to divide a number by a zero value;

The latency of single-bit storage faults (figure 10) is a distribution of the form $K_1 e^{-K_2 x}$. K_2 is approximately 0.036 for all different failure types. K_1 depends on the type of the failure. Because it is nearly an exponential function, one can say that the probability for a process to die due to an erroneous bit in one of the processor registers is nearly the same at every instruction.

This is not the case in the second experiment as shown in figure 11. The probability of a process to die is much higher in the next few steps than later.

6 Conclusion

In this paper, we presented two novel approaches which, put together, provide a powerful fault injection method. The *Mixed-Mode Fault Injection* approach combines software-implemented and simulation-based fault injection by utilizing the advantages of these techniques. The key point is to improve the accuracy of results because fault/error effects, such as the omission of a single microinstruction, are hard to emulate using pure software-implemented techniques.

The *Operational-Profile-Based Fault Injection* approach makes the system dynamics transparent to the system analyst. Full control and observation over the system‘s components is a must in order to understand the system‘s behavior completely. Simulation combined with monitoring offers full control and observation of the modeled system components. With detailed knowledge of the internal dynamic behavior, the number of fault simulation runs can be significantly decreased, since injection is reduced to the important fault scenarios. The relevant set can be determined by a detailed analysis of the golden run.

Both approaches use simulated fault injection techniques which makes it possible to join them together into a powerful fault injection method. It is fast and accurate because fault simulation is restricted only to the fault injection phase while using real hardware otherwise. Within the simulation phase, the number of fault injections can be limited to those which are likely to force an erroneous system behavior. The method provides detailed

insights into the fault/error effects which is a prerequisite in system tuning, in particular in tuning system dependability.

7 References

[1] P.J. Ashenden: *The VHDL Cookbook*. University of Adelaide, South Australia, Technical Report, 1990

[2] B. Benyó, J. Hönig, W. Hohl, A. Pataricza, B. Sally, and V. Sieh: Fault Injection Based Validation of Fault-Tolerant Multiprocessors. *8th Symposium on Microcomputer and Microprocessor Applications*, pp. 85-94, Oct. 1994

[3] D. Brahme, J. Abraham: Functional Testing of Microprocessors. *IEEE Trans. Computers*, vol. C-33, no. 6, Jun. 1984

[4] G. Choi, R. Iyer, and V. Carreno: FOCUS: An experimental environment for validation of fault sensitivity analysis. *IEEE Trans. Computers*, vol. 41, no. 12, pp. 1515-1526, Dec. 1992

[5] E.W. Czeck and D.P. Siewiorek: Observations on the Effects of Fault Manifestation as a Function of Workload. *IEEE Trans. Computers*, vol. 41, no. 5, pp. 559-566, May 1992

[6] M. Dal Cin, W. Hohl, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hönig, F. Hofmann, C.U. Linster, E. Michel, A. Pataricza, T. Thiel, and S. Turowski: *Architecture and Realization of the Modular Expandable Multiprocessor System MEMSY*. MPCS‘94, Conference on Massively Parallel Computing Systems, Ischia, 1994

[7] K.K. Goswami and R.K. Iyer: *DEPEND: A Simulation-Based Environment for System Level Dependability Analysis*. Technical Report, CHRC 92-11, Univ. of Illinois at Urbana-Champaign, Jun. 1992

[8] K.K. Goswami and R.K. Iyer: Simulation of Software Behavior Under Hardware Faults. *Int. Symp. Fault-Tolerant Computing, FTCS-23*, IEEE Computer Society, pp. 218-227, Jun. 1993

[9] D.J. Hatley and I.A. Pirbhai: *Strategies for real-time system specification*. New York Dorset House Publ., 1987

[10] R.K. Iyer and D. Tang: *Experimental Analysis of Computer System Dependability*. Technical Report, CHRC, Univ. of Illinois at Urbana-Champaign, May 1994

[11] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson: Fault Injection into VHDL Models: The MEFISTO Tool. *Int. Symp. Fault-Tolerant Computing, FTCS-24*, IEEE Computer Society, pp. 66-75, Jun. 1994

[12] G.A Kanawati, N.A. Kanawati, and J.A. Abraham: FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Trans. Computers*, vol. 44, no. 2, pp. 248-260, Feb. 1995

[13] H. Kopetz: The Failure Fault (FF) Model. *Int. Symp. Fault-Tolerant Computing, FTCS-12*, IEEE Computer Society, pp. 14-17, Jun. 1982

[14] J. C. Laprie: *Dependability: basic concepts and terminology*. Springer-Verlag, 1992

[15] R. Lipsett: *VHDL*. Kluwer Academic Publisher, 1989

[16] H. Madeira and J.G. Silva: Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking. *Int. Symp. Fault-Tolerant Computing, FTCS-24*, IEEE Computer Society, pp. 350-359, Jun. 1994

[17] B. Mohr: Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis System Possible? In J.J. Dongara, B. Tourancheau (ed.): *Advances in Parallel Computing*, vol. 6, pp. 103-124, Elsevier, 1993

[18] Motorola, Inc: *MC88100 RISC Microprocessor User‘s Manual*. Prentice Hall, Englewood Cliffs, 1990

[19] D. Powell, E. Martins, J. Arlat, and Y. Crouzet: Estimators for Fault Coverage Evaluation. *IEEE Trans. Computers*, vol. 44, no. 2, pp. 261-273, Feb. 1995

[20] M. Rimén, J. Ohlsson, and J. Torin: On Microprocessor Error Behavior Modeling. *Int. Symp. Fault-Tolerant Computing, FTCS-24*, IEEE Computer Society, pp. 76-85, Jun. 1994

[21] H.A. Rosenberg and K.G. Shin: Software Fault Injection and its Application in Distributed Systems. *Int. Symp. Fault-Tolerant Computing, FTCS-23*, IEEE Computer Society, pp. 208-217, Jun. 1993

[22] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kowknacki, J. Barton, R. Dancey, A. Robinson, and T. Lin: FIAT — Fault Injection Based Automated Testing Environment. *Int. Symp. Fault-Tolerant Computing, FTCS-18*, IEEE Computer Society, pp. 102-107, Jun. 1988

[23] W. Wang and K.S. Trivedi: The Impact of Fault Expansion on the Interval Estimate for Fault Detection Coverage. *Int. Symp. Fault-Tolerant Computing, FTCS-24*, IEEE Computer Society, pp. 330-337, Jun. 1994