Control and Cybernetics

vol. **33** (2004) No. 4

Fault injection stress strategies in dependability analysis

by

J. Sosnowski, P. Gawkowski and A. Lesiak

Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19, 00-665 Warsaw, Poland
e-mail: jss@ii.pw.edu.pl

Abstract: The paper deals with the problem of testing computer system's susceptibility to hardware faults by means of software implemented fault injectors. Basing on our experience with fault injection techniques we present various strategies of fault stressing in relevance to fault impact analysis in the function of the application input data profile, fault injection profile in time and space, resource activities etc. We discuss the problem of test result qualification and significance. Fault hardening at the software level is also outlined. The considerations presented are illustrated with numerous experimental results obtained in Windows and Linux environments.

Keywords: fault injection, fault modeling, test coverage, fault detection and fault tolerance evaluation.

1. Introduction

As computer systems become more complex, designers have to deal with various error detection and fault tolerance mechanisms (in hardware and software). An important issue is to analyze fault effects in the system. Doing this at high abstraction levels with analytic methods is significantly limited. In practice, designers have to go to lower levels of system implementation and to physical fault models in order to get a better insight into fault effect propagation etc. Hence, we observe recently an increasing interest in fault injection techniques.

Faults can be injected into systems or their models. The second approach relates to different abstraction levels e.g. functional or RTL defined in VHDL. Some simulators of this kind have been proposed in Arlat et al. (2003), Carderilli et al. (2002), Carreira et al. (1998), Leveugle (2000), Rebaudengo and Reorda (1999), Sieh et al. (1997), Velazco et al. (2002). In this case we face the problems of model accuracy and simulation time. These problems are eliminated by fault

injection (physical or logical) into the target or prototype system. Physical fault injections are accomplished by corrupting a logic value at circuit pins, disturbing voltage on power lines, bombarding hardware with heavy ion radiation, laser beam etc. These techniques need special and expensive equipment and the class of injected faults and experiment controllability are limited (see, e.g., Chen et al., 1997; Constantinescu, 2003; Folkesson et al., 1998; Madeira and Silva, 1994; Samson, 1998; Vargas et al., 2003). More flexible are the software implemented fault injectors, which disturb the states of CPU registers and memory locations (e.g. Baldini et al., 2000; Carreira et al., 1998; Choi and Iyer, 1992; Kanawati et al., 1992; Madeira et al., 2002; Segall and Lin, 1988).

Fault injection experiments should mimic the occurrence of faults within the analyzed system so as to evaluate the dependability features of the system, e.g. fault tolerance effectiveness, safety and reliability. It is not possible to inject all faults and simulate all conditions that could occur during system operation. Hence, fault susceptibility evaluation needs statistical methods. Cukier et al. (1999) discuss mathematical methods of estimating confidence limits for the result samples. Some other aspects of fault stressing are described by Tsai et al. (1999). Depending upon the goal of the performed analysis (dependability evaluation, identification of weak points in the design), we have to design test strategies with an appropriate test stimuli profile and fault effect observation schemes. This problem has been neglected in the literature, as most authors restricted their considerations to a general and coarse-grained analysis. In our approach we tune the fault injection campaign to experiment objectives and application properties. For this purpose we have developed quite sophisticated and flexible fault injectors (Section 2) with high level of experiment controllability/observability and flexible result granularity and qualification. These tools allow us to identify factors influencing test strategies (Section 3), e.g. fault stress coverage (class of faults, localization in time and space, test profiles) and resource activity. In this paper we do not only generalize our observations from many experiments but also develop special test scenarios to illustrate critical interpretation issues (Section 4). We show that the presented approach improves test effectiveness.

2. Fault injection tools

We have studied the problem of software implemented fault injections (SWIFI) for several years (Derezińska and Sosnowski, 2002; Gawkowski and Sosnowski, 2001a, 2002a, b; Sosnowski et al., 2003), using special fault injectors developed at the Institute of Computer Science of the Warsaw University of Technology. Two injectors (FITS, MTFI) relate to Windows and one (LI) to Linux environment. FITS is targeted for classical applications, whereas MTFI covers multithreaded applications (neglected in the literature). These tools have systematically been enhanced with various monitoring and data logging procedures very useful in system behavior analysis.

In the SWIFI injectors faults are specified as disturbances of processor registers, disturbances of the executed code and disturbances of memory locations. The following fault types are possible: bit inversion (XOR), bit setting, bit resetting, bridging (logical AND or OR of coupled bits). The disturbed bits are specified in the fault mask register. It is also possible to select pseudorandom generation of the mask register (within specified category, e.g. single or k-bit faults). Duration of faults is specified by the number of instructions for which the fault must be active (starting from the triggering instruction). The injector provides high flexibility in specifying the moment of fault injection - fault-triggering point (in time and space), fault type and location. All these specifications can be defined explicitly by the experiment designer or generated in a pseudorandom way. Fault triggering specifies the moment of fault injection. It can be related to the execution time of the analyzed application, the executed codes or the memory references. We discuss this in more detail in Section 3.

The fault injector compares the generated states of registers with those found during the golden run (fault free). The golden run collects information on executed instructions, states of registers, generated events, exit code, calculated results etc. Such a complete dynamic image allows us to find various statistics useful in the application characterization (e.g. code, data and stack area, distribution of register states, executed instructions). During experiments we collect detailed or aggregated test results (such flexibility is also assured for the golden run). Result aggregation can be done at various levels. In most cases we use coarse-grained aggregation with five classes of test results: C - correct result, INC - incorrect result, S - fault detected by the system (specification of the number and types of registered exceptions), T - time-out, U - user defined messages (generated by the application, they may overlap with other categories).

Fine-grained result specification takes into account tolerance margins, fault severity levels, side-effects (e.g. loss of data integrity, disturbing internal states influencing rarely used functions), statistics of collected exceptions or user messages etc. Recent COTS (commercial-off-the-shelf) systems comprise various mechanisms for fault detection or tolerance. They are implemented in hardware and software. In hardware they are mostly related to parity or more complex error detection and error correction codes used in memories (cache, RAM, disc) or transmission channels. Moreover, microprocessors comprise built-in detectors which signal various exceptions: access violation (within RAM), in page error, array bounds exceeded, data type misalignment (wrong word boundaries), illegal instruction, etc. (Sosnowski et al., 2003).

The fault injectors developed assure many possibilities in designing simulation experiments. Moreover, they deliver various statistics related to the analyzed application (e.g. static and dynamic distribution of instructions or referenced addresses, test coverage, resource activity, register state changes frequency) as well as to the performed fault injections (e.g. distribution of fault injections in time and space, error detection or recovery latency). These statistics are very useful while interpreting final results and evaluating the fault

stressing level. For facilitating interpretation of simulation results we have introduced an original resource activity measure. The activity ratio (AR_r) is the percentage of the execution time during which the considered resource r (e.g. a specified register, memory location) holds useful data (from its loading to the last read operation). The non-activity ratio is $NA_r = 1 - AR_r$. The activity ratio is defined as follows:

$$AR_r = (T_{r1} + T_{r2} + ... + T_{rk})/T$$

where T is the time of the application operation and T_{ri} are so-called active periods during which resource r_i (e.g. RAM location, processor accumulator register - EAX) stores data that is used in calculations etc. Active periods T_{ri} can be found by calculating the time between loading the considered resource with new data and the last use of this data (i.e. the last read operation). To evaluate the activity ratio of a given resource r we have to analyze all executed instructions in the program, identify the type of the operation (related to resource r) and execution time stamp. An instruction can perform simple operation read from or write to some resource r. Complex operations like *read/write* or *write/read* are also possible. However, in the activity analysis only the first operation is important (*read or write*). The identified type of operation determines whether the analyzed instruction starts a new activity period (the first operation is *write*) or extends its duration (the first operation is *read*). This analysis is quite complex, because instructions can use the resource indirectly or do some operations conditionally (according to the states of different resources). Other problems relate to the structure of the analyzed resource. Some instructions use only selected parts of the resource e.g. only specific bytes or bit fields of a register. In such cases it is preferable to analyze separately every part of the resource. Identification of resources used by a given instruction sometimes is not easy due to complex addressing modes (e.g. indirect memory addressing) or relative indexing (e.g. FPU stack registers). The general idea of the developed algorithm generating lists of activity periods for all used resources is given below (in Visual Basic notation):

```
FOR EACH r IN ResourcesSet
    r.ActivePeriodsList.AddNewPeriod(0)
NEXT r
FOR EACH Ins IN InstructionStream
  FOR EACH r IN ResourcesSet
    SELECT CASE InvestigateInstructionActions(Ins, r)
      CASE read
        r.ActivePeriodsList.CurrentPeriod.ExtendTo(Ins.Timestamp)
      CASE write
        r.ActivePeriodsList.AddNewPeriod(Ins.Timestamp)
      CASE ELSE
    END SELECT
  NEXT r
NEXT Ins
```

All activity periods of a given resource r are held in a list (*r.ActivePeriodsList*) associated with this resource. The activity period is defined by the starting and terminating timestamps. At the beginning the algorithm initializes all the activity period lists with null periods (i.e. the starting and terminating timestamps are equal and relate to the first instruction in the program). This is done in the *FOR....NEXT* loop with the operation *AddNewPeriod(0)* performed for resource r in the set of all used resources (*ResourcesSet*). Next, the content of the lists is filled in successively according to the performed read and write operations. These operations are identified with the function *InvestigateInstructionActions(Ins, r)*, which returns the type of the first operation performed by instruction *Ins* on the resource r. For the *read* operation the current activity period is updated by assigning the considered instruction time stamp as terminating the activity period (function *ExtendTo(Ins, Timestamp)* performed on the resource r list). The write operation appends to the resource list a new activity period (with the same starting and terminating instruction time stamp. This is performed by function *AddNewPeriod(Ins, timestamp)*. The lists of active periods are updated in nested loops running over all resources and instructions.

3. Fault stresses and test profiles

Dependability evaluation of complex and real applications creates many problems. In particular, they relate to the test profile of the analyzed application, fault injection policy (the class and the number of faults, their distribution in time and space), qualification of experiment results, experiment execution time etc.

The location, timing, type and conditions for faults being injected exert a significant influence on test results, and so they should be carefully defined. Faults can be injected in different functional blocks, e.g. specified processor registers, floating point unit register stack, control or state registers, and specified RAM locations. In practice, not all system resources (functional blocks) are used in the analyzed applications, moreover, if used, they can perform different functions in the realized algorithm. Faults injected into the unused resources have no effect on system operation. Another issue relates to the timing of fault injection. Injecting faults into a code segment after its execution or into a register before its reloading does not activate errors.

When designing experiments we have to take into account the operational profile of the analyzed applications. It is important to assure compatibility of the test profile with the operational one. For this purpose we can use statistics of input data and module utilization (activity ratio AR_r). In order to limit the number of experiments, we select the representative test scenarios to cover all possible situations. Depending upon data combinations we may have different resource activity, program and data flow, which influence significantly program sensitivity to faults. In Carreira et al. (1998) some correlation between error

detection capabilities and data combinations has been reported and explained in reference to program flow fluctuations. In our experiments (Section 4) we found that data impact on fault sensitivity is much more complex. Let us consider three classes of data processing:

1. finding relations between data objects (e.g. $a < b$, $a \bmod(k)\, b = c$),
2. finding data object properties (e.g. set cardinality) and data objects fulfilling some criteria (e.g. $\max\{a, b, c.....k\}$),
3. performing complex calculations involving all data.

In the first case many faults will not influence the final result. Moreover, the fault effect may depend upon the values of data e.g. small values of data can change easily their relations due to faults (compare Section 4). Similar observation holds also for type 2 data processing. Complex and pipelined calculations are more susceptible to erroneous results due to faults. In practice, we have a mixture of different data processing activities - the same algorithm can be implemented in different ways with higher or lower use of intermediate variables (for partial results), different resource activity etc.

When selecting test data for experiments we have to assure representativeness of the test scenarios. In this process some functional or structural program coverage measures are helpful. The functional coverage is related to specifications of the program modules, their functions and mutual interactions, whereas the structural coverage relates to program data and control flow (Briand and Pfahl, 2000). We use a specially developed tool, which measures such structural features as:

block coverage – coverage of blocks composed of code fragments without branching (as program is composed of branch free segments of code comprising entry and exit points);

decision coverage – measures the fraction of decisions executed during testing;

c-use coverage – counts the number of combinations of an assignment to a variable and the use of this variable in a computation that is not part of a conditional expression;

p-use coverage – counts the number of combinations of an assignment to a variable, the use of this variable in a conditional expression, and all branches based on the value of the conditional expressions;

all-use coverage – c-use or p-use;

du-path coverage – counts the number of paths from the definition of a variable to its use, which contains no redefinition of this variable.

Structural coverage measures are considered here as well correlated with fault detection capabilities. Many experiments proved that reducing the test set of the application in such a way that the test coverage is stable influences marginally the effectiveness of the test. In Section 4 we shall give some illustrations of using coverage measures to optimize test sets for fault injection experiments. For calculation oriented programs, this relates to representative values and relations of the data items only. Transaction oriented applications

need also specification of the tested transactions or operation sequences. Real time applications depend strongly on the controlled object's behavior and may require environment simulators (Derezińska and Sosnowski, 2002; Gawkowski and Sosnowski, 2001b).

Having specified the test profile, we face the problem of choosing fault injection moments (fault triggering). These can be related to the program execution time, executed instruction, referenced memory location etc. In time-triggered injections we have the knowledge of the execution time and inject faults within this period e.g. at random time moments with equal distribution. In code-triggered injections various strategies are possible:

CS1 – equal distribution within the executed code;
CS2 – equal distribution restricted to the executed instructions with specified features (e.g. ALU related instructions, branches, FPU instructions);
CS3 – more sophisticated profiles of distributions.

In CS1 we can define segments of the code which will be disturbed, select specific loop iterations etc. It is important to note that time- and code-triggered fault injections may have quite different stressing capabilities. For example, in applications with some small program segment executed for a long time (typical for many microcontroller applications), time-triggered injections will result in stressing mostly this segment of the code. Code-triggered injections give the possibility of better instruction coverage and assure better experiment controllability. Moreover, execution of operating system functions or other supporting procedures may disturb time-triggered injections. In code triggering we can specify code areas which are of interest for submission to fault injections. Limiting fault triggering to specified instruction codes gives the possibility of checking the impact of specified functional blocks (e.g. FPU, ALU). Moreover, it allows us to mimic permanent faults of the instruction sequencer, decoder etc.

Some comment is needed about fault injections into the data area. It is reasonable to disturb only the used data area. Hence during the golden run, we have to identify such areas. Some applications deal with a large size of the data area (operations on files, documents), moreover the data structure may be quite complex. In such cases we correlate fault injections with these structures. For example, while disturbing documents we distinguish control and data fields, specify different distribution strategies (e.g. fault bombardment in specified subareas at the beginning, in the middle and at the end of the document, see Gawkowski and Sosnowski, 2002b).

Another problem relates to the test result analysis. For simple calculation oriented applications the correct result is unique. In the case of real-time applications quite often some fluctuation of generated signals (in value and time) is acceptable. More difficult is dealing with the indeterminism, which appears in applications with parallelism or multithread processing, like the programs related to communication protocols. In such a case we define various classes of system responses and use specially designed (application dedicated) supporting

procedures. In these situations the general result classes: correct (C), incorrect (INC), system exceptions (S) and user messages (U) are appropriately extended. In practice, it is most important to distinguish various levels of correctness or incorrectness (Section 4.2).

The result qualification is based on the comparison of the test outcome with the golden run reference data. In calculation oriented applications the final result is uniquely defined and can be compared. For long calculations this leads to long experiment duration, each fault injection needing complete program execution. Some optimization is possible by checking partial results. However, this approach needs more sophisticated collection of reference data during the golden run. Quick identification that the injected fault does not activate an error or that the error is masked allows us to inject another fault in the same program run. We may skip fault injections into non-active resources etc. Further acceleration of fault injections is made possible by starting test runs from intermediate points. For this purpose during the golden run intermediate system states have to be stored at specified checkpoints. This reduces the delay of fault injection while assuring full controllability of triggering points. Moreover, in this case fault effects can be identified sooner (by inspecting the system state in the next checkpoint).

In the case of real-time or transaction oriented applications it may be difficult to define the final result. If the analyzed system operates continuously (and delivers appropriate outputs), an injected fault can disturb some internal state of the system and its impact may be visible after a long delay (e.g. in a newly initiated function). Hence an important issue is to identify such states of the system and include them in the analysis. Another problem relates to acceptable output disturbances of short duration (typical for microcontrollers delivering outputs to elements with some inertia). Hence test result analysis may be quite complex and application oriented (see, e.g. Gawkowski and Sosnowski, 2001b).

The performed experiments can be targeted to checking fault susceptibility of the considered application or to the analysis of the effectiveness of error detection and fault tolerance mechanisms. In the latter case we face the problem of selecting appropriate benchmarks of applications, e.g. using extensively CPU functional blocks (a mixture of arithmetic integer or floating point operations, loops or branches with different ranges, different cache hit ratio etc.), memory or I/O intensive processing, transaction or data processing oriented. An important issue is also to collect reports on fault injection and to correlate the results obtained with fault injection distribution, executed program flow paths, resource activity etc. Our injectors deliver such statistics. All aspects of test strategies are illustrated in Section 4.

4. Simulation experiments

We have performed many fault injection experiments using FITS, MTFI and LI fault injectors (Section 2) for a wide spectrum of applications. The experience

gained is very useful in designing test scenarios. In this section we concentrate on three problems: test coverage (Section 4.1), result interpretation and experiment tuning (Section 4.2). The test coverage relates to input data test profiles, fault location and triggering strategies. These features have significant impact on experiment results. Knowing this impact makes it easier to interpret the results and optimize experiments. The presented experiments have been performed for a sample of programs and bit flip (transient) faults, which dominate in modern technologies (Vargas et al., 2003; Velazco, 2002). All programs have been executed in IBM PC Windows environment except for one (clearly specified) in Linux.

4.1. Test coverage problems

In fault injection experiments an important issue is the definition of the fault stress profile, in particular, the selection of input data, the strategy of injecting faults (in time, space, and type).

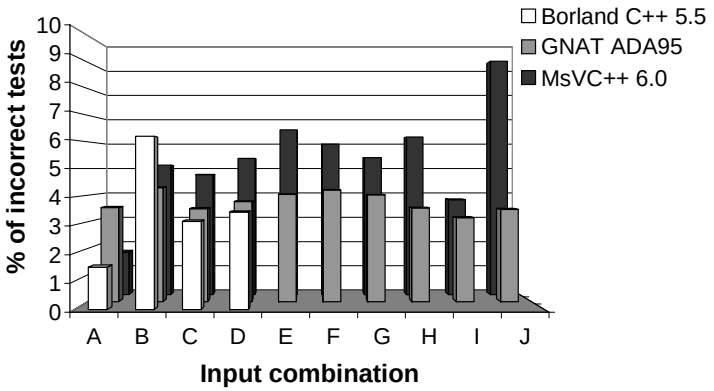

Figure 1. Incorrect results for faults injected into registers (program Qsortf)

Let us start with the problem of fault susceptibility to input data. In Fig. 1 we give the percentage of incorrect results for the Qsortf program (quick sort of floating point numbers) disturbed by transient faults injected into registers. The experiments have been carried out for the same program compiled with Borland C++, GNAT ADA and MSVC++ compilers. All experiments (random fault injections) have been repeated for 10 different input data combinations differing in the input data set ordering. For faults injected into the code, incorrect results fluctuated in the ranges 22-25.5%, 12-16% and 10-11%, respectively for Borland C++, GNAT ADA and MSVC++. For faults injected into data area, incorrect results fluctuated in the ranges: 22-26%, 8-15.2% and 9-11%, respectively. We observe different fault susceptibility depending upon program compilation and also some fluctuation of results as a function of the input data. This confirms

the need to support the selection of the input data with some tools, so as to cover all instructions, program control flow etc.

In order to facilitate selection of representative test cases, we can use test coverage measures (Section 3). In Fig. 2 we illustrate basic coverage measures for up to 15 test data sets of the Qsortf program. It was easy to assure the coverage level exceeding 70% using single data combination A - test 1. By adding two data combinations B and C, we assured a significant coverage increase (test 3). Subsequent combinations D - J practically did not change the coverage. Further increase has been achieved with specially developed four additional combinations (tests 11 - 15 in Fig. 2). In general, we face the problem of assuring high-test coverage at the minimal number of test cases. So for the application considered we can eliminate test cases D-J and use the set composed of 7 test cases.

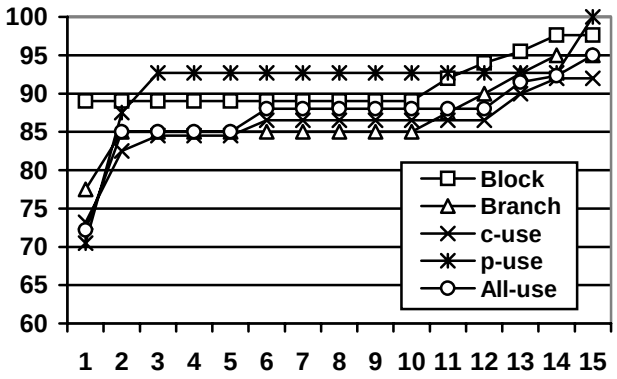

Figure 2. Test coverage (in percent) for Qsortf program as a function of 15 test sets.

For some applications with big impact of input data on program control flow (e.g. real-time applications - see Gawkowski and Sosnowski, 2001b, 2002b) we can observe significant test result differences depending upon the input. This effect is negligible in fixed calculations, e.g. matrix operations.

While designing experiments we have to specify the places where to inject faults (registers, RAM segments etc.). In the referenced papers the authors publish mostly the averaged results for random injections. Such information is not precise and it does not show dominant fault susceptibility resources in the analyzed program. Hence, it is reasonable to disturb specific functional blocks, e.g. selected registers, memory locations, ALU, FPU. These blocks have different impact on fault susceptibility due to various activities in different applications. This is illustrated in Table 1, which shows fault susceptibility of CPU registers for four applications: MATM (multiplication of matrices), Qsort

(sorting integer numbers), MAX (finding maximum value in a given set of integer numbers), INT1 and INT2 (a mixture of integer calculations performed in C and C++, respectively). For the MAX application we give results for two different input combinations (A and B).

Table 1. Percentage of correct results (C) versus register non-activity ratio (NA)

	MATM	Qsort	MAX-A	MAX-B	INT1	INT2
Faults injected into register EAX						
C	46.1	92.4	20.6	94.7	42.1	61.3
NA	*45.4*	*87.3*	*3.5*	*3.5*	*41.5*	*60.2*
Faults injected into register EBX						
C	10.4	99.8	7.9	12.2	60.1	100
NA	*7.8*	*8.9*	*2.6*	*2.6*	*60.2*	*100*
Faults injected into register ECX						
C	0.2	79.7	26	8.8	28.2	63.3
NA	*0.0*	*67.8*	*1.7*	*1.7*	*26.8*	*55.6*

Table 1 shows the percentage of correct results (C) for faults injected into three registers. In italic we give also the non-activity ratio (NA) of the registers (calculated by the injector - Section 2). For calculation oriented applications (MATM, INT1, INT2) we observe very good correlation between C and NA. In Qsort and MAX programs correct result percentages exceed the non-activity ratio, which is caused by specific data relations (e.g. result comparison) which are less susceptible to faults. The Qsort program sorts the input data set in a loop by checking relations between elements and exchanging their positions if needed. Register EBX is used to store the indices of the compared elements. In the experiment the input data are preliminarily sorted, so that the faults injected into the EBX register resulted in irrelevant additional iterations (without influencing the final result). Hence despite the low inactivity ratio (8.9%) of the register EBX we obtain high percentage of correct results (99%). In the MAX program register EAX is used to store the currently identified maximum value in the analyzed input vector, and its inactivity ratio is low (3.5%). By disturbing this register with transient faults we obtain 20.6% of the correct results for the input vector A and 94.7% for vector B. Vector B comprises elements with high value and the maximum one was the last one verified in the loop. The result is determined mostly by the last iteration, and so the injected faults have low impact on the final result (94.7% correct results).

Another interesting result has been obtained (with LI injector) for the application Qsortf, which sorts floating point numbers and uses the FPU unit. In spite of significant percentage of floating point instructions (14%), the faults injected into FPU registers had low impact on the final result. We have injected

faults into the following registers: CWD (control word: specification of the stack top, FPU precision and rounding strategy), SWD (FPU status word: identified exceptions and comparison results), STK data stack registers (8x80-bit registers) and TWD stack status register (for each stack register two bits specify its content: valid, invalid, empty, zero).

We have obtained very high percentage of correct results (Table 2). Disturbing of the control register had practically no impact on the results of comparison (hence it is not shown in Table 2), while disturbing of the status register is critical only during comparison instructions, moreover only 3 bits are critical. Bits related to exceptions are sensitive, if exceptions are enabled. The FPU data stack comprises eight registers and practically only one or two are used. They are sensitive to faults during data comparisons. We also observed that disturbing of the little significant bits of the mantissa has low impact on results. Most of injected faults were not critical for the realized algorithm.

Table 2. Detailed fault injection results for program Qsortf

Fault location				
SWD	TWD	STK	RND	INS
C – correct results				
99.76	95.22	98.56	98.56	21.8
INC – incorrect results				
0.24	4.78	0.24	1.20	11.7
S – system exceptions				
0.0	0.0	1.2	0.24	62.1
T – time-out				
0.0	0.0	0.0	0.0	4.4

In order to get a better insight into the fault effects, we injected faults into the specified bits of some registers. For instance, three bits of the CWD register indicating the top of the stack were the most sensitive to faults (30-50% incorrect results), whereas the remaining bits practically did not show any impact. This proved that practically only the top of the stack was used for storing data. In the RND column we give the results for faults injected at random in all FPU registers. They confirm high robustness of FPU to faults but this is due to low percentage of resource utilization. More sensitive to faults are instructions within the program code (INS).

In the literature the authors present only averaged results for faults injected into the code, and so tracing of fault effect propagation is not possible. In our simulators we are able to perform fine-grained analysis targeted to finding the most critical code segments, instructions etc. or to characterization of the architectural platform used. For Intel IA-32 architecture we have found that different instructions have different fault susceptibility, moreover, code bits 7

and 15 are less susceptible than other bits. More results are given in Gawkowski and Sosnowski (2004).

Let us consider the third aspect of the experiment coverage, namely fault triggering strategy (related to execution time or program code - Section 3). In time-triggered fault injections an important issue is to check the obtained distribution of fault injections. We illustrate this for the Mqsort program performing sorting in two threads. This program uses some procedures from the dynamically linked library (DLL), related mostly to thread management. Assuming 100 elements in the input set and random distribution of the triggering points in time, we obtained (using the MTFI injector) a somewhat strange results (Table 3). For 93% of instructions there were no more than five injections per instruction and these injections contributed only 6.94% of all injections. The most frequent triggering points related to only 2.38% of instructions of the whole program and these injections contributed 89.9% of all injections. Deeper analysis proved that most of these instructions were included in the DLL procedures. In the case of sorting a larger input set (100,000 elements), we obtained a better distribution due to relatively lower impact of DLL procedures. In particular 1-10 and 11-20 fault injections related to 87.03% and 39.64% of instructions, respectively. For the remaining instructions (12.97%) the fault injection frequency was 11-208 and these injections contributed 60.36% of faults.

Table 3. Distribution of fault injections into instructions (program Mqsort)

Number of fault injections per instruction	Percentage of covered program instructions	Percentage of all injections
1-5	93.54%	6.94%
6-10	2.04%	0.92%
11-15	0.68%	0.56%
16-20	0.68%	0.66%
21-25	0.68%	0.96%
26-1609	2.38%	89.96%

In code related triggering injections we have better controllability of distribution of fault injection moments, and so equal distribution can be easily assured, moreover fault injections can be limited to the considered application (excluding the DLL code or the invoked functions of the operating system) or even its specified segment. The latter issue is important in real-time applications where some program segments are quite often executed frequently and other ones scarcely. Hence, equal distribution of fault injection into executed (dynamic) instructions could result in a very large number of experiments needed to assure statistical significance of results. Thus, it is better to analyze independently the fault susceptibility of segments for different operational profiles (usage frequency) and then calibrate the final results (Section 4.2).

The results presented relate to bit flip faults (XOR). In Gawkowski and Sosnowski (2002b) we have analyzed the impact of the fault type (stuck-at, XOR), multiplicity and duration. Long duration faults are easier to detect, since many registers show asymmetric sensitivity to stuck-at-0 and stuck-at-1 faults. For practical purposes it is reasonable to concentrate on transient (XOR) faults, which create most problems.

4.2. Experiment interpretation and tuning

As it was shown in Section 4.1, we can obtain different experimental results depending upon the assumed fault stress strategy etc. Hence interpretation of these results should be done carefully in association with the experiment and application features. While defining the test strategy we should take into account the goal of the performed experiments and possibilities of experiment acceleration. We will discuss these issues in the sequel.

At the beginning we have to comment on the problem of result qualification. In the literature the authors deal with the general result classes: C, INC, S and T (Section 4.1) and they consider applications with unique correct results. Quite often some limited result accuracy is accepted. Moreover in many applications it is reasonable to use a fine-grained classification of system responses. We illustrate this for a distributed communication protocol (Table 4).

Table 4. Test results for transmission protocol in a grid network system

Fault location	Test result classes					
	C1	C2	C3	I1	I2	S
Reg.	67.3	0.55	0.05	0.4	0.4	31.3
Data	82.5	2.3	0.7	5.6	3.6	5.3
Code	17.7	0.5	4.0	7.2	11.6	59.0

Here we distinguish three correct and two incorrect result groups (Derezińska and Sosnowski, 2002):

C1 – correctly transmitted data and correct system behavior,
C2 – correctly transmitted data and some disturbances in internal states of protocol processes,
C3 – correctly transmitted data and some disturbances in control messages of the protocol,
I1 – incorrect data transmissions and correct behavior of the protocol and the system,
I2 – incorrect data and protocol messages.

The most important data is received correctly for C1-C3 cases (and partially for I1-I2). The result qualification can be further extended (finer granularity - see Derezińska and Sosnowski, 2002).

Similar problems arise for real-time, database and other non-computation oriented applications (Gawkowski and Sosnowski, 2002b). In the case of a car immobilizer (Gawkowski and Sosnowski, 2001b) we have defined the correct behavior as the one which was consistent with the non-faulty behavior at the level of the generated output signals. The inconsistencies of up to two consecutive control loop iterations (100ms) were accepted due to some inertia of the system. In general, various subclasses of correct results (differing in value or delivery time) and incorrect results can be defined by taking into account the fault severity levels related to the application analyzed. For facilitation of the result identification the developed fault injectors comprise some special mechanisms.

System exceptions and messages generated by the applications can also be classified in a more detailed way. This allows us to check the effectiveness of hardware fault detection mechanisms. For Intel IA-32 architecture we found that most exceptions were generated by memory access violation detector (50-100%), the illegal instruction detector generated exceptions in 0-14% of all exceptions. The latter mechanism is much more efficient for Motorola processors.

Having defined result categories, we face the most important problem of defining test strategies. This relates to the distribution of injected faults in space and time, fault types etc. While evaluating system dependability we are interested in assessing the probability of incorrect, correct results etc. We can evaluate this with the SWIFI fault injector, although the relation between the physical fault model probability and the model used in the experiments should be defined. It is convenient to assume equal probability of physical faults for all hardware elements. Considering that the activity of available hardware resources is limited, we face the problem of injecting faults into unused circuits. This leads to high time overhead of experiments. It is more efficient to disturb the used resources and then calibrate results by taking into account the percentage of the chip area of the disturbed functional blocks (similarly as in Carderilli et al., 2002). Here we can disturb only those resources (or their segments) which may have a real impact on the considered application. The presented detailed analysis allows us to find such points, e.g. top of the stack, stack status register flags and specified status register bits in the FPU unit (see Qsorf application in Section 4.1).

For complex applications composed of many modules it may be easier to arrange fault injections for these modules independently. Here we have to find typical inputs/outputs for the co-operating modules, which could significantly improve experiment effectiveness. We illustrate this for the MIX2TMR application, which performs some integer calculations in three versions (triple redundancy), and the final result is delivered by a voting procedure. After having performed calculations for randomly chosen 10,000 input data we have got only 273 different results, which are used in voting. Moreover, dominant value is 0 (in more than 74% of cases). The percentage of division by zero exception was 1%. Such statistics show the possible reduction of the number of experiments for testing the voting procedure.

If we are interested in testing the effectiveness of fault detection or tolerance mechanism, we should concentrate on finding fault sensitive points and check the effectiveness of the used mechanisms. Equal distribution of injected faults into the whole program may hide such points. We illustrate this for a sample of applications with embedded fault detection and fault masking mechanisms. Such mechanisms can rely on data and program redundancy (e.g. triplication and voting), assertions etc., discussed in Gawkowski and Sosnowski (2001a, 2002b) and references therein. In Table 5 we give some results for the matrix multiplication program (MAMUL+) with embedded checksum column and row, a mixture of integer calculations with triplicated code and voting (MIX2TMR), and the Qsort program with final assertion checking (Qsort+). In all cases the detected faults invoke retry procedure. Apparently, the mechanisms used in Qsort+ are not so efficient as in MATMUL+ and MIX2TMR.

Table 5. Reducing fault susceptibility with software procedures

Fault location	MATMUL+		MIX2TMR		Qsort+	
	C	INC	C	INC	C	INC
EAX	99.9	0.1	100	0.0	83.1	7.2
ESP	39.1	0.0	40.7	0.1	34.6	3.1
EIP	86.1	0.9	98.8	0.4	90.4	6.6
Code	86.5	1.0	86.3	0.2	20	9.3
Data	51.6	0.4	100	0.0	29.7	53.4
Reg.	88.2	0.0	82.7	0.3	81.8	2.1

In classical experiments, the authors use random fault injections in code, registers and data. In the case of fault hardened systems, this approach usually gives high percentage of detected or tolerated faults and no direct indication of the most sensitive points. Table 5 shows that faults injected in some specific resources (e.g. ESP - stack pointer register) are critical. Similarly, the faults injected into error detection or masking procedures result in a significant percentage of incorrect results. Duplication or triplication of internal variables etc. can improve it, as we have shown in Gawkowski and Sosnowski (2002b). To get a better insight into this problem, we can check fault detection and handling procedures independently. This is especially important if these procedures (e.g. comparison, voting, assertions) are used frequently within the analyzed application (fine granularity).

Another important issue is to find the representative experimental results. By repeating experiments with fault injection for the same program, we obtain test results that may differ (due to pseudorandom injections and partial test coverage). Hence we should evaluate the standard deviation of the results and decide how many fault injections should be performed. Fig. 3 shows the standard deviation (in percent) of experiment results (C, INC, S and T) for the

Qsortf application (compare Table 2) disturbed by faults injected into the code. The plots show simulation results in dependence of the fault triggering coverage (the percentage of instructions during whose execution faults have been injected). Input data for the tested application was the same in all experiments. The standard deviation is in the range: 1.8-6.1%, 1.2-4.8%, 2.1-7% and 0.8-3% for C, INC, S and T results, respectively. The standard deviation increases with code coverage decrease.

While comparing the standard deviation with average values of test results (24.4, 9.5, 62.2 and 4.0%, respectively), we can observe that the relative standard deviation is higher for results of lower average values. For the Qsort (quick sort of integers) application we performed similar experiments but faults were injected into the accumulator register EAX. In this case the standard deviation is lower than for faults injected into the code. This is due to the higher impact of disturbed instructions on program flow than on the register EAX. For the Qsort application the standard deviation of incorrect results ranged from 1.2 to 3.7% (for 5-100% coverage) and the average percentage of incorrect results was 5.6%. It is worth noting that the average test results C, INC, S and T for Qsortf application were respectively in the ranges: 20.51-24.36%, 9.53-12.51%, 62.16-62.63% and 3.95-4.69% for 10-100% triggering coverage. The performed experiments confirmed that for complex applications we could use partial coverage of fault injections and obtain representative results. The number of injected faults should be tuned to the required confidence interval of the results according to the classical statistical theory.

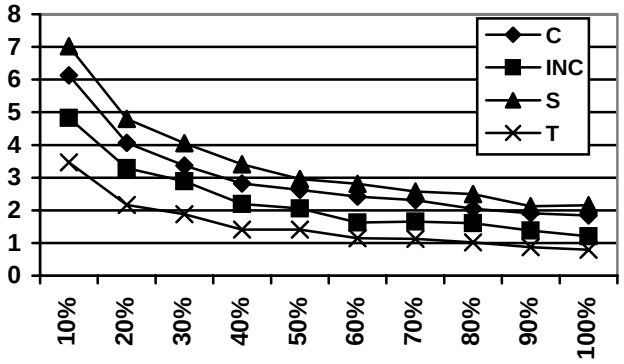

Figure 3. Standard deviation of results (percents) for fault injections into Qsortf code.

While designing experiments with fault injection, we face the problem of defining fault classes, their locations etc. This should be done in correlation with the objective of the performed analysis. In particular, the designer may be

interested in the effectiveness of fault detection and fault tolerance mechanisms etc. In most cases a large number of faults is needed to obtain representative and statistically significant results. Hence, these faults must be generated in an automatic way, e.g. pseudorandomly (various parameters of the generator can adapt this process appropriately). Since the time overhead of experiments could be significant for complex programs, the fault selection process is important. We should inject faults into locations with a high probability of being activated. This relates also to the problem of selecting appropriate input data sets for the analyzed program. On the other hand, if we inject faults randomly, we should correlate the obtained results with some properties of the analyzed application, e.g. resource activity (Section 4.1).

To evaluate the system fault coverage (e.g. tolerated or detected) on the basis of fault injection experiments, we have to take into account the fault injection space (sp) and time (tp) profiles as well as the realistic fault profile (e.g. equal distribution in time and device cross area for radiation faults). Hence the system fault coverage is defined as:

$$FC = \sum_i sp_i \; tp_i \; FC_i$$

where FC_i is the i-th resource fault coverage in the experiment, and i runs over all system resources e.g. data, code segments, registers. In fault hardened systems for some resources FC_i can be equal to 100%, so in the experiments we can concentrate on critical resources only, e.g. fault-handling procedures.

If we are interested in assuring a high level of system dependability, we should identify fault sensitive points and check the effectiveness of the used mechanisms. So, the test scenarios should assure stressing only those code areas, registers (or their segments) etc. which really might impact upon the considered application. In Section 4.1 we have analyzed the impact of FPU registers on test results for Qsortf. The detailed analysis showed the most critical points related to the top of the stack, its status register flags and specific FPU status register bits. Further acceleration of stressing may require selection of the most critical triggering points.

For complex applications fault injection experiments may lead to long duration. Hence some optimization is required (Section 3). We can limit experiments to fault handling procedures or inject faults only during the active periods of disturbed resources or even just before the last operations (these points are identified in our algorithm given in Section 2). To get the overall dependability evaluation we have to normalize the obtained fault injection results with the ratio of activity periods etc.

5. Conclusion

Large complex applications and time constraints make exhaustive fault injection experiments impractical and difficult for interpretation. Hence an important is-

sue is to design carefully test strategies. This includes specification of fault stresses (fault types, space and time distribution), test profiles (related to control and data flow in the tested application) and result qualification. We have illustrated this for several sample applications.

The pre-injection analysis of the application properties (control flow features, operational profile, resource activity etc.) is helpful in finding efficient test strategies. The collected preliminary statistics from pseudorandom fault injections as well as from the analyzed application (in the golden run) can be used for the final experiment tuning. Detailed reports and statistics from the performed experiments assure better result interpretation. This is especially important in the case of experiments with limited controllability or some nondeterminism like time-triggered fault injections or multithreaded applications. Such analysis facilitates confirming reliability and finding week points in fault hardened applications being designed.

By combining fault injection profiles with the real fault distribution, we can estimate system fault susceptibility. Carderilli et al. (2002) analyzed the relation between SWIFI faults and irradiation faults by taking into account the functional block cross-section. An interesting analysis of fault effect propagation in CPU functional blocks is given in Kim and Somani (2002), this problem requiring further research.

Acknowledgement

The authors wish to thank A. Derezińska, P. Włoda-wiec and T. Czarnecki for their help in arranging some experiments. This work was supported by grant KBN 4T11C 049 25.

References

Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E. and Leber, G.H. (2003) Comparison of physical and software implemented fault injection techniques. *IEEE Trans. on Computers* **52** (9), 1115-1135.

Bernrojo, L., Gonzales, I., Corno, F., Reorda, M.S., Squillero, G. and Lopez, C. (2002) An industrial environment for high level fault tolerant structures insertion and validation. *Proc. 20th IEEE VLSI Test Symposium*, 229-236.

Baldini, A., Benso, A., Chiusano, S. and Prinetto, P. (2000) 'BOND': An interposition agents based fault injector for Windows NT. *Proc. IEEE Defect and Fault Tolerance in VLSI Symposium*, 387-395.

Briand, L.C. and Pfahl, D. (2000) Using simulation for assessing the real impact of test coverage on defect coverage. *IEEE Trans. on Reliability* **49** (1), 60-70.

Carderilli. G.C., Kaddur, F., Leanori, A., Ottavi, M., Pontarelli, S. and Velzaco, R. (2002) Bit flip injection in processor based architectures: a case study. *Proc. IEEE On-Line Testing Workshop*, 117-128.

Carreira, J., Madeira, H. and Silva, J.G. (1998) Xception: a technique of the experimental evaluation of dependability in modern computers. *IEEE Trans. on Software Engineering* **24** (2), 125-136.

Chen, M., Tsai, T.K. and Iyer, R.K. (1997) Fault injections and tools. *IEEE Computer* **30** (4), 75-56.

Choi, G. and Iyer, R. (1992) Focus: an experimental environment for fault sensitivity analysis. *IEEE Trans. on Computers* **41** (12), 1515-1526.

Constantinescu, C. (2003) Experimental evaluation of error detection mechanisms. *IEEE Trans. on Reliability* **52** (1), 53-57.

Cukier, M., Powell, D. and Arlat, J. (1999) Coverage estimation methods for stratified fault injection. *IEEE Trans. on Computers* **48** (7), 707-723.

Derezińska, A. and Sosnowski, J. (2002) Experimental checking of fault susceptibility in a parallel algorithm. *Proc. IEEE Int. Conf. on Parallel Computing in Electrical Engineering*, 33-38.

Folkesson, P., Svensson, S. and Karlsson, J. (1998) A comparison of simulation based and scan chain implemented fault injection. *Proc. IEEE Fault Tolerant Computing Symp.*, 284-293.

Gawkowski, P. and Sosnowski, J. (2001a) Experimental evaluation of fault handling mechanisms. *Lecture Notes in Computer Science* **2187**, Springer-Verlag, 109-118.

Gawkowski, P. and Sosnowski, J. (2001b) Evaluation of fault effects in programmable microcontrollers. *Proc. 5th IFAC Workshop PDS 01*, Pergamon, 121-126.

Gawkowski, P. and Sosnowski, J. (2002a) Experimental validation of fault detection and fault tolerance mechanisms. *Proc. IEEE Int. High Level Design Validation and Test Workshop*, 181-186.

Gawkowski, P. and Sosnowski, J. (2002b) Using software implemented fault inserter in dependability analysis. *Proc. IEEE Pacific Rim Int. Symposium on Dependable Computing*, 81-88.

Gawkowski, P. and Sosnowski, J. (2004) Evaluation of transient fault susceptibility in microprocessor systems. *Proc. of Digital System Design Euromicro Symposium*, IEEE Comp. Soc., 432-439.

Kanawati, G., Kanawati, N. and Abraham, J. (1992) "FERRARI": a tool for the validation of system dependability properties. *IEEE Proc. Fault Tolerant Computing Symp.*, 336-344.

Kim, S. and Somani, A.K. (2002) Soft error sensitivity characterization for microprocessor dependability enhancement strategy. *Proc. IEEE Dependable System Network Symposium*, 416-428.

Leveugle, R. (2000) Fault injection in VHDL descriptions and emulation. *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, 414-419.

Madeira, H. and Silva, J.G. (1994) RIFLE: a general purpose pin-level fault injector. *Lecture Notes in Computer Science* **852**, Springer-Verlag, 199-216.

Madeira, H., Some, R.R., Costa, F.D. and Rennels, D. (2002) Experimental evaluation of a COTS system for space applications. *Proc. IEEE Int. Conference on Dependable Systems and Networks*, 325-330.

Rebaudengo, M. and Reorda, M.S. (1999) Evaluating the fault tolerance capabilities of embedded systems via BDM. *Proc. IEEE VLSI Test Symposium*, 452-459.

Samson, J. (1998) A technique for automated validation of fault tolerant designs using laser fault injection. *Proc. IEEE Fault Tolerant Computing Symp.*, 162-187.

Segall, Z. and Lin, T. (1988) FIAT: fault injection based automated testing environment. *Proc. IEEE Fault Tolerant Computing Symp.*, 102-107.

Sieh, V., Tschade, G. and Balbach, F. (1997) Verify: evaluation of reliability using VHDL-model with embedded fault descriptions. *Proc. IEEE Fault Tolerant Computing Symp.*, 32-36.

Sosnowski, J., Gawkowski, P. and Lesiak, A. (2003) Software implemented fault inserters. *Proc. of IFAC PDS2003 Workshop*, Pergamon, 293-298.

Tsai, T.K., Hsueh, M.Ch., Zhao, H., Kalbarczyk, Z. and Iyer, R.K. (1999) Stress based and path based fault injection. *IEEE Trans. on Computers* **48** (11), 1183-1201.

Vargas, F., Brum, D., Prestes, D., Bolzani, L. and Lettmin, D. (2003) On the mitigation of conducted electromagnetic immunity by means of SW-based fault handling mechanisms. *Proc. IEEE Latin America Test Workshop*, 130-135.

Velazco, R., Corominas, A. and Ferreyera, P. (2002) Injecting bit flip faults by means of a purely software approach. *Proc. IEEE Int. Defect and Fault Tolerance in VLSI Symposium*, 108-116.