Comparing and Validating Measurements of Dependability Attributes

Daniel Skarin, Raul Barbosa, Johan Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
$\{$*skarin, raul.barbosa, johan*$\}$*@chalmers.se*

***Abstract*—This paper investigates sources of uncertainty in measurement results obtained using three different fault injection techniques. Two software-implemented and one test port-based technique are characterized and compared. The three techniques can be used to inject the same faults, which are defined in a shared database. Due to the uncertainties associated with the techniques, which we identify and discuss, the results of injecting a given fault may differ to some extent. The paper analyzes the results of using the three techniques to inject faults into two experimental targets: a brake-by-wire controller and a partitioning operating system. The objective of the experiments is to determine whether the results of the different techniques are metrologically compatible and, consequently, meaningful when disseminated and compared. Our observations indicate that, even though the outcome of many individual experiments is affected by uncertainties, the three techniques produce similar average results over a large number of experiments.**

I. Introduction

The role of computer systems and infrastructures in modern society has increased the importance of thoroughly assessing their dependability. To this end, one may use analytical as well as experimental approaches, depending on the type of system under evaluation and its development stage. An effective approach to the experimental evaluation of dependability attributes is fault injection, which is recognized for its ability to evaluate real systems, prototypes or models of target systems. It is suitable for fault removal, by increasing the test coverage for fault handling mechanisms, and for fault forecasting, by providing probabilistic information on how a system will perform after deployment [1].

In spite of steady advances, quantitative evaluation of dependability attributes remains a complex task lacking standard processes and techniques. This has recently motivated a greater awareness for the need to incorporate metrology concepts into dependability studies [2]. To do so, it is necessary to regard any tools such as those used for fault injection and data collection as measuring instruments, and to scientifically obtain measurement results that can be compared and reproduced by others.

The difficulties in comparing and reproducing results of dependability measurements arise from a wide range of uncertainties associated with measurement procedures, instruments and target systems. Examples are non-representative sampling with respect to workload, faultload, and time instants chosen for collecting measurements; approximate implementations of a measurand's definition; and instrumental uncertainty due to software defects in any tools used in the process. Such factors cause variations in measurement results. It is therefore necessary to estimate the magnitude of such variations and to assess the *metrological compatibility* of measurement results [3], *i.e.*, whether different measurements, possibly obtained in different studies, relate to the same measurand or not.

This paper examines the metrological compatibility of measurement results obtained using different fault injection techniques. Over the years, numerous fault injection tools and techniques have been described in the literature (with surveys in [4] and [5]), aiming at introducing errors in systems and measuring relevant aspects of their operation, *e.g.*, error coverage of specific mechanisms or the latency with which errors are handled. Measurands such as coverage and latency are usually well defined. However, due to the uncertainties associated with each fault injection technique, the results of injecting a given fault may differ to some extent. This paper analyzes the results of injecting a set of faults using three different techniques, in order to determine the factors that can affect the compatibility of results.

The first is a test port-based technique implemented using the Nexus standard. Modern microprocessors are often equipped with I/O ports for testing and debugging, known as test ports, that provide the ability to control and monitor the execution. These features make it convenient to use test ports such as Nexus for fault injection [6]. The second technique provides software-implemented fault injection (SWIFI) using debugging and monitoring functions available in advanced processors. It places the fault injection code in exception-handling routines intended for debugging – an approach adopted in [7] and [8]. The third technique does not require any specific hardware, and injects faults by instrumenting the executable file with fault injection code before it is downloaded to the target system. Like the previous one, this is a SWIFI technique, since the fault injection code is executed on the target system (an approach frequently used by authors in the field [9], [10], [11]).

The main contributions of this paper are the following:

- It identifies the sources of uncertainty observed in our experimental platform and discusses possible means to minimize their impact.

- It describes an experimental evaluation of the metrological compatibility of the results provided by the three techniques. Previous research has studied uncertainty with respect to timing measurements [12]. We analyze the outcomes of the experiments focusing on the value domain. The methodology adopted consists in injecting the same set of faults (defined in a shared database) with the three techniques, and comparing the measurement results.
- It addresses the issue of validating the results of dependability studies (one which is often neglected). Complex tools and techniques, requiring some form of validation, are usually involved in dependability measurements. By comparing the outcome of diverse fault injection methods we can increase our confidence in the correctness of their implementation, and therefore reduce the instrumental uncertainty.

The paper is organized as follows: The techniques for fault injection examined in this paper are described in Section II and sources of uncertainty are discussed in Section III. Section IV describes the results of the experimental evaluation, targeting a brake-by-wire controller and a partitioning operating system. Those results are discussed in Section V and the conclusions drawn from this study are summarized in Section VI.

II. The fault injection techniques

We used the GOOFI-2 fault injection tool [13] for the experimental evaluation. The tool supports the three techniques evaluated in this paper: *test port-based injection*, *exception-based injection* and *instrumentation-based injection*. All techniques are capable of emulating soft errors, which are disturbances in the state of integrated circuit elements, caused by transient hardware faults [14].

GOOFI-2 emulates soft errors using the single bit-flip error model. Errors are uniquely defined using time-location pairs. The location corresponds to a bit in a register or a memory element; the time corresponds to the execution of an instruction, identified by its code address together with an invocation count that specifies the number of iterations before injection. The following sections describe how the different techniques inject errors into a microprocessor.

A. Test port-based injection

The test port-based technique uses a Nexus interface to inject errors into Freescale's MPC565 and MPC5554 microprocessors. Nexus is a standard that defines an interface along with a minimum set of features for debugging. These features include support for starting and stopping the microprocessor, setting instruction and data breakpoints, and reading and modifying registers and memory values. The features provided by the Nexus test port are well suited for fault injection purposes.

A major advantage of the test port-based technique is that errors can be injected into registers and memory without requiring any modifications to the program running on the microprocessor. Techniques that use a Nexus port to inject errors can potentially achieve a low level of temporal intrusiveness as the Nexus standard provide support for real-time access to memory. In practice, however, it is difficult to fully exploit all features offered by Nexus without compromising key properties such as repeatability of experiments.

B. Exception-based injection

Exception-based injection is a SWIFI technique that takes advantage of debug features available in the MPC565 microprocessor. Faults are injected into registers and memory by exception routines, which are activated by hardware breakpoints configured for specific instruction addresses. The approach is similar to the one used by Xception [7] and RT-Xception [8].

The exception routines are specifically designed for each injected fault. At the start of a fault injection experiment, GOOFI-2 downloads the machine code implementing the exception routine via a *fault injection console* running on the target system. This means that the target system must be modified in such a way that the fault injection console is first to execute at startup. After the exception routine is placed on the target system, the fault injection console sets the hardware breakpoint used to trigger the fault injection, and starts the execution of the target program.

Memory values are logged at the end of an experiment. A hardware breakpoint that identifies the end of the experiment is set by the exception routine after the injection of an error. This breakpoint will cause the fault injection console to resume control of the microprocessor, and allow GOOFI-2 to read the values from memory locations in the target system. GOOFI-2 can also reach the fault injection console by resetting the MPC565 microprocessor.

Exception-based injection cannot be used together with a Nexus debugger. The reason is that the MPC565 does not allow the debug registers to be updated by software when the debugger is connected. This means that the debugger is not available for detailed analysis of "interesting" errors when exception-based injection is used. Another drawback of exception-based injection is that the program code of the target system must be extended with instructions for fault injection, which leads to higher intrusiveness compared to test port-based injection.

C. Instrumentation-based injection

Instrumentation-based injection is a runtime SWIFI technique that injects errors into registers and memory using software only. This is a general technique that is not limited to a specific microprocessor. The injection of an error is triggered by a *software breakpoint*, which is placed at the instruction that identifies the time instant when a fault should be injected.

The software breakpoint is inserted by replacing the instruction at the breakpoint address in the target program with a branch to a fault injection routine. The routine keeps track of the number of invocations, and injects the error when the specified invocation count is reached. The original instruction, which is replaced by the branch instruction, is executed before the instrumentation routine returns to the instruction following the software breakpoint. Our current implementation lacks support for setting software breakpoints at branch instructions.

The instrumentation is done by modifying the machine code for the target system's executable file. The fault injection routine is placed at memory addresses that are unused by the target program, and the software breakpoint is inserted by replacing the instruction at the breakpoint address. We use the same experimental setup for the instrumentation-based injection as for the test port-based injection. This makes it possible to use the Nexus port to program the instrumented executable to the microprocessor's flash memory, and to log values from the target system.

III. Sources of uncertainty

Generally speaking, uncertainty expresses doubt about the validity of a measurement result, and may also provide quantitative measures of that doubt. In that context, a measurement result is associated with an uncertainty parameter, which characterizes the dispersion of values that could be attributed to a measurand. In [15], general sources of uncertainty are listed. With no claim of being exhaustive, this section describes uncertainties (highlighted in italics) specifically in fault injection.

The use of representative workloads and faultloads is essential in fault forecasting and fault removal experiments. Hence, *non-representative sampling* is an important source of uncertainty. This makes the collected dependability measures meaningful of a system under normal operating conditions. Furthermore, if the results are to be disseminated, as pushed by dependability benchmarking initiatives [16], this is a source of uncertainty that deserves careful attention.

This source of uncertainty concerns not only the faultload and the workload, but also the time instants when data is collected on the target system, and the data items that are read. This means that the output of a given task must be sampled with a given frequency and that the data items that represent the output must be chosen appropriately.

Determinism of the target system is needed for results to be reproducible. It is also necessary to ensure repeatability, which is an important property of the fault injection tool. Repeatability refers to the ability to repeat the injection of a specific fault and obtain the same result. However, achieving a fully deterministic system is difficult due to, *e.g.*, rounding-effects in floating point values [17] and performance improving features in microprocessors such as caches and branch prediction.

Finite resolution of instruments affects measurements in the time and value domains. The binary representation of values in memory and registers can usually be accessed without losing precision. Nevertheless, resolution is a problem if a processor uses an internal precision that is higher than what can be observed by the fault injection tool. As computers operate on discrete values, resolution of instruments is mostly a concern when measuring time.

Measurements are often obtained using an *approximation of a measurand's definition*. In dependability studies there is usually a precise definition of what a system's correct behaviour is. This can be used to define properties that are interesting in any given case and appropriate measurands. In principle, it is possible to create good definitions of the measurands. However, at some point we must realize these definitions in practice by implementing, for example, a tool that classifies the outcome of each injection.

Due to the finite amount of detail that such an implementation can handle, approximations are often introduced in the realization of the measurand. If, for instance, we are observing the behaviour of a system and it produces correct output at the expected time, we may say that the system is reliable in the presence of faults. However, there may be latent faults in some parts of the system that are not considered as part of the output during the analysis.

Assumptions in the measurement procedure can lead to deviations in the measurements, which are caused by events other than the injection of a fault. Measurements for dependability evaluations are often obtained under the assumption that the target system is only affected by the injected fault. In reality, the target or measurement system can be affected by some unforeseen external event affecting the measurement results, or internal events affecting the target system, *e.g.*, design flaws in the hardware or software.

Instrumental uncertainty is often neglected in dependability studies. The tools and techniques involved in dependability measurements are usually complex, thus requiring some form of validation. The fault injection component should be verified against a reference component whenever available, to ensure that it does inject the intended faults. Furthermore, the components that analyze the data collected during each experiment may also contain software defects, leading to incorrect classification of experiments. Special care must be taken to develop reliable tools.

Initialization uncertainty can influence the outcome of an injected error. Processors are usually led to a known state at startup, but memory is normally not initialized in its entirety by target programs. After an error is injected, a program may come to use non-initialized values; in such cases the contents of unused memory locations and registers can influence the

```
1    ...
2    2130: lfs     f0,8604(r9)  # error injected in r9
3    2134: fcmpu   cr7,f13,f0
4    2138: bgt     cr7,2140
5    213c: b       2158
6    2140: # code for f13 > f0
7    ...
8    2158: # code for f13 ≤ f0
9    ...
```

Figure 1. Example of code affected by initialization uncertainty.

impact of the injected error. The parts that are used by the program in normal conditions are likely to have initialized values (otherwise the target system would probably contain some defects).

Figure 1 shows an example of code that can be affected by initialization uncertainty. An error injected into register `r9`, just before Line 2, can cause the floating point register `fr0` to be loaded with a value from a non-initialized memory location. If this faulty value represents a value that is outside the supported range of floating point values, the processor may raise an exception. On the other hand, if the incorrect value loaded into `fr0` is within the supported range, the compare instruction at Line 3 can instead cause the error to be masked. However, the error may cause data errors if the conditional branch at Line 4 is affected, or if the incorrect value in `fr0` is later used in calculations. Consequently, even if a specific error is injected several times, the outcomes of such experiments may differ depending on how a given memory location is initialized.

The problem of initialization is closely related to the repeatability of experiments. In order to ensure perfect repeatability of every experiment, the fault injection tool would have to copy the entire state of memory at start-up and restore it in each experiment. Even though this ensures repeatability, it does not solve initialization uncertainty, as we are interested in knowing how the results would vary if the system had a different state in the uninitialized memory segments.

Spatial intrusiveness reflects changes made to the system under test to allow the experimental evaluation. Such changes include adding instructions to the target program for the injection of faults, as in the case for SWIFI techniques. By doing such a modification, memory regions in the original program that are uninitialized, or initialized to some specific value, instead hold valid instructions or data in the modified program. The outcomes of fault injections in these two program versions can differ, as an injected fault in the modified program can access memory values that are not present in the original program. Spatial intrusiveness also arises if the memory layout of the target program is changed. This happens, *e.g.*, when a program is placed in a different memory type for the experimental evaluation.

The spatial intrusiveness of the three fault injection techniques differ significantly. The test port-based technique does not modify the target system, and the technique has no intrusiveness with respect to space. The exception-based and instrumentation-based techniques, however, both require that the target program is extended with additional instructions. The fault injection console used for the exception-based injection requires about 5kb of code and data for our implementation in C. The implementation of the instrumentation-based technique requires at most 21 instructions to be added. For the experiments with the partitioning kernel, 4kb of data and 4kb of code is needed due to the minimum page size of the memory management unit.

Temporal intrusiveness is especially important in real-time systems. Assessing the impact of faults in such systems requires the fault injection tool to have a negligible impact on the target system with respect to time. To obtain meaningful measurements, the temporal intrusiveness must be lower than the resolution used for time measurements.

We have investigated the temporal intrusiveness for the three techniques in detail. For the test port-based injection, it is mainly caused by the time needed for sending commands and data between the fault injection tool and the target system. The test port-based tool, which is a Java program running on Windows, communicates with the target system using a Nexus debugger and a development environment. The communication between these different components generates high and not entirely deterministic temporal intrusiveness, which is unsuitable for real-time systems.

The temporal overhead for the exception-based injection is caused by the execution of the exception routine. In our implementation, the exception routine requires at most 50 machine instructions to handle an exception, inject an error, and configure a breakpoint that is used to terminate the experiment. These instructions are executed only once for each injected fault. Thus, the temporal intrusiveness of exception-based injection is constant and very low.

The temporal intrusiveness of setting software breakpoints in the instrumentation-based technique may be high in some circumstances. A software breakpoint replaces the instruction that is used to trigger the fault injection routine. The overhead of this technique depends on the amount of times that the original instruction is executed. Table I shows an estimate of the temporal overhead, in number of instructions executed by the target system, of setting a software breakpoint for fault injection. The results were obtained for 10 workloads of the MiBench set of benchmarks [18], using the PIN binary instrumentation tool [19] on a IA32 processor to trace the instructions executed by each workload. The calculations are made considering that 21 machine instructions are executed each time the software breakpoint is reached (the maximum value observed in our implementation).

It is possible to observe that the average overhead is well under 1% for all workloads. However, all workloads have some instructions (usually less than ten) that are executed

Table I
INSTRUCTION COUNT OVERHEAD OF SETTING A SOFTWARE BREAKPOINT FOR FAULT INJECTION.

Workload	Instruction Count	Unique Instructions	Max. Iterations	Avg. Iterations	Max. Overhead	Avg. Overhead
basicmath	54 060 249	5 744	360 916	9 412	14.01%	0.37%
crc32	75 350 090	4 203	1 368 865	17 928	38.15%	0.50%
gsm-decode	8 887 763	8 615	31 920	1 032	7.54%	0.24%
ispell	8 356 837	14 459	187 384	578	47.09%	0.15%
jpeg-decode	7 038 219	16 222	65 536	434	19.55%	0.13%
pgp-sign	7 333 695	23 983	48 471	306	13.88%	0.09%
sha	12 462 696	11 377	102 333	1 095	17.24%	0.18%
stringsearch	163 965	3 179	14 592	52	186.89%	0.67%
susan-edges	2 119 798	6 691	7 220	317	7.15%	0.31%
typeset	68 019 206	57 116	482 721	1 191	14.90%	0.04%

very frequently – the instructions at the core of the innermost loop of the algorithm. Since those instructions have a very high number of iterations, they incur a large overhead when instrumented with a software breakpoint. We can therefore say that in rare cases the temporal intrusiveness of instrumentation-based injection is very high, even though for the vast majority of instructions the overhead can be considered low.

IV. EXPERIMENTAL EVALUATION

We investigate metrological compatibility of measurements results by analyzing the results of 10 000 fault injections targeting a brake-by-wire controller and a partitioning kernel. The goal is to identify uncertainties that cause differences in the results measured with the three techniques, focusing on the value domain, *i.e.*, the correctness of the results produced by the workloads.

A. Experiments targeting the brake-by-wire controller

This section presents results obtained by the three fault injection techniques using a prototype brake controller. The controller implements an Anti-lock Braking System (ABS). Such systems are used in road vehicles to maintain steering ability and decrease braking distance by preventing wheel-locking during braking. The prototype ABS controller used in this study has been specifically designed to be included in future brake-by-wire systems. As there is no mechanical back-up in such systems, the brake controller must be highly reliable and fault-tolerant.

1) Target system and experimental setup: The brake controller uses a combination of simple hardware and software mechanisms to detect and recover from soft errors. We have previously evaluated the effectiveness of this approach using extensive experiments conducted with the test port-based technique [20]. The error detection consists of a run-time check of a program variable holding the controller's state. In addition to this software check, the brake controller uses hardware exceptions to detect errors. Two different strategies are used to recover the brake controller from errors. When an error is detected by the software check for the controller's state variable, the brake controller rolls back to a previous state. For errors that are detected by hardware exceptions in the microprocessor, we restart the brake controller and reinitialize all variables in the brake controller program.

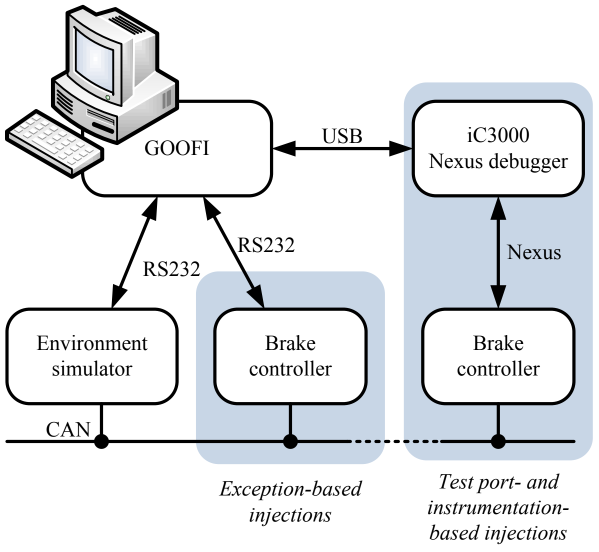

Figure 2. Experimental setup for the brake-by-wire experiments.

The brake controller is allowed to produce outputs different from the ones produced by a fault-free controller. This makes it possible to use a non-perfect error recovery, which is an approach that is valid for many control systems [21]. Outputs affected by errors, however, are not allowed to cause critical brake controller failures such as a loss of braking or a locked wheel. Incorrect outputs are only allowed to be produced if they have a benign effect on the system., *i.e.*, if they have no significant effect on the wheel being controlled.

Figure 2 shows the experimental setup that we use for the experiments with the brake-by-wire controller. GOOFI-2 is used to inject errors into an emulated brake system, which consists of two computer nodes based on the MPC565 microcontroller from Freescale. One node executes the brake controller program while the other node executes a simulation model of a vehicle, which includes models of a wheel, a brake pedal, a wheel speed sensor, and a brake actuator. The

Table II
SUMMARY OF PROGRAM OUTCOMES FOR THE BRAKE-BY-WIRE EXPERIMENTS.

Program outcomes	Test port-based	Instrumentation-based	Exception-based
Register errors			
No impact	228 (22.8%)	237 (23.7%)	223 (22.3%)
Data corrupted	764 (76.4%)	755 (75.5%)	775 (77.5%)
Detected, not recovered	6 (0.6%)	6 (0.6%)	0 (0.0%)
Crash	2 (0.2%)	2 (0.2%)	2 (0.2%)
Memory errors			
No impact	356 (35.6%)	356 (35.6%)	356 (35.6%)
Data corrupted	644 (64.4%)	644 (64.4%)	644 (64.4%)
Detected, not recovered	0 (0.0%)	0 (0.0%)	0 (0.0%)
Crash	0 (0.0%)	0 (0.0%)	0 (0.0%)

Table III
SUMMARY OF ERROR DETECTIONS FOR THE BRAKE-BY-WIRE EXPERIMENTS.

Error detections	Test port-based	Instrumentation-based	Exception-based
Register errors			
Hardware exceptions	488 (48.8%)	490 (49.0%)	519 (51.9%)
Software check	83 (8.3%)	83 (8.3%)	56 (5.6%)
No detection	429 (42.9%)	427 (42.7%)	425 (42.5%)
Memory errors			
Hardware exceptions	370 (37.0%)	370 (37.0%)	370 (37.0%)
Software check	35 (3.5%)	35 (3.5%)	35 (3.5%)
No detection	595 (59.5%)	595 (59.5%)	595 (59.5%)

brake system emulator executes for 4 s of real-time execution in our experiments. During this time, the simulation node executes code that models a vehicle with an initial speed of 30 m/s, and for which maximum braking is requested 10 ms after the start of an experiment. The two nodes exchange sensor readings and actuator commands via a CAN-bus, and these values are also saved to a workstation via a serial link (denoted RS232 in Figure 2).

The brake system emulator was developed as models using Simulink and Matlab from MathWorks, Inc. We used the TargetLink code generator from dSPACE GmbH to generate C code, and the GNU Compiler Collection (GCC) to create an executable file. The brake controller program also includes startup code for the MPC565 microprocessor and code for scheduling and the communication with the environment simulation node. The brake controller program used with the exception-based technique also includes code for the fault injection console described in Section II. Except for the startup code for the MPC565, the fault injection console does not modify any instructions or the memory layout of the original brake controller program.

2) Experimental results: We randomly selected 1 000 errors to be injected into registers and 1 000 errors to be injected into memory. As mentioned earlier, we define errors using time-location pairs. The time-location pairs were selected using a pre-injection tool, which ensures that errors are injected immediately before a register or memory element is read. This avoids injecting errors into "dead" registers, *i.e.*, registers that will never be read again. The time instants were selected among instructions of the brake controller program executed during the calculation of the brake actuator command for a single control loop. The locations were selected among general purpose registers and the data and stack segments of the memory.

Table II and Table III show a summary of the program outcomes and error detections for the in total 6 000 experiments with the brake-by-wire controller. The experiment outcomes were classified into the following groups:

- *No impact* – All outputs were identical to the ones produced by a fault-free brake controller.
- *Data corrupted* – The brake controller produced outputs different from the ones produced by a fault-free brake controller. As the brake controller uses a non-perfect error recovery, this group can includes experiments in which the injected error was detected by the software or hardware mechanisms.
- *Detected, but not recovered* – The microprocessor raised an exception that caused the processor to enter debug mode. This may happen in experiments using the Nexus debugger, *i.e.*, the test port-based and instrumentation-based injections, as there are some exceptions that cannot be recovered due to limitations in the experimental setup.
- *Crash* – The brake controller stopped producing outputs.

We observe that all errors injected into memory caused identical results to be produced, but some errors injected into registers by the three techniques caused different outcomes. The results obtained using the exception-based and the test port-based techniques were identical for 933 out of 1 000 of the register errors. The exception-based and instrumentation-

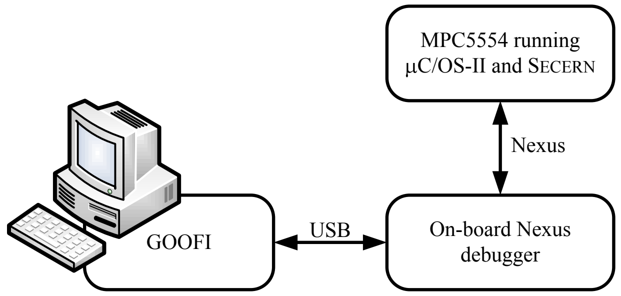

Figure 3. Evaluation platform for μC/OS-II and SECERN.

Table IV
ACTIVATION OF THE INJECTION BREAKPOINT FOR THE PARTITIONING KERNEL EXPERIMENTS.

Breakpoint activation	Test port-based	Instrumentation-based
Register errors		
Number of experiments	1000	1000
Breakpoint reached	499	499
Breakpoint not reached	501	501
Memory errors		
Number of experiments	1000	1000
Breakpoint reached	520	520
Breakpoint not reached	480	480

based techniques caused identical results for 921 out of 1 000 errors injected into registers. For the instrumentation-based and test port-based technique, 986 out of 1 000 errors caused the exact same results to be produced.

We investigated the errors that differed in their classification to identify which uncertainties that affected the results. For the instrumentation-based and test port-based technique, we observed that the differences in the results were caused by initialization uncertainty. This source of uncertainty also affected the results obtained by the exception-based technique. The spatial intrusiveness of this technique, however, caused additional differences compared to the results obtained by the test port-based and instrumentation-based techniques.

The spatial intrusiveness of the exception-based technique was caused by the instructions added to the brake controller program, and by the memory usage during the execution of those instructions. We observed that the stack usage of the fault injection console caused parts of the data memory to be initialized when the brake controller program started to execute. These memory values were non-initialized in the experiments using the test port-based and instrumentation-based techniques.

B. Experiments targeting the partitioning kernel

This section presents the results of the experimental evaluation of uncertainty targeting the μC/OS-II real-time kernel [22] with the SECERN partitioning extension. The extension provides robust partitioning among different tasks running on the kernel in order to ensure that a faulty task is unable to affect the execution of either the kernel or other tasks. In previous work, we have used fault injection to assess the correctness of the partitioning mechanisms [23]. Through those experiments it was possible to find several vulnerabilities in the extension and correct them. In this paper we target the updated version of SECERN, and investigate whether different fault injection techniques can obtain compatible measurements.

1) Target system and experimental setup: The extended version of the kernel runs on a computer board featuring a Freescale MPC5554 microprocessor [24], based on the PowerPC architecture. The processor core includes a memory management unit (MMU) which provides, among other services, memory protection. In addition to memory protection, the extension provides support for system call checks, stack pointer checking at context switches, application-specific checks and general processor exceptions to detect errors.

The experimental setup consists of a workstation, with GOOFI-2 and the winIDEA development environment, controlling an MPC5554 development board [25]. The development board includes an on-board Nexus debugger. Figure 3 depicts the experimental platform.

The base version of μC/OS-II that we used lacks support for isolating applications from one another and from the operating system. One of the key modifications to μC/OS-II is the distinction between processes and threads, where each process owns a private address space that groups together one or more execution threads. Each process acts as a container which is sometimes called a *partition* or a *sandbox*.

When any of the error detection mechanisms is triggered, SECERN handles the error in one of two central exception handlers. The *recoverable* exception handler is activated when the detected error is confined to a single process (*i.e.*, partition) and it is possible to delete that process and continue executing. All errors that can safely be assumed to be recoverable are forwarded to this exception handler. The remaining errors are forwarded to the *unrecoverable* exception handler. The currently implemented version enters an infinite loop in such cases.

2) Experimental results: We configured the system to execute two processes (or tasks), each one with a single thread. The two threads executed, in an infinite loop, a data processing routine and released the CPU until the next iteration. Both threads executed a workload consisting of a wavelet transform, which takes an array of input data an produces an output array containing the result of the transform.

We executed a set of experiments targeting the task with the lowest priority (task 2). The code addresses to set the fault injection breakpoint were selected to cover the entire code segment of task 2, and faults were selected randomly targeting general purpose registers and memory locations (the data and stack segments of task 2).

We injected 1000 faults into registers and 1000 faults

Table V
SUMMARY OF KERNEL AND TASK OUTCOMES FOR THE PARTITIONING KERNEL EXPERIMENTS.

Outcomes	Test port-based	Instrumentation-based
Register errors		
Kernel crash	0 (0%)	0 (0%)
Task 1: correct output	499 (100%)	499 (100%)
Task 1: wrong output	0 (0%)	0 (0%)
Task 1: removed	0 (0%)	0 (0%)
Task 2: correct output	444 (89.0%)	443 (88.8%)
Task 2: wrong output	10 (2%)	10 (2%)
Task 2: removed	45 (9.0%)	46 (9.2%)
Memory errors		
Kernel crash	0 (0%)	0 (0%)
Task 1: correct output	520 (100%)	520 (100%)
Task 1: wrong output	0 (0%)	0 (0%)
Task 1: removed	0 (0%)	0 (0%)
Task 2: correct output	448 (86.2%)	448 (86.2%)
Task 2: wrong output	68 (13.1%)	68 (13.1%)
Task 2: removed	4 (0.8%)	4 (0.8%)

into memory locations, using the test port-based and the instrumentation-based techniques; there is currently no support for exception-based injection in the MPC5554, since the processor board provides no communication interfaces and the exception-based injection tool requires at least a serial port to download code and data.

To speed-up the fault injection process, the partitioning kernel setup does not collect the sequence of instructions executed in the fault-free experiment (*i.e.*, it does not create a program trace) and, unlike the MPC565 setup, the pre-injection analysis is not performed. To set a breakpoint, we choose a random code address from the entire range of user-defined addresses. Since that address might not be reached once the program executes, there is a number of experiments in which a fault is never injected and the outcome is exactly the same as that of a fault-free experiment. Such experiments are discarded during analysis and classification.

Table IV shows that the fault injection breakpoint was reached, in this set of experiments, in about half of the executions. In the remaining experiments the fault injection breakpoint was not reached and therefore no fault was injected. There is a complete match between the two fault injection techniques. This means that the execution of the workloads until the fault is injected has no observable uncertainty.

We analyzed the experiments where a bit-flip was actually injected – 499 register errors and 520 memory errors – to determine whether it was correctly handled by the kernel. The classification process takes into account the activation of the centralized exception handlers (recoverable and unrecoverable) and the output of the tasks to determine whether or not the fault was handled.

Experiments were classified according to the outcome of each task (correct output, wrong output, and removal) and the outcome of the kernel (operational or crashed):

- *Kernel crash* – The unrecoverable exception handler is executed or the experiment reaches a timeout at an instruction which is not set as a valid end point or breakpoint.
- *Correct output* – The task continues executing and there is no difference between its output after the fault was injected and its output in the reference experiment.
- *Wrong output* – The task produces results differing from the reference output at some point of its execution.
- *Removed* – One of the error detection mechanisms is triggered, the recoverable exception handler is executed and the task is removed.

Table V shows the classification of the fault injection experiments for the test port-based injection and instrumentation-based injection techniques. We can observe that all errors were correctly handled by the kernel, since there were no kernel crashes and task 1 (the fault-free task) always produced correct results. In our previous paper, the kernel was tested under a similar faultload and the software defects that were exposed at that time were removed. It is therefore unsurprising that this faultload is unable to expose more defects that might still exist. The uncertainty with this respect is to have a representative sampling with respect to faultload as well as workload.

Nevertheless, the outcomes of task 2 provide some insight into whether the same measurement can be made by different techniques. There is nearly a complete match between the results of test port-based and instrumentation-based injections. For register errors, there was only a single experiment differing in classification, in which a correct output was obtained through test port-based and the instrumentation-based technique classified it as a task removal. The error was injected into a resource which was apparently unused, so the recoverable exception handler must have been triggered by some other problem which we were unable to reproduce. We repeated that same experiment several times, with both injection techniques, and the outcome was always classified as correct output.

The different outcome for the experiment classified as task removal is attributed to assumptions in the measurement procedure, which is a source of uncertainty described in Section III. We have observed some very rare cases when the programming of the flash, which is used to store the target program, fails without giving any warning or error message. As the flash is programmed for each experiment using the instrumentation-based technique, this uncertainty can explain the different outcome for the experiment.

Regarding the memory experiments, there is a complete match between the techniques. In this case, the trend is similar to the brake-by-wire experiments: the tendency is that memory errors cause mostly data errors, manifested as wrong output. For register errors, the trend is also similar in that most of the errors that have an impact on the system are detected by the error detection mechanisms.

However, the outcomes of register errors are nearly identical for both injection techniques, whereas in the brake-by-wire experiments we observed some differences. The main reason is that the partitioning kernel initializes all data segments with zeros, the unused portions of memory are not mapped in the MMU and the read-only segments cannot be written to due to the memory permissions set in the MMU. Consequently, there is very little to no initialization uncertainty in the partitioning kernel experiments. Any attempt by an instruction to access an unused (and therefore uninitialized) section of memory results in a recoverable exception generated by the MMU, *i.e.*, these errors are detected and the faulty task is deleted.

V. Discussion of results

In the experiments targeting the brake-by-wire controller, we observed that the exception-based and test port-based technique caused identical program outcomes and error detections to be produced for 93.3% of the injected errors into registers. For the exception-based and instrumentation-based technique, the results were identical for 92.1% of the injected register errors. When an error is injected into a register used for calculating the effective address of a load or a store instruction, the system will make operations on different memory locations. Thus, since those locations might be uninitialized on the brake-by-wire controller, there is a great deal of initialization uncertainty in the register error experiments of the brake-by-wire controller.

In the case of memory errors affecting the brake-by-wire controller, we observed identical outcomes for all experiments. The main tendency of memory errors is to cause pure value errors. When a corrupted value is read from memory, the target system tends to use that value in calculations that have systematic effects on the workload, independent of the technique used for the fault injection.

Initialization uncertainty is not present in the partitioning kernel experiments, since the kernel initializes all used memory segments at start-up and the MMU prevents tasks from accessing any other areas in memory. Thus, as we can see in Table V, the match between experiments using test port-based injection and instrumentation-based injection is nearly perfect.

The evidence gathered in the brake-by-wire experiments suggests that initialization uncertainty is a dominant factor in the variability of measurement results. It has a random effect, so it contributes to widening the confidence interval of a given set of measurements. Spatial intrusiveness is another significant source of uncertainty. The code added to the brake-by-wire program for the experiments using the exception-based technique caused certain memory values to become initialized, instead of being uninitialized. These values led to different outcomes for some experiments using exception-based injection, compared to the outcomes produced by the two other techniques.

One particularly important result of the experiments described in this paper is the validation of all techniques. Even though some estimations differ, the majority of individual experiments resulted in identical behaviour across the three techniques. We examined the few cases in which the outcomes differed, and concluded that these differences were not caused by bugs in the tool, but rather by measurement uncertainties. We can therefore be confident that the instrumental uncertainty is low.

VI. Conclusion

We identified and evaluated sources of uncertainty affecting measurements taken with three fault injection techniques: test port-based injection, exception-based injection, and instrumentation-based injection. In most cases, the outcome of injecting a given error is identical for the three techniques, increasing our confidence in that the intended errors are correctly injected, analyzed, and classified. The cases in which the outcomes differed were caused by measurement uncertainties.

The evidence collected in our experiments, targeting a brake-by-wire controller and a partitioning kernel, suggests that initialization uncertainty can have a significant impact on measurement results. The brake-by-wire controller only initializes specific memory variables and is consequently influenced by uninitialized values. In contrast, the partitioning kernel is not affected by this factor, given that it initializes the accessible portions of memory. Spatial intrusiveness was also identified as a significant source of uncertainty. The measurement results obtained using exception-based injection were the most affected by this uncertainty.

These uncertainties caused differences in the outcome of many individual experiments. Nevertheless, the final results produced from a large number of experiments are similar for the three techniques. We can therefore conclude that the three techniques are able to obtain metrologically compatible results.

References

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

[2] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi, "Foundations of measurement theory applied to the evaluation of dependability attributes," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, Jun. 2007, pp. 522–533.

[3] Joint Committee for Guides in Metrology (JCGM), "International vocabulary of metrology – Basic and general concepts and associated terms (VIM)," 3rd edition, 2008.

[4] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[5] J. A. Clark and D. K. Pradhan, "Fault injection: A method for validating computer-system dependability," *IEEE Computer*, vol. 28, no. 6, pp. 47–56, Jun. 1995.

[6] P. Yuste, J. Ruiz, L. Lemus, and P. Gil, "Non-intrusive software-implemented fault injection in embedded systems," in *Dependable Computing: Proceedings of the First Latin-American Symposium (LADC 2003)*, ser. Lecture Notes in Computer Science, vol. 2847, 2003, pp. 23–38.

[7] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, Feb. 1998.

[8] J. C. Cunha, M. Z. Rela, and J. G. Silva, "Can software implemented fault-injection be used on real-time systems?" in *Dependable Computing - EDCC-3, Proceedings of the Third European Dependable Computing Conference*, ser. Lecture Notes in Computer Science, vol. 1667, 1999, pp. 209–226.

[9] J. H. Barton, E. W. Czeck, Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, Apr. 1990.

[10] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, Feb. 1995.

[11] E. Fuchs, "An evaluation of the error detection mechanisms in MARS using software-implemented fault injection," in *Dependable Computing - EDCC-2, Proceedings of the Second European Dependable Computing Conference*, ser. Lecture Notes in Computer Science, vol. 1150, 1996, pp. 73–90.

[12] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi, "Towards making NekoStat a proper measurement tool for the validation of distributed systems," in *Proceedings of the Eighth International Symposium on International Symposium on Autonomous Decentralized Systems (ISADS'07)*, Mar. 2007, pp. 377–386.

[13] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, Jun./Jul. 2010, to appear.

[14] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[15] Joint Committee for Guides in Metrology (JCGM), "Evaluation of measurement data – Guide to the expression of uncertainty in measurement," First edition, 2008.

[16] K. Kanoun and L. Spainhower, Eds., *Dependability Benchmarking for Computer Systems*. Wiley, 2008.

[17] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Transactions on programming languages and systems*, vol. 30, no. 3, pp. 12–41, May 2008.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC 2001)*, Dec. 2001, pp. 3–14.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, Jun. 2005, pp. 190–200.

[20] D. Skarin and J. Karlsson, "Software mechanisms for tolerating soft errors in an automotive brake-controller," in *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009), Supplemental Volume*, Jun. 2009, pp. D34–D38.

[21] J. Cunha, A. Correia, J. Henriques, M. Rela, and J. Silva, "Reset-driven fault tolerance," in *Dependable Computing EDCC-4: Proceedings of the 4th European Dependable Computing Conference*, ser. Lecture Notes in Computer Science, vol. 2485, 2002, pp. 102–120.

[22] J. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*, 2nd ed. CMP Books, 2002.

[23] R. Barbosa and J. Karlsson, "Experiences from verifying a partitioning kernel using fault injection," in *Proceedings of the 12th European Workshop on Dependable Computing (EWDC 2009)*, May 2009.

[24] Freescale Semiconductor, Inc., *MPC5553/MPC5554 Microcontroller Reference Manual (Rev 4.0)*, Apr. 2007.

[25] iSYSTEM AG, *EVB-5554 Evaluation and Development Kit for Freescale PowerPC MPC5554 Microcontroller (User's Manual)*, Jul. 2007.