DBench

Dependability Benchmarking

IST-2000-25425

Fault Representativeness

Report Version: Deliverable ETIE2

Report Preparation Date: June 2002

Classification: Public Circulation

Contract Start Date: 1 January 2001

Duration: 36m

Project Co-ordinator: LAAS-CNRS (France)

Partners: Chalmers University of Technology (Sweden), Critical Software (Portugal), University of Coimbra (Portugal), Friedrich Alexander University, Erlangen-Nürnberg (Germany), LAAS-CNRS (France), Polytechnical University of Valencia (Spain).

Sponsor: Microsoft (UK)

Project funded by the European Community under the "Information Society Technologies" Programme (1998-2002)

Contents

Fault Representativeness

Authored by: Pedro Gil[♦♦], Jean Arlat[*], Henrique Madeira[++], Yves Crouzet[*], Tahar Jarboui[*], Karama Kanoun[*], Thomas Marteau[*], João Durães[++], Marco Vieira[++], Daniel Gil[♦♦], Juan-Carlos Baraza[♦♦], and Joaquín Gracia[♦♦]

[*] **LAAS** [++] **FCTUC** [♦♦] **UPVLC**

June 21, 2002

Abstract

The determination of a *representative faultload* has been identified as one of the key challenges for the DBench project. Besides it has long being recognized as a pragmatic way to assess the behaviour of computer systems in presence of faults, the application of fault injection in the context of dependability benchmarking is far from being straightforward.

Indeed, *fault representativeness* is a very decisive factor in relation with the necessary properties of a dependability benchmark — especially, *fairness*, i.e., forming a consistent reference for assessing alternative solutions. The fulfilment of this property has a strong impact on the selection of the faultload, and especially with respect to its representativeness.

In order to tackle this problem, we have first sketched a conceptual framework to precisely identify what were the main notions that are governing the problem of fault representativeness, especially in the context of a fault injection experiment. Besides its usefulness to describe these relevant notions involved, this framework proved useful for structuring the set of specific experiments that we have devised to get better insights on the issue of fault representativeness. The deliverable then addresses, in turn, more specific issues relevant to the determination of representative faultloads with respect to hardware faults, software faults (distinguishing operating systems and application layers) and operation faults application (i.e., fault induced by operators). Finally, a brief discussion on the insights obtained and some recommendations for the derivation of representative faultloads conclude the deliverable.

1 Introduction

Among the various attributes (such as workload, faultload, measurements and measures) that have been introduced within the general framework defined by the DBench project [Madeira *et al.* 2001] to precisely characterize a dependability benchmark, the determination of a *representative faultload* has been identified as one of the key challenges.

Of course, among the suitable techniques available to generate such a faultload, fault injection appears as privileged approach. In particular, fault injection can be seen as a means for testing fault tolerance mechanisms with respects to special inputs they are meant to cope with: *the faults*. In addition, fault injection also proved very much useful for characterising the behaviour of computerised systems and components in presence of faults.

Numerous techniques for injection have been proposed [Carreira *et al.* 1999], ranging from i) simulation-based techniques at various levels of representation of the target system (physical, logical, RTL, PMS, etc.), ii) hardware techniques (e.g., pin-level injection, heavy-ion radiation, EMI, power supply alteration, etc.), and iii) software-implemented techniques that support the bit-flip model in memory elements. Many tools have been developed to facilitate the conduct of experiments based on these various techniques [Hsueh *et al.* 1997].

Building up on the advances made by these research efforts and on the actual benefits procured, fault injection progressively made its way within industry, where it is actually part of the development process of many manufacturers, integrators or stakeholders of dependable computer systems (e.g., Ansaldo Segnalamento Ferroviario, Astrium, Compaq/Tandem, ESA, SAAB Ericsson Space, Honeywell, IBM, Intel, NASA, Siemens, Sun, Volvo, just to name some). This confirms the pertinence and the usefulness of the approach.

Nevertheless, the application of fault injection in the context of dependability benchmarking is far from being straightforward. Indeed, *fault representativeness*, i.e., the plausibility of the supported fault model with respect to actual faults, is a concern that has often been related to fault injection-based experiments, even when they were targeted to the evaluation of a specific computer architecture. Such a concern, is even more acute in the case of dependability benchmarking, due to the fact that the ultimate objective is to compare alternative systems. Indeed, the faultload should comply with the main properties that need to be supported by dependability benchmarks.

More generally, along the same line as the necessary properties identified in Deliverable CF2 (see [Madeira *et al.* 2001]) and in addition to them, several important requirements have to be met so that dependability benchmarks can be actually recognized. These include: *agreement* among the dependability community, *acceptance* by the end-users at-large (providers, integrators, stakeholders, etc.), *usefulness*, by supporting the provision of meaningful measures, *fairness*, by forming a consistent reference for assessing alternative solutions. Agreement and acceptance are definitely the ultimate targets that need to be aimed at. Of course, usefulness and fairness are paramount in supporting these. We have identified early in the definition of the project, that the fulfilment of the fairness issue had a strong impact on the

selection of the faultload attribute, and especially with respect to representativeness property. Accordingly a specific effort has been devoted to tackle this problem.

Towards this goal, nevertheless, we advocate that the study of the impact and consequences of an injected fault (i.e., the error propagated) offers a pragmatic and sensible means to address the representativeness issue. We will further illustrate this argument in the sequel.

One important question is to figure out whether a focused set of fault injection techniques (ideally, a single one) could be identified as sufficient to generate a faultload for many classes of faults, or whether a distinct technique is needed for each class. Accordingly, one of the outcomes of this specific study is to provide some objective insights concerning the faultload dimension for dependability benchmarking whose detailed characterization will be carried out in Deliverable T23. The ultimate goal is to identify the technology that is both *necessary* and *sufficient* to generate the faultload to be included into a dependability benchmark.

This deliverable is organized as follows. Chapter 2 describes the context of representativeness concerning fault injection techniques and motivates our specific effort to address the issue of fault representativeness in the framework of DBench. It also identifies the various types of faults on which we have focused our attention: hardware, software and operation[1] faults. The following sections successively address each of these fault types. While hardware faults are dealt with in Chapter 3, software faults are considered in two separate sections by distinguishing the operating system point of view (Chapter 4) and the application point of view (Chapter 5). Chapter 6 covers the issues related to operation faults. Finally, Chapter 7 concludes this report by providing a synthesis of the results obtained and expressing some general recommendations.

[1] For sake of brevity, we use the term *operation* faults to designate those operational faults that are committed by users operating (with) the system during their interactions with the target system.

2 Context and Classes of Fault Addressed

This section is composed of three main parts. The first one motivates this specific effort by precisely identifying the problem posed by the generation of a *faultload* to be used in the context of dependability benchmarking in the light of the uncertainties concerning the types of faults actually covered by the available fault injection techniques. The second part provides a comprehensive framework that defines the main issues involved, and on which we have structured our focused effort. Finally, the last part briefly introduces the various studies presented in the subsequent sections.

2.1 From Real or Injected Faults to Dependability Benchmark Faultload

As is the case of performance benchmarks, it is expected that a dependability benchmark will heavily rely on experiments and measurements carried out on a target system.

Analogously to what was introduced for fault injection-based evaluation [Arlat *et al.* 1990], the experimental dimension of a dependability benchmark can be characterized by an *input* domain and an *output* domain. The input domain corresponds to activity of the target system (*workload* set) and the set of injected faults (*faultload* set). In particular, the *workload* defines the activation profile for the injected faults. The output domain corresponds to a set of observations (*measurements* set) that are collected to characterize the target system behaviour in the presence of faults. A set of specific or comprehensive *dependability measures* can then be derived from the processing of the *measurements* set, as well as the exploitation of the data available concerning the *workload* and *faultload* sets.

It is definitely the determination of the *faultload* set — the very novel part of the input domain with respect to performance benchmarking — that poses the most significant problem in order to achieve a fair assessment of the dependability measures of interest. At first glance, it would seem necessary to confidently match the techniques being considered to generate the faultload with respect to the real faults that are targeted. Nevertheless, having to support only a limited number of techniques would be definitely beneficial from the usability point of view. Moreover, it is worth pointing out that what is important is not to establish an equivalence in the fault domain, but rather in the error domain, as similar errors can be induced by different types of faults.

The concepts and terminology used to account for threats and describe fault pathology are in accordance with wide spread usage in the dependability community (e.g., see [Avizienis *et al.* 2001]). A system may *fail* either because it does not comply with the specification, or because the specification did not adequately describe its function. An *error* is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service. A *fault* is the adjudged or hypothesized cause of an error. A fault is *active* when it produces an error; otherwise it is *dormant*.

Accordingly, two main related issues and questions have to be considered:

1) Fault representativeness (of a fault injection technique): To what extent the errors induced are similar to those provoked by real faults or by a representative fault model?

2) Fault equivalence (of fault injection techniques): To what extent distinct fault injection techniques do lead to similar consequences (errors and failures)?

So far, the investigations carried out concerning the comparison of i) some specific fault injection technique with respect to real faults (e.g., see [Chillarege & Bowen 1989, Daran & Thévenod-Fosse 1996, Madeira *et al.* 2000]) and ii) several injection techniques (e.g., see [Choi & Iyer 1992, Yount & Siewiorek 1996, Folkesson *et al.* 1998, Fuchs 1998, Stott *et al.* 1998]) have shown mixed results. Some were found to be quite equivalent, while others were identified as rather complementary.

In particular, in [Daran & Thévenod-Fosse 1996], it was found that about 80% of the mutations considered led to errors similar to real software faults. On the other hand, the study reported in [Fuchs 1998] revealed that: i) the compile-time form of the software-implemented fault injection technique used to inject faults in the *code segment* of the executed application was found to sensitize the error detection mechanisms included into the MARS target system in a way similar to the physical fault injection techniques considered (pin-forcing, heavy-ion radiation, EMI), ii) faults injected into the *data segment* led to significantly distinct behaviours.

Thus, the variability of these results motivated us to carry out a comprehensive set of co-ordinated experiments to provide a good appraisal of the questions raised previously about the faults actually covered by the various fault injection techniques available.

Initially, most studies related to the application of fault injection on a prototype of a fault-tolerant system or a real target system relied on physical fault injection, i.e., the introduction of faults through the hardware layer of the target system (e.g., see [Arlat 1992]). Physical techniques actually enable real faults to be injected in a very close representation of the target system especially without any alteration to the software being executed. Still, such an approach is nowadays impractical for many computers systems due to the speed and reachability constraints that characterise modern processors and integrated circuit components. Nevertheless, in spite of the difficulties in both developing support environments and conducting such experiments, the application of physical level techniques can still be envisaged for embedded systems based on small-scale controller devices. In particular, this application can be facilitated by taking advantage of the testability-support capabilities (in particular those based on the IEEE Std 1149.1. standard for boundary scan structures and methods) featured by most moderns ICs (e.g., see the Scan Chain-Implemented fault injection technique supported by the GOOFI tool [Aidemark *et al.* 2001]). Nevertheless, such a technique suffers from a non-negligible intrusiveness, in particular in the time domain.

A trend favouring the injection of perturbations through the software layer for simulating physical faults (i.e., software-implemented fault injection — SWIFI for short) has emerged in the last decade (e.g., see [Barton *et al.* 1990, Kanawati *et al.* 1995, Carreira *et al.* 1998, Stott *et al.* 1998, Arlat *et al.* 2002]). Such an approach facilitates the application of fault injection by overcoming several problems associated with physical techniques (such as controllability,

repeatability, etc.). Moreover, recent studies have shown that SWIFI was also able to emulate to some extent software faults (e.g., see [Madeira *et al.* 2000]). Accordingly, from a pragmatic viewpoint, SWIFI would seem a privileged technique for generating the faultload in the context of dependability benchmarking. Nevertheless, more experimental work and related analyses are needed to get more evidence on the faults actually covered by SWIFI and more generally to understand the underlying error creation and propagation mechanisms.

To cope with the temporal intrusiveness problem mentioned earlier, especially when the technique depicted in [Rodríguez *et al.* 2002]), an attractive alternative is to take advantage of the standard debugging technology that is now available for supporting embedded processors designed to run real-time programs. The successful application of such an approach was reported in [Krishnamurthy *et al.* 1998]. Among the debugging technology now available, the Nexus[2] embedded processor debug interface is an open industry standard that provides interesting characteristics that can be useful for fault injection as well. For example, it is possible to access memory "on-the-fly". This special function was primarily meant for debugging hard real-time systems (without stopping nor affecting the system under test). It can readily be used for injecting faults using a special, non-intrusive, SWIFI technique, as it allows reading and writing the memory while the processor is running, without any significant overhead.

2.2 Fault Pathology, Fault Representativeness and Fault Equivalence

In this subsection, we provide a global framework where the main issues related to the assessment of the relationship between real (reference) faults and injected faults in a target system can be explicitly identified. The various items addressed concern: the target system levels, fault injection level, distance from fault injection level to reference faults level as well as to the levels where the faulty behaviour is observed. The impact of the fault tolerance mechanisms on the fault pathology and the necessary trade-offs between representativeness/equivalence property and ease of application (portability, etc.) of the technology used for faultload generation are then briefly discussed. A preliminary (shorter) version of this discussion appears in [Arlat & Crouzet 2002].

2.2.1 Target System Levels

Several relevant (ordered) levels of a computer system can be identified (e.g., physical-device, logic, RTL, algorithmic, kernel, middleware, application, operation). At each of these various levels faults may occur and consequences (errors, failures) may be observed. To make things clear, in such context, operation faults correspond to fault committed by users during their interactions with the target system.

Concerning faults, these levels may correspond to levels where real faults are considered and (artificial) faults can be injected. Concerning errors, the fault tolerance mechanisms (especially, the error detection mechanisms) provide convenient built-in monitors.

[2] Also known as IEEE-ISTO 5001™-1999 (See http://www.ieee-isto.org/Nexus5001).

2.2.2 Fault Injection Level

For characterising the behaviour of a computer system in presence of faults, it is not necessary *a priori* that the injected faults be "close" to the target faults (reference), it is sufficient that they induce similar behaviours.

Indeed, similar errors can be induced by different types of faults (e.g., a bit-flip in a register or memory cell can be provoked by an heavy-ion or as the result of an error provoked by a software fault).

What matters is not to establish an equivalence in the fault domain, but rather in the error domain (Figure 2.1). As shown in the figure, the RTL level — on which the SWIFI technique focuses — is a privileged level that covers the consequence of faults originating from a wide range of levels.

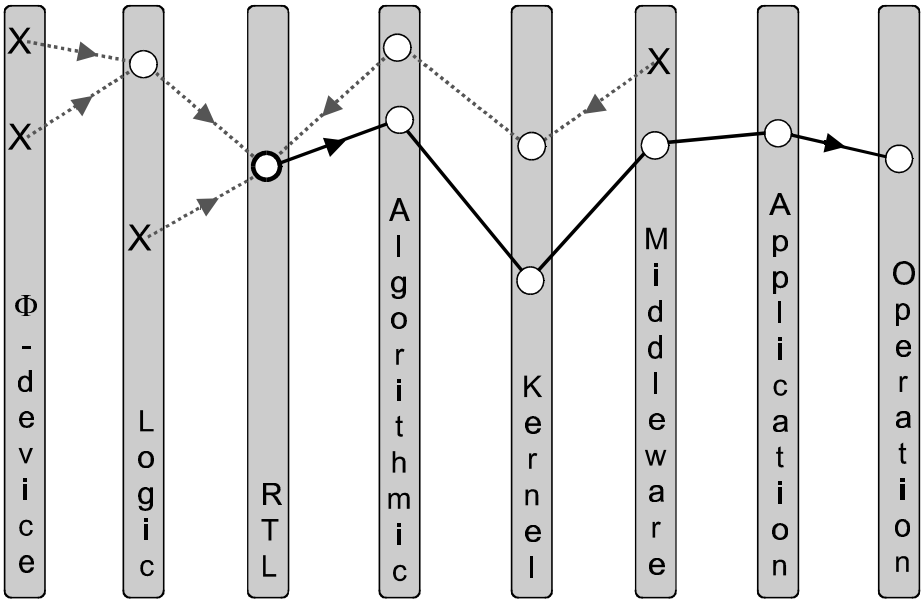


Figure 2.1: Target system levels and fault pathology

2.2.3 Distances of Injection from Reference and Observation

What matters is that the respective error propagation paths *converge* before the level where the behaviours are observed. This is depicted on Figure 2.2. Two important parameters can be defined on these various levels:

- distance d_r, that separates the level where faults are injected from the reference fault level(s);
- distance d_o, that separates the level where the faults are injected from the levels their effects are observed to be compared.

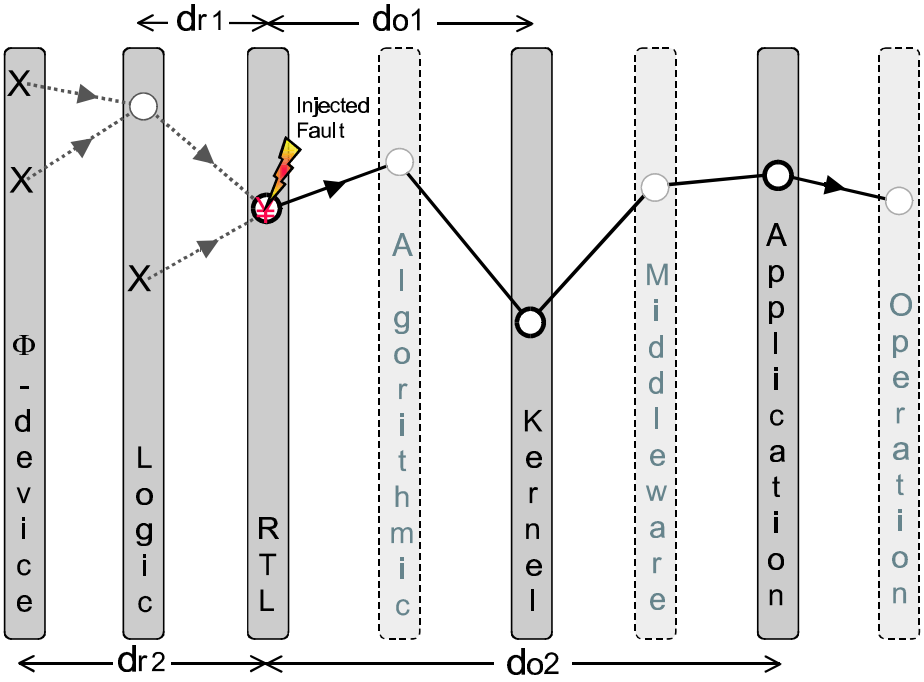


Figure 2.2: Reference fault and observation distances

The shorter ***dr*** and the longer ***do***, the more it is likely that the injected faults will exhibit behaviours similar to those provoked by the targeted reference faults.

The first claim directly results from the fact that a zero distance characterises the (limiting) case when injected faults are the same as reference faults. The rationale for the second one is based on the fact that the right-most levels correspond to higher abstractions (interpretations) of system states; accordingly it is more likely that the observations of erroneous behaviours that are associated to such levels will be perceived as equivalent outcomes, in particular failure modes (e.g., hangs, erroneous results, etc.).

2.2.4 Fault Tolerance Mechanisms

In practice, it may be the case that the presence of a specific fault tolerance mechanism (FTM) on one target system (and not on the other one(s)) will alter the error propagation paths.

This has a significant impact on the scope of (real) faults actually covered by the injected faults, whenever the FTM is implemented at level located between the level of the targeted faults and the level where the faults are injected and thus intercept the error propagation paths.

Indeed, assuming a perfect (100%) coverage for the FTM, then representativeness (with respect to the targeted faults) of the benchmark using the faultload characterized by the injected faults would then be zero. This could be simply accounted for by introducing another distance parameter: the *distance* ***dm*** separating the level where the faults are injected from the level where the fault tolerance ***mechanism*** is acting.

Such a knowledge about a FTM and its location and the related distance ***dm*** is important to select the appropriate level where faults are to be injected to emulate faults located at a given reference level. To illustrate the problem, if one considers a system where memory is being

protected by EDAC mechanisms, then the application of bit-flips via the SWIFI technique, where faults are injected at the RTL level by software, would not allow to readily emulate the same behaviours as those induced by real SEUs affecting the memory circuits. In this case, this would mean that the faults would need to be injected rather at the physical level.

It is thus clear that the claim made earlier that ***dr*** should be short is also related to the impact of FTMs: indeed, the shorter ***dr*** the less it is likely that a FTM is located between the reference and injection levels.

Additional concerns with respect to the knowledge about the presence of FTMs can be identified that are related to: i) selecting the most appropriate faultload (for example, when the objective is to test a FTM, then, whenever possible, a short distance ***dm*** should be preferred), and ii) getting some insights whether the interpretation of results of a benchmark can be extrapolated confidently beyond the faultload that has been actually applied.

2.2.5 Necessary Trade-Offs

From a dependability benchmarking point of view, it might not always be possible or cost-effective to have access to the actual structure of the target system to identify *a priori* a faultload complying with the representativeness property.

Accordingly, an alternative could be to favour a standard fault injection technique that is not perfect (from the representativeness point of view: e.g., errors induced cannot be confidently linked to a specific set of real faults), but that is nevertheless useful, as it covers a large spectrum of faults and is easy to implement[3].

Such trade-offs will be investigated in WP3, where benchmark prototypes will be proposed and evaluated.

2.3 The Faults Addressed and the Conducted Experiments

The experiments being conducted in the framework of the specific effort devoted to representativeness and equivalence of fault injection techniques have targeted faults concerning various of the levels previously identified for a target system: hardware, software and operation.

Hardware faults encompass the physical device, logic and RTL levels. Hardware faults are actually an essential dimension of our dependability benchmarking effort, especially for what concerns embedded systems that are considered as one of the application domains for which prototype benchmarks will be studied within DBench. Specific analyses of error propagation were carried out by means of fault injection experiments using VHDL-based simulation.

For software we have explicitly distinguished the case of the operating system (kernel and middleware levels) and the application level.

[3] In such a case, there might exist a need to establish a dialogue with the provider of the system being benchmarked in order to derive a fair interpretation or post processing of the benchmark measurements.

For what concerns operating system point of view, in this study, we have focused our effort on Linux, mainly because its Open Source statutes significantly facilitates the supporting of the internal controllability and observability features needed by the experiments to be conducted. Linux has now become a "real world" operating system that is being used in a wide range of applications including those with dependability requirements. Windows 2000 will be the other generic OS being considered by the project for developing prototype benchmarks targeting OSs.

Concerning the application point of view, the representativeness experiments have targeted transactional applications running on Windows 2000 and Oracle database. Indeed, this is one of the application domains that has been selected to develop prototype benchmarks. Linux will be also considered in future work.

Operation faults, i.e., faults committed by users during their interactions with the transactional system, reportedly correspond to a large proportion of faults affecting real use of data base systems. Accordingly, we have carried a specific study to address this issue.

In each case, a comprehensive work combining analysis and experimentation (involving several fault injection techniques) has been carried out. Analysis was in particular related to the consideration of real faults.

This analysis was meant for the derivation of fault/error models to support the conduct of fault injection experiments, by proposing either some specific classes of faults to be readily injected (i.e., with zero reference distance[4]), or some specific "watch" mechanisms to monitor the fault consequences from the typical error patterns caused by real faults as was used in the case of the OS experiments.

[4] It is worth noting that this was especially the case for the experiments concerning operation faults where scripts simulating the occurrence of real faults (derived from interviews and log data analyses) were essentially used.

3 Hardware Fault Representativeness

As was indicated in Section 2.3, hardware faults encompass the Physical Device, Logic and RTL levels. Nowadays, hardware faults are an essential dimension of our dependability benchmarking effort, especially for what concerns embedded systems that are considered as one of the application domains for which prototype benchmarks will be studied within DBench.

In order to study the fault representativeness in these levels, a *bottom-up* methodology based on a detailed understanding of the root-cause of faults must be applied. Fault models are then deduced from the physical causes and mechanisms implied in the occurrence of faults. This is related with *physics of failure methods* (also *called reliability physics methods*), that ideally would lead to a meaningful reliability prediction at the right abstraction level [Amerasekera & Najm 1997].

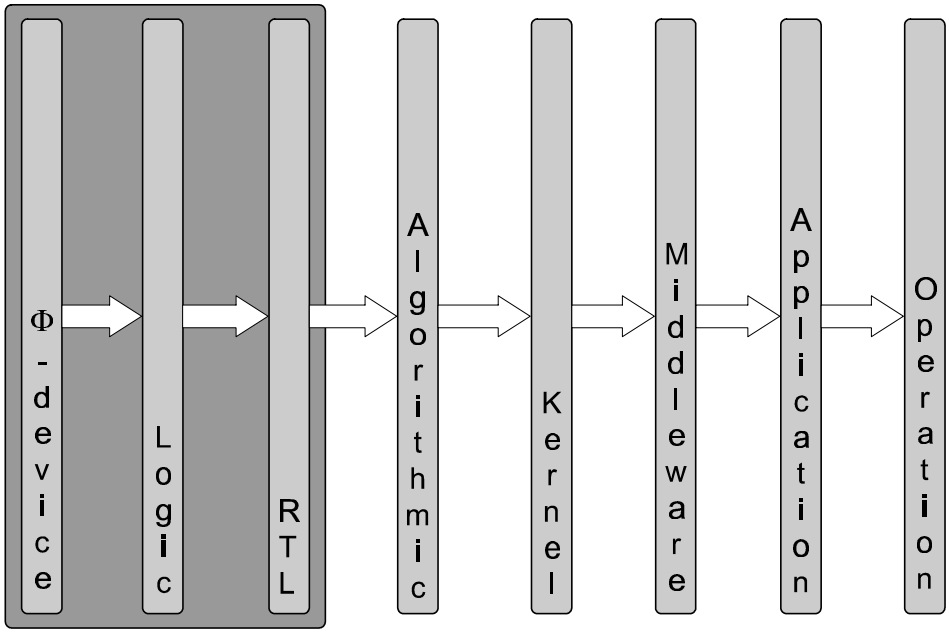


Figure 3.1: Hardware faults propagation at different abstraction levels in a system.

In this chapter, the connections between the Physical Device, Logic and RTL levels are established, as it is reflected in Figure 3.1. In Section 3.1, the connection between Physical Device, and Logic and RTL levels is done using a theoretical study of the physical mechanisms related to faults in modern ICs. From this study, a set of fault models for Logic and RTL levels are deduced. From the theoretical study we can conclude that, in future technologies faults in combinational logic will have a great importance.

To demonstrate the representativeness of the models deduced in Section 3.1, in Sections 3.2 and 3.3, a number of fault injection experiments using VHDL-based simulation are carried out. The aim of these experiments is to study the incidence of faults at RTL level. VHDL

[IEEE 1993] is a widely utilised standard for digital design, useful to make structural and behavioural descriptions. It has some elements that aid the fault injection process.

The experiments performed in Section 3.2 are oriented to verify the effect that injecting faults at Logic level produces at RTL level. The basic idea is to inject faults into combinational logic and study the fault manifestation in register elements. The results obtained show that their effects are very harmful in the registers of the system. However, as not all the registers of a system are identical (and, what is more important, not all the registers are accessible by software) a deeper study is carried out in Section 3.3. Here, the registers of a microprocessor are classified depending on their accessibility in *user* registers and *hidden* registers. When analysing the effects of injecting faults into both types of registers, we have found that the number of failures generated by injecting in *hidden* registers is not negligible.

3.1 Representativeness of Hardware Faults into Logic and RTL Levels

3.1.1 Fault Mechanisms and Models: Introduction

Although the most used fault models are stuck-at (0, 1) (for permanent faults) and bit-flip (for transient faults), as the integration density of VLSI circuits rises, it becomes more necessary to introduce newer, more complex models.

Depending on their duration, faults are classified in:

- Permanent, which remain in existence indefinitely
- Transient, with a usually short temporal duration
- Intermittent, similar to transient as they a have a temporal duration, but that appear and disappear repeatedly in time, without a periodical behaviour.

In next subsections we briefly describe the physical mechanisms implied in every type of fault, showing their corresponding fault models [Gil 1999].

3.1.1.1 Permanent Faults

Permanent faults are related to irreversible physical defects in the circuit. These defects can be produced during the manufacturing process or the normal operation (see Figure 3.2). In this case, a number of wearout mechanisms can occur in the long term, revealed initially as **intermittent faults** until they finally provoke a permanent fault.

Note that manufacturing faults can also produce subsequent faults during the circuit operation. This relationship has been represented in the figure with a dashed line.

In Figure 3.2 we can see the models deduced for permanent faults (inside the ovals), and the main related physical mechanisms (inside the rectangles). The figure shows some well-known causes (labelling the arcs) that originate these faults [Siewiorek 1994, Amerasekera & Najm 1997, Pradhan 1986]. Besides manufacturing defects (lattice, mask-layout, package, etc.), some causes tied to wearout (electrical stress, hot electronic trapping, thin-oxide breakdown, electromigration etc.) are shown. To deduce the fault models at Physical Device level, we have classified the mechanisms related to the causes into two groups.

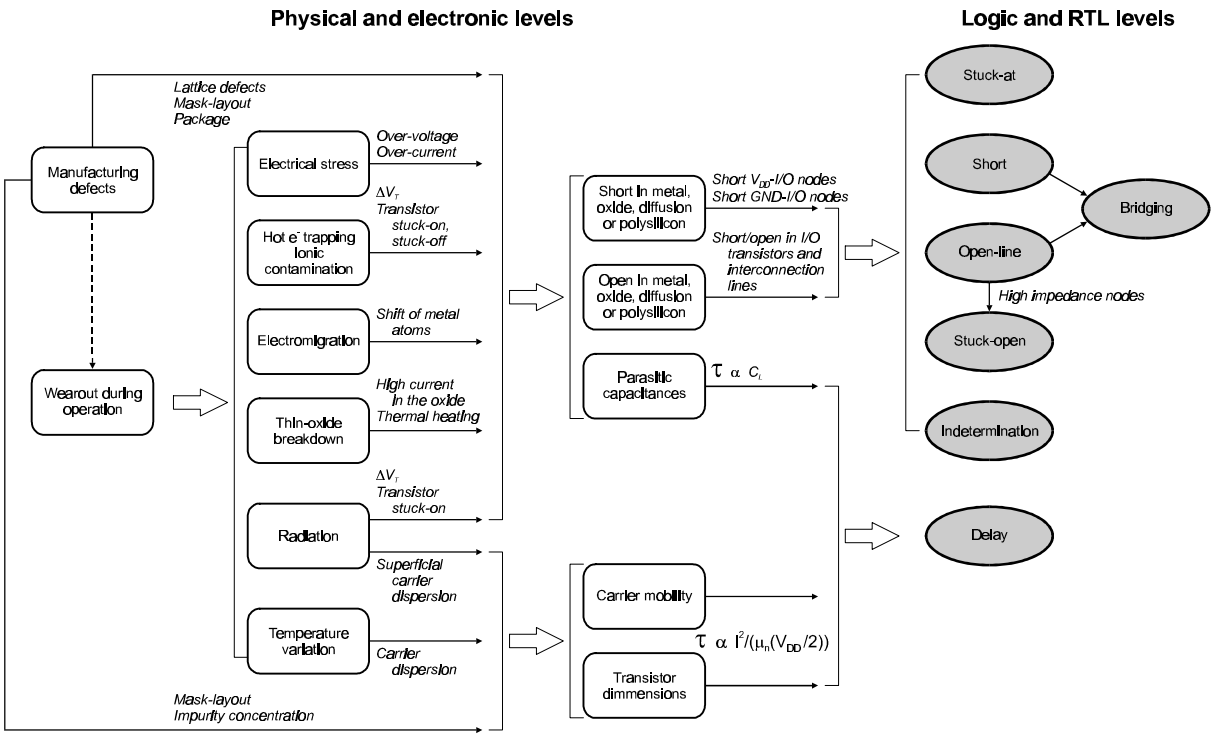


Figure 3.2: Some causes and mechanisms of permanent fault models.

In the first group we consider short/open faults in the transistor layers (metal, oxide, etc.). Depending on their effectiveness, these faults can manifest as transistor stuck-on/stuck-off or classic short/open. These faults in transistor layers can produce short faults between power lines (V_{DD}, GND) and the logic circuit I/O nodes. They can also cause short/open faults in both the I/O transistors and the connection lines between logic circuits. The effect of these faults at logic and RTL levels is a set of fault models sometimes related to each other:

- **Stuck-at (0, 1)**. This is the most used fault model
- **Short** in the connection lines of logic circuits
- **Open-line** in the connection lines of logic circuits
- **Bridging** (a combination of short and open-line)
- **Stuck-open**. This fault is due to floating high impedance nodes, which hold the previous logic value for the retention time, until the discharge of the output parasitic capacitances by leakage currents.
- **Indetermination**. In this case, the fault is due to either a short in the logic circuit outputs or an open in the inputs.

In the second group we include some faults which affect the switching delay of MOS transistors and the charge/discharge delay of parasitic capacitances in I/O connections: modification of parasitic capacitances at transistor level, carrier mobility, and transistor dimensions. Their effect at logic and RTL levels is a permanent modification of the logic circuit delays, so their fault model is **delay**.

3.1.1.2 Intermittent Faults

These faults are due to some of the mechanisms seen for permanent faults, particularly those caused by wearout. For that reason, the fault models applicable to intermittent faults are the same as for permanent faults.

3.1.1.3 Transient Faults

These faults, also called soft errors and single event upsets (SEUs) can appear during the operation of a circuit, due to different causes, either internal or external. In contrast to permanent faults, transient faults do not introduce a physical defect in the circuit. The treatment of these faults is difficult, which serves to emphasise their importance, because they cannot be located spatially and their duration is short. Moreover, they do not have a well-defined model, due to the variety of local and external phenomena that can originate them.

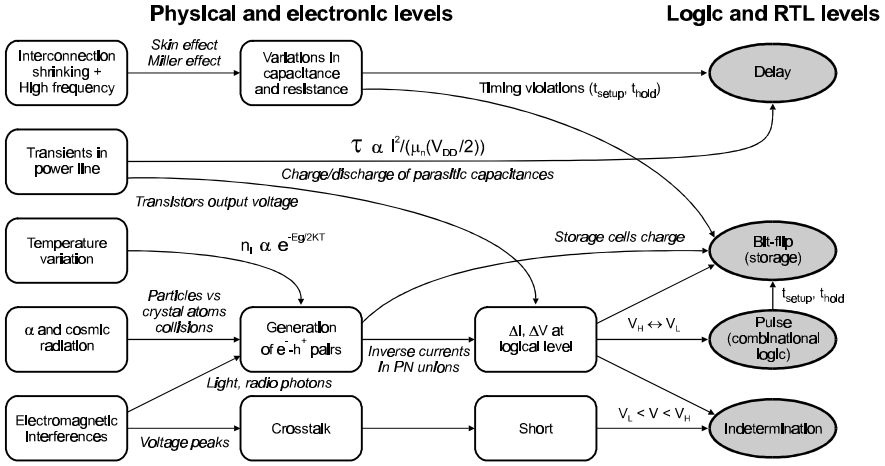


Figure 3.3: Some causes and mechanisms of transient fault models.

As Figure 3.3 shows, **bit-flip** (in storage), **pulse** (in combinational logic) and **indetermination** models allow to represent transient physical faults of different types: transient in power supply, crosstalk, electromagnetic interferences (light, radio, etc.), temperature variation, α and cosmic radiation (the last mechanism is very important in space and avionics applications). These physical faults can vary directly the values of voltage and current of the logical levels of circuit nodes, as for instance in power supply noise. They also can generate e^--h^+ pairs, which are swept by the electric field of the depletion zones of PN unions in transistors. This produces a current of e^--h^+ pairs, which may alter the charge in storage (SRAM, DRAM, and registers) cells [Amerasekera & Najm 1997] and flip their value (bit-flip). Electron-hole current can also change the logical levels in combinational circuit nodes, and therefore cause their indetermination or commutation, with the apparition of pulses (we can consider this faults as bit-flips, but in combinational logic). We have called **pulse** the model for this type of fault (produced in combinational logic) to differentiate from the bit-flip (produced in memory circuits). A pulse can in his turn affect signal timing (set-up, hold) of data to be written in memory circuits, generating storage cell bit-flips.

The **delay** model allows to represent physical faults due to transients in power supply (V_{DD}), which can alter the switching delay ($\tau \propto l^2/(\mu_n(V_{DD}/2))$) of MOS transistors, or the charge/discharge delay of parasitic capacitances in input/output connections.

Remark also a mechanism due to modern manufacturing features. The main factor is the interconnection shrinking when running at high frequencies. Under these circumstances, both skin and Miller effects are reinforced, provoking alterations in the RC time constant that can violate time margins (mainly $t_{set\text{-}up}$ and t_{hold}) leading to data corruption.

One of the most important concerns about transient faults is their duration, but very little has been written about this matter.

3.1.2 New Trends in Deep Submicron Technologies

Dependability of VLSI circuits is becoming more and more important as these devices are increasingly used in reliable and even safety-critical systems. Semiconductor technology advances have tremendously increased the performance of computing systems over the past decades. Although, shrinking geometries, lower power voltages and higher frequencies have a negative impact on dependability by increasing the rates of occurrence of transient, intermittent and permanent faults.

Next, we want to study the impact of the new technologies on physical fault mechanisms. Also, we will try to relate these mechanisms with fault models at Logic and RTL abstraction levels. Taken as the reference the mechanisms and models indicated in the previous section, our objective has been to point out the main issues and differences.

3.1.2.1 Permanent Faults

We will concentrate the study on some groups of fault mechanisms of high significance for new technologies: oxide damage, metallization and package and assembly. In every group, we will emphasise on some particular fault mechanisms. For every particular mechanism, we will explain the problem, describe its effect, and indicate the corresponding fault model(s) at Logic or RTL level.

3.1.2.1.1 Oxide Damage

Among the various fault mechanisms related to oxide damage, we will focus on oxide breakdown, hot carrier injection and plasma damage.

Oxide Breakdown

Damage to the gate oxide is one of the major concerns in MOS processes. As a consequence, maintaining *Gate Oxide Integrity* (GOI) is extremely important to process control. GOI is one of the trade-offs defining the gate oxide thickness in a new process, since thinner gate oxides are usually more sensitive to wearout and damage for a given supply voltage. Hence, GOI requirements have an important role in defining the maximum supply voltage at which circuits designed in a given technology can be operated.

Oxide thickness is shrinking as device geometry's scale down:

- In 1978, $t_{ox} \approx 750$ Å
- In 1988, $t_{ox} \approx 250$ Å
- In 1998, $t_{ox} \approx 80$ Å
- In 2002, $t_{ox} \approx$ 25-30 Å, which is not far from the physical limit of SiO_2 due to direct tunnelling phenomena.

The mechanisms related to gate oxide breakdown are complex and the issues involved are numerous [Hu & Lu 1999, Stathis 2001]. Basically, breakdown is a two-step process:

- Wearout: Defects or traps generated inside oxides and at interface due to current through oxides. The current is due to Fowler-Nordeim or direct tunnelling effects.
- Breakdown: Defects become high, leading to local high current densities followed by local thermal runaway.

Some approaches for modelling fault mechanisms are based on Percolation Theory of wearout and breakdown. Also, new models for Ultrathins have been introduced, as for example "V-model", where an electron must travel full thickness of oxide before slamming into opposite interface [Hawkins 2000].

The oxide wearout mechanism or Time-Dependent Dielectric Breakdown (TDDB) occurs at weaknesses in the oxide film due to defects, and it is towards the reduction of these defects that most effort in GOI is directed. Defects are usually due to the presence of impurities in the thermally grown oxide, or the location of broken SiO_2 bonds. Improvements in GOI have been shown through optimisation of the gate oxidation pre-clean, and the passivation process above the gate conductor [Strong *et al.* 1993].

Effects of Failure Mechanisms

Damage to the gate oxide can result in **excessive leakage** at the input and output pins, increased standby power dissipation in the IC and a **decrease in circuit speeds**. The current can cause damage both at the interface and in the bulk, thus generating more defects. Eventually the number of defects will be large enough so that the high current in the oxide can cause thermal heating and catastrophic damage. Effects of oxide breakdown are particularly of concern in memory cells where gate leakage could result in loss of stored data. In Electrically Erasable Programmable Read Only Memories (EEPROM) which rely on conduction through the gate oxide for operation, the oxides are subjected to high electric fields. These devices are more susceptible to weaker gate oxides.

Models at Logic Level

Besides **short/open-line** (see Figure 3.2) related to "strong" damage in transistors at breakdown step, it is necessary to introduce also some models for wearout step.

The increase of leakage current in the transistor gate can also increase the output current of the logic gates, as it is shown in Figure 3.4. This will affect the output voltages, leading

eventually to indeterminate logic voltages. So, we can represent this fault using the **indetermination** model at the output of the gates.

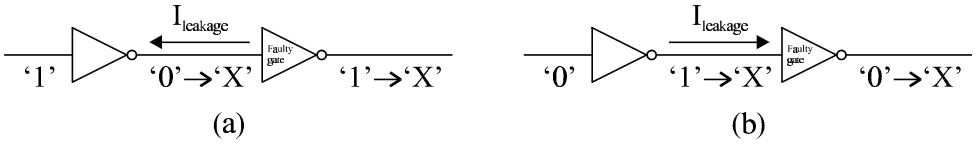


Figure 3.4: Indetermination fault caused by the increase of leakage current at the transistor gate. (a) Perturbation of '0' value. (b) Perturbation of '1' value.

The increase of leakage current can also affect the **switching speed of the gates**. Figure 3.5 shows **rise-time and fall-time** model for a CMOS inverter. The gate either charges or discharges a capacitive load C_L. In rise-time case, it can be assumed [Pucknell & Eshraghian 1994] that the PMOS transistor stays in saturation for the entire charging period of the load capacitor C_L. So, PMOS transistor is modelled as a current source. NMOS transistor is turned-off. Similar reasoning can be applied to the discharge of C_L through the NMOS transistor. This is in saturation and PMOS transistor is turned-off.

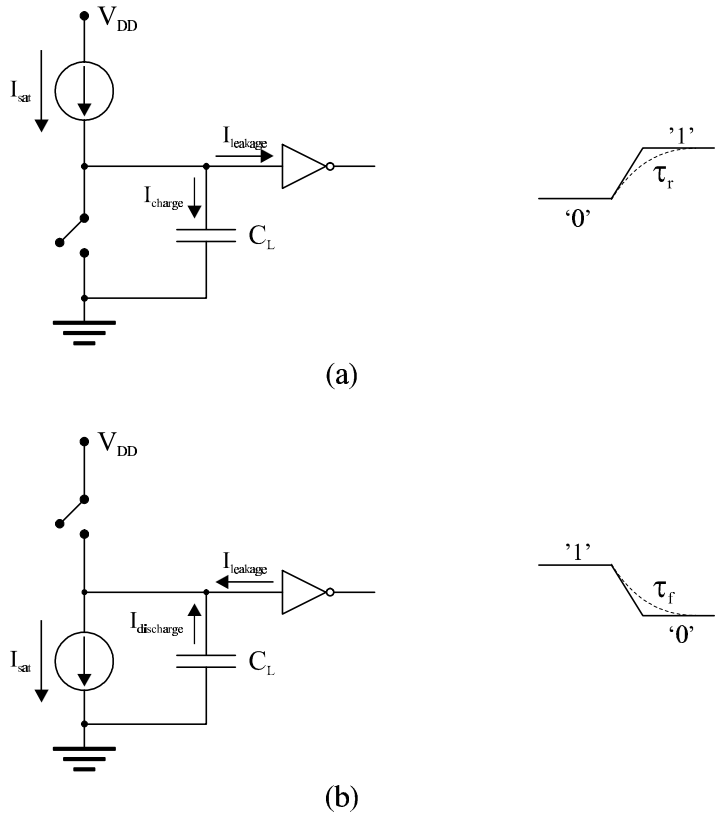


Figure 3.5: Effect of leakage current in timing characteristics. (a) Rise-time. (b) Fall-time.

In both cases, the increase of $I_{leakage}$ due to thin-oxide defects can modify the net charge current of C_L, decreasing its value, since:

$$I_{charge} = I_{saturation} - I_{leakage}$$

Therefore, the time of charge/discharge is delayed. Rise and fall time increases. This can be represented by means of **delay** model, changing (increasing) the gate delay.

Another timing effect related with $I_{leakage}$ is **data retention failure** in memories (particularly significant in EEPROM memories) [Lee *et al.* 2001]. The value of stored data can be modified in time and the data can be lost. This change can be modelled by **indetermination** or **bit-flip** of the cell content.

Hot Carrier Injection

This mechanism is extremely important in submicron CMOS devices where hot carrier concerns are one of the items that define transistor design [Rodder *et al.* 1995]. The degradation in the transistor performance is usually attributed to the presence of fixed oxide charge due to electron and hole trapping in the oxide. Charge trapping is generated by the lateral electric field in the MOS transistor channel. The carriers in the channel, electrons in NMOS and holes in PMOS, will have energy distributions with tails close to their respective barrier heights at the $Si\text{-}SiO_2$ interface. Electrons have higher mobility and channel kinetic energy, and so are more likely than holes to create charge that damage oxides (n-channel is more vulnerable).

Effects of Failure Mechanisms

Gradual degradation of some transistor parameters (V_T, g_m, I_{DS} characteristics) occurs [Hawkins 2000]. The type of degradation depends on the type of transistor:

- In NMOS transistors:
    - Decrease in drain current (I_{DS})
    - Decrease in transconductance (g_m)
    - Increase in threshold voltage ($V_{T)}$
- In PMOS transistors:
    - Effective “shortening” of channel
    - Increase/decrease in I_{DS}
    - Increase/decrease in g_m
    - Increase/decrease in V_T

In a NMOS transistor, for instance, electron-trapping in thin-oxide decreases the channel conductivity and increases V_T. From the expressions that relate I_{DS} and g_m with V_T:

$$I_{DS} = K(V_{GS} - V_T)^2 \text{ (saturation region)}$$

$$I_{DS} = K\left[2(V_{GS} - V_T)V_{DS} - (V_{DS})^2\right] \text{ (linear region)}$$

$$g_m = \left.\frac{\delta I_{DS}}{\delta V_{GS}}\right|_{VDS=constant} = 2K(V_{GS} - V_T)$$

It is clear that electron-trapping causes I_{DS} and g_m decreasing: as V_T increases, $(V_{GS}–V_T)$ decreases, reducing I_{DS} and g_m.

The effect of hole trapping in NMOS transistor is the opposite, that is, V_T decreases. The reason is that the positive charges increase the conductivity of n-channel.

The models at transistor level are the **stuck-off** and **stuck-on** faults, which represents a pseudo-open or pseudo-short between drain and source, with different degrees of effectiveness according to the channel conductivity [Favalli *et al.* 1991].

Models at Logic Level

Pseudo-open and pseudo-short can cause the **indetermination** in output voltages, as the resistance of the output transistors has an intermediate value: $R_{linear} < R_{pmos}, R_{nmos} < R_{off}$.

Timing characteristics can be also disturbed by the change in V_T. From Figure 3.5, and supposing $I_{leakage} \approx 0$ (the load gate is fault-free) and $I_{IL} \approx 0$ (this is a realistic assumption in CMOS gates), we can estimate rise-time and fall-time:

- Rise-time estimation

  The saturation current for PMOS-transistor is given by

$$I_{DSp} = K_p(V_{GS} + |V_{Tp}|)^2$$

  This current charges C_L and, since its magnitude is approximately constant, we have

$$V_{out} = \frac{I_{DSp}t}{C_L}$$

  Substituting for I_{DSp} and rearranging we have

$$t = \frac{C_L V_{out}}{K_p(V_{GS} + |V_{Tp}|)^2}$$

  We now assume that $t = \tau_r$ when $V_{out} = +V_{DD}$, so that

$$\tau_r = \frac{C_L V_{DD}}{K_p(V_{GS} + |V_{Tp}|)^2}$$

  As $V_{GS} \approx -V_{DD,}$

$$\tau_r \approx \frac{C_L V_{DD}}{K_p(V_{DD} - |V_{Tp}|)^2}$$

  This result compares reasonably well with a more detailed analysis in which the charging of C_L is divided, more correctly, into two parts: (1) saturation and (2) linear region of the transistor [Pucknell & Eshraghian 1994].

- Fall-time estimation

  Making similar assumptions and reasoning for NMOS transistor, we may write for fall-time:

$$\tau_f \approx \frac{C_L V_{DD}}{K_n (V_{DD} - |V_{Tn}|)^2}$$

Expressions for rise and fall time indicate that **V_T changes** can cause remarkable modification (notice the non-linear dependency) in **switching delays**. For instance, if V_T increases, switching delays also go up.

The logic model for this mechanism is obviously, **delay**, that is, the modification of the gate delay at logic abstraction level.

Another indication of switching speed may be obtained from the parameter ω_0 (**frequency response**):

$$\omega_0 = \frac{g_m}{Cg} = \frac{2\mu(V_{GS} - V_T)}{L^2}$$

where C_g is the gate/channel capacitance.

This shows that switching speed depends on V_T and on carrier mobility and inversely as the square of channel length. A fast circuit requires that g_m be as high as possible. Changes in g_m or V_T affect to ω_0.

Plasma Damage

This is a mechanism that is becoming more important with new process equipment [Shin *et al.* 1993]. Plasma processes such as etching, deposition, cleaning and ion implantation can all cause damage to the thin oxides in MOS devices. Advanced technologies increasingly use “dry” plasma processing rather than the “wet” chemical processing used in older technologies. The incident plasma builds up charge on metal or polysilicon electrodes usually in the region where the area or periphery is large. This charge is then transferred to the region of the electrode where a high electric field enables some current flow to take place, such as across a thin gate oxide [McVittie 1996].

Effects of Failure Mechanism

The most commonly observed effect is that of **increased oxide leakage** current after plasma damage takes place. This is due to a reduction in the oxide breakdown voltage. In MOS devices, **degradation of the I_D-V_{GS} characteristics** are also observed [Rangan *et al.* 1999, Pagaduan *et al.* 2001] usually in the form of an **increase in the threshold voltage V_T** [Lin *et al.* 1996].

Models at Logic Level

The models can be the same as those of wearout step in gate oxide breakdown, related to the increase of the leakage current. About degradation of the I_D-V_{GS} characteristics and V_T, it is possible to use the same models than those introduced for hot carrier injection.

3.1.2.1.2 Metallization

From this type of fault mechanisms, we can remark electromigration, stress voiding and others (including several mechanisms with similar behaviour and effect).

Electromigration

As technologies advance in sub-micron features sizes and on-chip transistor density increases, the requirements for electromigration immunity are being pushed to the limit. The combination of smaller metal line-widths and higher current density requirements have brought the problem to the forefront of reliability physics. This agrees with the expression for the Mean Time to Failure (MTTF) related to electromigration (EM) [Hawkins 2000]:

$$MTTF = \frac{A}{J^2} e^{Ea/kT}$$

where A is the metal cross-section area, J is the current density, Ea is the activation energy, k is the Boltzman's constant, and T is the Absolute temperature.

Present and future technologies will use 3-8 metal layers. This could give 50-500 million vias and contacts for ICs with 20-100 million transistors. Modern vias are small (< 0.5μm, aspect ratio 2-4, so A is reducing) and fragile, they often have enough metal to allow conduction, but subsequent electromigration is a failure mechanism.

The inclusion of small percentages of copper (< 4% and usually about 1%) in aluminium have enabled higher current densities (J) to be achieved before electromigration occurs. This is attributed to the adsorption of copper at the aluminium grains boundaries thereby occupying positions necessary for the movement of aluminium metal atoms [Ghate 1981].

Nowadays, semiconductor industry is replacing aluminium with copper in interconnects on a wide scale [Tammaro 2000, Ogawa *et al.* 2001]. Copper is used for high performance ICs to reduce intermetal capacitance and increase connectivity (its resistivity is 30 to 40 percent lower than aluminium). This trend will have a positive impact on permanent failure, as copper provides a higher electromigration threshold, comparing to aluminium (higher activation energy Ea).

Effects of Failure Mechanism

Electrically the failures can cause an increase of the interconnect resistance leading to an eventual **open circuit**. Electromigration can also induce **electrical shorts** between two levels of metal or adjacent metal levels. In addition, bond lifts have also been observed as a result of void formation at the bond lands.

Models at Logic Level

Open/Short in metal connections at transistor level. This can cause many types of faults at logic level [Gil 1999]: **stuck-at**, **short**, **open-line** (eventually **stuck-open**), **bridging**, **indetermination** and **delay** (see Figure 3.2).

Stress Voiding

During the wafer manufacturing process, the deposition of the different layers and the associated shrinkage can result in large mechanical stress in the individual layers [Jones 1987]. The thin film metal lines are particularly sensitive to such stress. The transfer of atoms from areas of high stress to equalise the stresses along the line can result in the deformation of the metal. In submicron semiconductor processing, the large tensile stress has been associated with the thermal mismatch between the metal and the passivation films [Tezaki *et al.* 1990].

Effects of Failure Mechanism

Void formation and notching are the physical manifestations of this mechanism [Oates 1993]. Whisker growth has also been observed as result of compressive stress [Turner & Parsons 1982]. Metal movement takes place until the stress has been reduced. Failure is caused by either increased resistance and **open-circuits** in the case of void formation, and **short circuits** due to whisker growth.

Models at Logic Level

Since the faults at Physical Device level can be either open or short in metal connections, the models at Logic level will be the same as for electromigration, because the physical mechanisms are similar.

Others

Also the effect of other mechanisms such as contact migration, via migration and microcracks and failures related with step coverage at wafer topography can be summarised in short/open of metal interconnections [Amerasekera & Najm 1997]. So, the models at Logic level are the same than in electromigration and stress voiding.

3.1.2.1.3 Package and Assembly

Large chip sizes also mean large and complex package design and this is also a major reliability concern. The complexity of the package can make this one of the main product yield limiters in large microprocessor production.

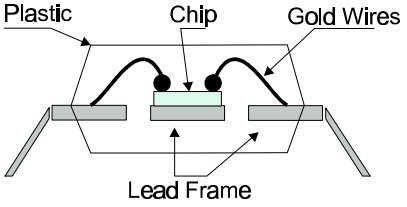


Figure 3.6: Cross-sectional view through a molded plastic package.

A cross-section through a molded package is shown in Figure 3.6 [Amerasakera & Najm 1997]. Four major failure mechanisms can be identified:

1) Die attachment failures associated with mounting of the die to the lead frame.
2) Bonding failures associated with the thin gold wires that make the bond between the chip and the lead frame.
3) Delamination between the plastic packaging and the die.
4) Moisture-related failure mechanisms such as corrosion, delamination and the popcorn effect.

In next subsections, we analyse the main effects of the failure mechanisms and finally we propose some fault models at logic level.

Die Attach Failures

Failure modes are burnout, parametric shifts or corrosion related. Additionally, cracks in the die during excessive mechanical or thermal stress conditions will be catastrophic. Many die attach failures are observed during thermal cycling and High Accelerated Stress Testing (HAST).

Bonding Failures

The most common failure observed is that of open circuits due to bond lifts. The formation of intermetallics can result in high resistances in the bond wire connection [Shirley & Blish 1987]. Wire sweeping due to excessive lagging can result in short circuits between adjacent bond wires. Whisker growth will also result in shorting between adjacent bond wires. High tension in the wires due to backwash, or bonding pressure issues can also lead to fractures in the bond wire and consequently open circuits. Thinning of the bond wire especially when using aluminium can result in localised heating in regions of high resistance along the wire. The thinning is due to oxidation of the aluminium wire, which reduces the effective cross-section. The high current densities lead to an electrical overstress type of thermal melting and eventually to open circuits [King *et al.* 1989].

Delamination and the Popcorn Effect

These are moisture-related mechanisms. Absorbed IC moisture can expand during subsequent temperature cycling and cause various failures. Among them, delamination between the die and the lead frame, and between the lead-frame and the encapsulant (cracking the die) are typical. The second effect is known as *popcorn* effect, because of the sound of the die cracking.

Wire bond degradation, package cracking, pattern shift or displacement of the metallization, and corrosion can lead to electrical failures due to increased leakage, intermittent or open circuits.

Corrosion

Failure is usually due to an increase in the metallization resistance and eventual open circuit. It is also possible to have an increase in the leakage current between adjacent tracks of metallization due to migration or dendrite growth.

Models at Logic Level

The four effects shown above can be modelled mainly with **shorts** and **opens** in wire interconnection lines between logic circuits. The faults can be permanent or even intermittent related with unstable connections.

3.1.2.2 Intermittent Faults

About this type of faults, the following considerations can be made:

- Intermittent faults generated by process variations and manufacturing residuals are going to represent another major source of errors in deep submicron circuits.
- They have higher rates than transient faults.
- As geometries shrink, some permanent faults will initially manifest as intermittent. For instance:
    - **Opens** in the narrower sections of the wires due to electromigration.
    - **Shorts** between adjacent or crossing conductors in the areas where the dielectric layer is thinner.
    - **Leakage** due to direct tunnelling of the current through oxide layers.
- Higher working frequencies increase the impact of timing uncertainties, that provoke violations of timing safety margins ($t_{set\text{-}up}$, t_{hold}) [Constantinescu 2001].

Fault models must be similar to those of permanent faults. In fact, wearout mechanisms reveal initially as intermittent faults until they finally provoke a permanent fault. So, we can use the models described for permanent faults, considering that they appear and disappear with random periods.

3.1.2.3 Transient Faults

Traditionally, radiation (particularly α particles and cosmic rays) has been pointed out as the main cause of transient faults. However, other mechanisms due to high working frequency and new shrinking techniques used in deep submicron technologies, are revealing as other important sources of transient faults.

3.1.2.3.1 Radiation of a Particles

Radioactive impurities (like thorium, uranium, etc.) are present in the materials used in ICs packages. Such radioactive elements can emit α particles with high energy (up to 8 MeV). When the radiation collides the semiconductor, e^{-}-h^{+} pairs are generated. If the number of electrons moving is high enough to exceed the critical charge (Q_{crit}) of the affected transistor, then its behaviour can be altered. This fault mechanism affects especially to DRAM cells and

dynamic registers. For instance, a DRAM cell storing a '1' will change its value to '0', but if the stored value is a '0', nothing happens [Amerasekera & Najm 1997]. This effect can be modelled with a "selective" **bit-flip**.

In [Juhnke & Klar 1995], the soft error rate (SER) due to α particles is calculated, considering future technologies. Their results pointed that for a given technology (specified by its critical charge) as supply voltage decreases the SER increases greatly. In the case of the 0.6 μm technology, reducing the power supply to half the normal implied multiplying the SER by a factor of 8. If both scaling parameters (feature size and power supply reduction) are considered together, it can be expected a higher sensitivity to α particles in newer CMOS technologies, raising their influence in future semiconductor generations.

Transient faults due to α particles are considered almost instantaneous (several picoseconds [Amerasekera & Najm 1997]), but its effect persists until data are rewritten (or refreshed). In [Srinivasan *et al.* 1994], the current wave generated by an α particle is modelled as:

$$I(t) = \frac{Q_{total}}{t_f - t_r}\left\{e^{-t/t_f} - e^{-t/t_r}\right\}$$

where Q_{total} is the total charge collected, and τ_r and τ_f are respectively the pulse rise time and fall time constants (dependent on the technology). From this equation, using an electronic simulator, it is possible to generate a library (table) of pulse durations for different technologies (fixing τ_r and τ_f), by simulating the equation varying the value of Q_{total}.

3.1.2.3.2 Radiation of Cosmic Rays

When cosmic rays enter atmosphere, they collide with atmospheric atoms, producing cosmic particles: photons, electrons, protons, neutrons, pions, etc. Among these cosmic particles, high-energy neutrons (> 1 MeV) have been considered the main source of transient faults in CMOS devices. When a neutron impacts silicon, the charge of transistors can be seriously affected, in such a way that its value can flip (if the charge generated it is higher than the critical charge Q_{crit}). For this reason, the most frequently used fault model is **bit-flip**. Other effects can be also possible. For instance, if the charge generated is near Q_{crit}, the value of the transistor can become indeterminate (**indetermination** fault model) [Gil 1999].

The SER due to cosmic rays has been usually calculated neglecting the effect of low energy neutrons (< 1 MeV). In newer technologies, however, this is not possible, as the critical charge of transistors decreases almost quadratically with gate length (L_G) [Hazucha & Svennson 2000]. For this reason, lower energy neutrons (with a greater flux [Ziegler 1998]) are also able to provoke faults, increasing the neutron-induced SER.

Up to date, faults generated in combinational circuitry were neglected, considering only faults in storage (memory and registers). This was justified because the natural mechanisms of circuits were able to mask most of the faults induced in combinational circuitry [Liden *et al.* 1994]. These natural mechanisms are [Shivakumar *et al.* 2002, Constantinescu 2002]:

- Logical masking. This phenomenon occurs when a fault is produced in a portion of the circuit whose output does not affect the system output (because it is not active, or because the output depends on other portions of the circuit).
- Electrical masking. It consists on the attenuation of an erroneous pulse after trespassing a number of subsequent gates to the point that the pulse does not affect the output.
- Latching-window masking. This mechanism is related to the latching of a fault, becoming an error. Figure 3.7 shows how this mechanism works. The latching-window is the time between $t_{set\text{-}up}$ and t_{hold}. If an erroneous pulse starts before $t_{set\text{-}up}$ and ends after t_{hold}, it will be latched, provoking thus an error. If it ends before $t_{set\text{-}up}$ or starts after t_{hold}, the fault will be masked. Other possibilities can lead to store an indeterminate value in the latch.

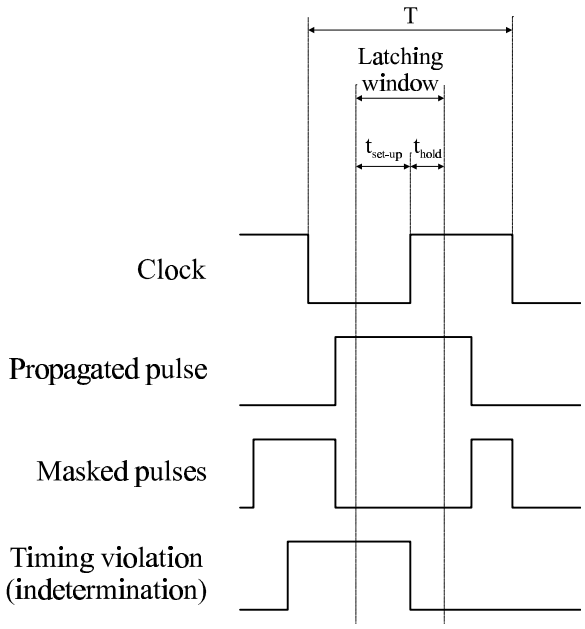


Figure 3.7: Latching window masking

However, in modern devices these masking phenomena seem to reduce (and even to disappear) for several reasons [Shivakumar *et al.* 2002, Constantinescu 2002]:

- The reduction in the number of gates between latches (due to deeper pipelining in processors) affects to logical and electrical masking.
- In newer technologies, impacts produce stronger pulses that cannot be electrically masked so easily.
- As working frequencies raise, the latching-window (t_{hold}–$t_{set\text{-}up}$) decreases, allowing that the probability that a combinational fault is latched (becomes an error) is higher.

In combinational logic, the effect of cosmic rays cannot be modelled as a bit-flip, because once the fault disappears, the output value returns to its correct value. That is, it behaves like a pulse. For this reason, the fault model for this type of transient fault has been called **pulse**, to distinguish from the classical bit-flip (produced in storage). It is important to remark that the incidence of pulses in combinational logic will increase in new technologies.

The duration of transient faults caused by cosmic particles is a matter not very deeply considered. In [Freeman 1996], the current wave generated in an impact is modelled as:

$$I(t) \propto \frac{Q}{T} \times \sqrt{\frac{t}{T}} \times e^{-t/T}$$

where Q is the charge collected due to the particle impact, and T is the charge time constant of the transistor for the technology under study (dependent of the technology). As mentioned for α particles, it is possible to generate a library of pulse durations for different technologies (fixing T), by simulating the equation varying the value of Q.

3.1.2.3.3 Other Causes

Besides radiation, other causes of faults are appearing, due to new interconnection features. For instance, the skin and Miller effects can originate timing violations ($t_{set\text{-}up}$, t_{hold}), thus altering in the delay of transistors (this corresponds to a **delay** fault model). These timing margin violations may corrupt data transferred.

The skin effect [Walker 2000] is the propagation of electrons along the surface of the wires. This effect makes the resistance of the interconnect to vary with frequency (the more frequency, the more resistance).

The Miller effect occurs when two adjacent wires switch simultaneously in opposite directions. When this happens, the effective capacitance between terminals is modified [Sylvester & Keutzer 1999].

3.1.2.4 Related works

Recent works predict that new technological advances will improve the quality of semiconductors, using better materials and processes. This increment in quality will reflect in a lower permanent fault rate [Constantinescu 2002]. However, other aspects also positive such as scaling the power supply and the feature size will make these devices very sensitive to transient and intermittent faults, and their rates will raise greatly in the future [Constantinescu 2001, Shivakumar *et al.* 2002].

In [Shivakumar *et al.* 2002], the evolution of the SER of the different types of circuits in a microcomputer (memory, latches and combinational logic) is quantified for different technologies. The results are very significant: although SER will raise in all types of circuits, combinational logic is impacted by the reduction in masking, leading to a strong increase in its SER, so that by 2011 the values will reach those of memories.

For those reasons, it is necessary to develop and apply some techniques to mitigate the impact of faults. These techniques include optimising the design and manufacturing processes (SOI, triple-well, etc.), providing error detection and recovery mechanisms (parity, ECC, assertion checking, redundancy, etc.), and other architectural solutions (simultaneous multithreading, double-execution, etc.) [Constantinescu 2002, Shivakumar *et al.* 2002].

3.1.3 Summary and Conclusions

In this section, the connection between Physical Device and Logic level has been done using a theoretical study of the physical mechanisms related to faults in modern ICs. After an introduction about fault models for permanent, transient and intermittent faults, new trends in deep submicron technologies are analysed.

Semiconductor technology advances have tremendously increased the performance of computing systems over the past decades. Although, shrinking geometries, lower power voltages and higher frequencies have a negative impact on dependability by increasing the rates of occurrence of permanent, intermittent and transient faults (mainly the last ones). Another negative consequence is the increase in the likelihood of multiple faults. We have concentrated the study in some fault mechanisms of high significance for new technologies:

- For permanent faults: Oxide breakdown, hot carrier injection, plasma damage, electromigration, stress voiding and package and assembly defects.
- For transient faults: Radiation of α particles and cosmic rays, and their effect on combinational circuits.
- For intermittent faults: process variations and manufacturing residuals.

From the study of fault mechanisms we have deduced a set of fault models at Logic level. It is interesting to emphasise that some faults cannot be represented only by means of stuck-at (in permanent faults) and bit-flip (in transient faults), being necessary the inclusion of new models: delay, indetermination, pulse and open-line. From the theoretical study, we have also deduced that in future technologies, faults in combinational logic will have a great importance. In addition, it is predicted that transient and intermittent faults will have an increasing importance respect to permanent faults in submicron devices.

3.2 Propagation of Faults in Combinational Logic into RTL Abstraction Level

The basic idea is to inject faults at Logic level and study the fault manifestation in register elements. To carry out these experiments, we have used a fault injection tool called VFIT (VHDL-based Fault Injection Tool) [Baraza *et al.* 2002] developed by the GSTF (Fault-Tolerant Systems Group) around a commercial VHDL simulator [Model Technology 2001], to run on an IBM-PC (or compatible) platform.

Some questions must be answered:

- What kind of faults can be identified at RTL level?
- What is the percentage of occurrence of each class of fault?
- What percentage of faults at Logic level propagates to RTL level?
- How many registers are perturbed by each fault injected?

We have simulated the VHDL model of the PIC16X [Microchip 2000], a simple 16-bit microcontroller. As the model was completely behavioural, we have modified the ALU in order to have a structural description. This structural description includes combinational

logic. Faults will be injected in this part of the model to study the propagation until the microcontroller registers. On the other hand, the registers were classified (depending on their accessibility by software) in:

- *User* registers (accessible)
- *Hidden* registers (not accessible).

We have applied the fault models deduced in Section 3.1. The injected faults have been:

- Transient: delay, indetermination and pulse.
- Permanent: delay, indetermination, high impedance and stuck-at.

Faults have been injected randomly in all ALU signals. The detailed conditions of the fault injection experiments are:

- **Injection tool**: VFIT
- **Injection technique**: Simulator commands
- **System**: PIC16X microcontroller
- **Number of faults**: 3000 per experiment
- **Workloads**:
    - Arithmetic series of n integer numbers (n = 10)
    - Bubblesort (n = 10)
- **Fault type**:
    - Transient: delay, indetermination and pulse
    - Permanent: delay, indetermination, high impedance and stuck-at
- **Fault duration**: It is generated randomly in the ranges [0.1T-1.0T], [1.0T-10T] and [10T-20T], where T is the clock cycle duration (in our system, T = 100 ns). It has been intended to inject short transient faults, with duration equal to a fraction of the clock cycle (the most common, as described in [Cha *et al.* 1993]), as well as longer faults, which will ensure in excess the propagation of faults.
- **Fault instant:** It is generated randomly in the range [0-$t_{workload}$], where $t_{workload}$ is the workload simulation duration without faults. For arithmetic series, $t_{workload}$ = 5.2 μs, and for bubblesort $t_{workload}$ = 80 μs.
- **Fault targets:** Any atomic combinational signal of the ALU and the general clock line of the microcontroller.
- **Observation targets:** As explained above, the propagation of the injected faults will be observed in the registers of the microcontroller.

We have defined the following figures to be calculated:

- $N_{propagated}$ = Number of propagated faults. A fault is considered propagated if at least one register is corrupted.

- $N_{corrupted}$ = Number of corrupted registers. A register is considered corrupted if it has at least a faulty bit.
- Percentage of propagated faults = $\frac{N_{propagated}}{N_{injected}} \times 100$, where $N_{injected}$ is the number of injected faults = 3000 per experiment.
- Multiplicity = Average number of corrupted registers per single injected fault = $\frac{N_{corrupted}}{N_{injected}}$

  Multiplicity gives an indication of how many registers have been corrupted by a singled fault injected. In fact, we can easily calculate the average number of corrupted registers per propagated fault, from the expression:

$$\frac{N_{corrupted}}{N_{propagated}} = \frac{Multiplicity}{Percentage\_of\_propagated\_faults} \times 100$$

3.2.1 First results

The first simulation results obtained with arithmetic series workload can be summarised in short:

- Percentage of propagated faults and multiplicity increases with fault duration in transient faults.
- Most of faults manifested as *bit-flip*. It is possible to consider other models (especially in transient faults), such as *delay* and *indetermination*, although they have less representativity (i.e. their proportion in relation to *bit-flips* is much lower).

A deeper simulation analysis has included the study of some additional aspects:

- Influence of the workload.
- Propagation of faults into *user* registers.
- Clock frequency influence.

3.2.2 Influence of the Workload

Two workloads have been applied in the experiments:

- The calculus of the arithmetic series of *n* integer numbers.
- Bubblesort for *n* integer numbers.

These belong to two types of workloads (arithmetic calculus and sorting) usually used in VHDL simulation-based fault injection campaigns. In fact, we have used the same algorithms in other injection campaigns carried out when using VFIT [Baraza *et al.* 2002].

The following aspects have been observed:

- Both the percentage of propagated faults and multiplicity differ remarkably. It seems that most complex workloads increase these factors because they produce a higher sensitisation

of faults. Figure 3.8 shows this fact for permanent and transient faults, representing the percentage of propagated faults.

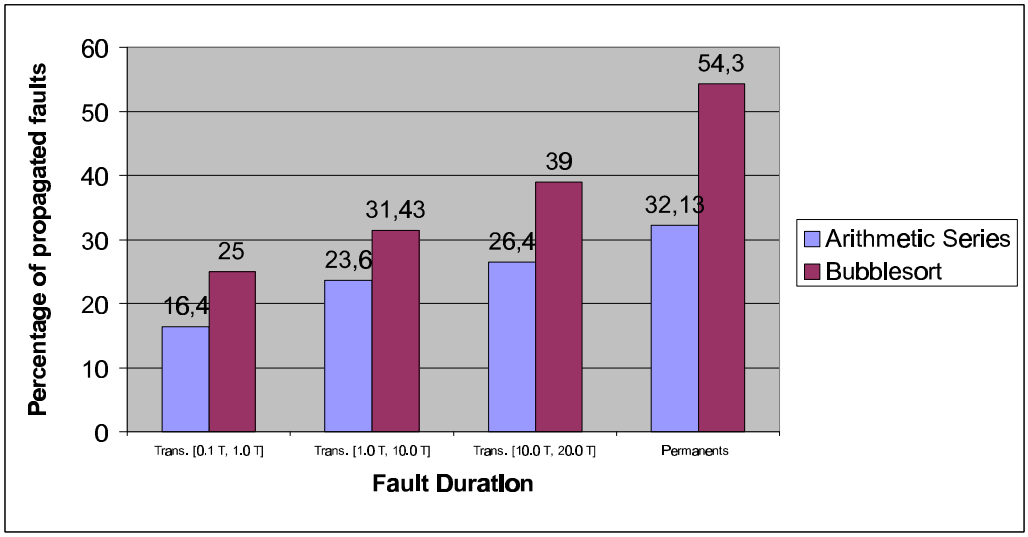


Figure 3.8: Influence of workload in the percentage of propagated faults, for different fault durations.

- The most common types of faults are still bit-flip and (with less influence) indetermination. There have been observed some changes in percentages. Figure 3.9 reflects this situation for permanent and short transient faults.

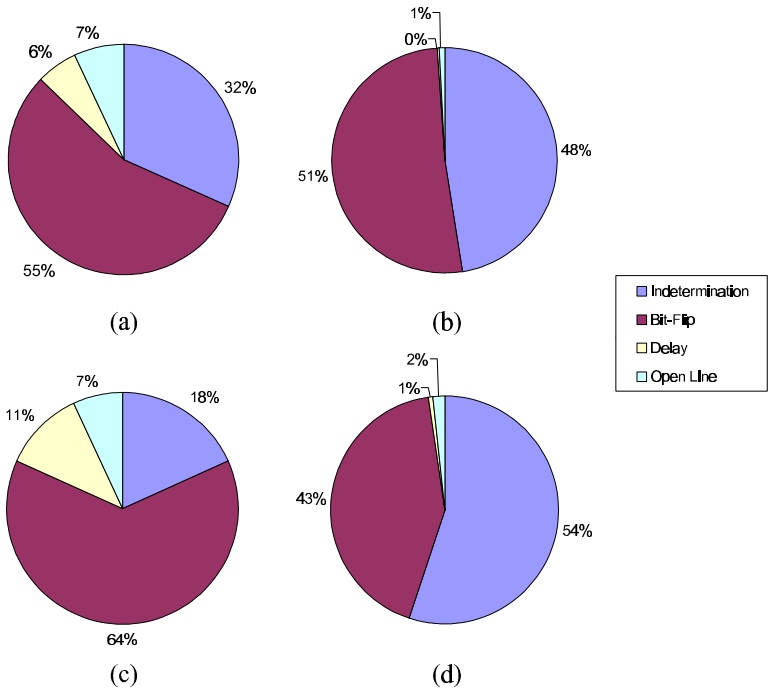


Figure 3.9: Influence of the workload in the distribution of fault types manifested in registers. a) Permanent faults. Arithmetic series. b) Permanent faults. Bubblesort. c) Transient faults (duration [0.1T, 1.0T]). Arithmetic series. d) Transient faults (duration [0.1T, 1.0T]). Bubblesort.

3.2.3 Propagation of Faults into User Registers

We have also studied the propagation of faults into *user* registers, that is, registers that are reachable via software. The objective of this injection campaign was to verify if the general tendencies shown in first results were accomplished for these registers. We also wanted to compare the incidence in *user* registers with the incidence in the global case considering all the registers of the system.

We have observed:

- Small differences in the percentage of propagated faults between *user* registers case and all registers case.
- Multiplicity in *user* registers case is notably lower than in the all registers case, as can be seen in Figure 3.10. The high value that presents the delay fault is due to the special signals (in our system) affected by this fault. In fact, this signal is the global system clock, and the delay fault changes its period. So, we realise that it is a very critical signal, because it synchronises the operation of the registers.

These two previous facts lead to the conclusion that almost all injected faults affect at least one user register, but there are many other non-user (*hidden*) registers affected at the same time.

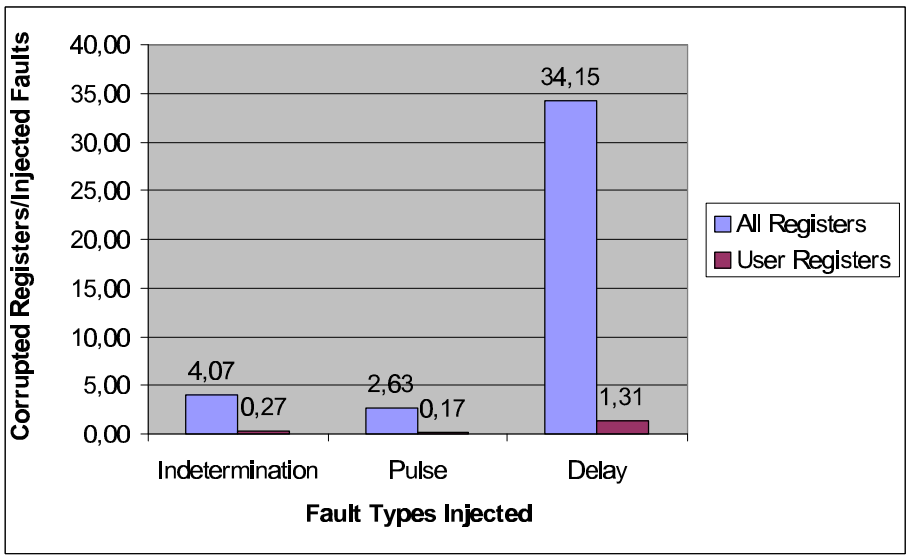


Figure 3.10: Incidence of the different fault types injected in *User registers* and *All registers*. Transient faults injected with fault duration [0.1T-1.0T]. Bubblesort.

Respect to the type of faults propagated in *user* registers, the most common faults are bit-flips followed by indetermination. For permanent faults, the percentages are nearly the same as in all registers case. For transient faults, we have observed higher bit-flip percentages than in all registers case.

3.2.4 Influence of the Clock Frequency

We have modified the clock frequency of the system. The main results are:

- The increase of the frequency provokes a raise of the percentages of propagated faults for transient faults, as expected. In fact, higher frequencies will raise the probability of storing transient erroneous data generated at logic level. Figure 3.11 shows this situation.
- No dependency on frequency for permanent faults.
- Small changes in multiplicity.
- Types and percentages of faults in registers have little differences. Bit-flip continues to be the most common type of fault.

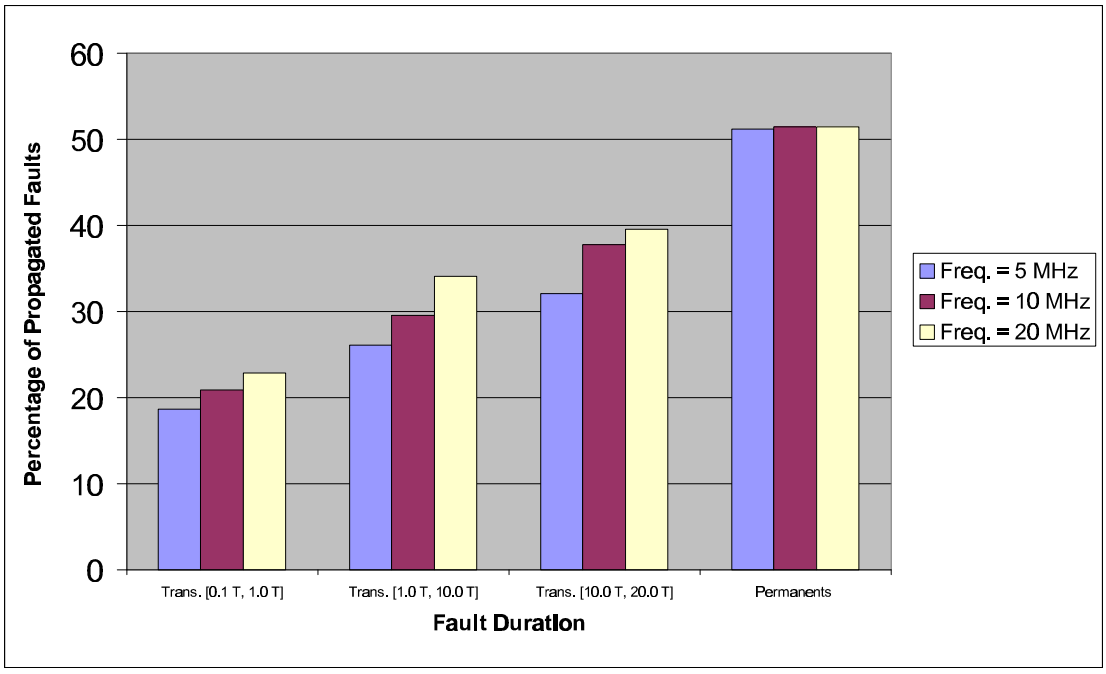


Figure 3.11: Influence of the clock frequency in the percentage of propagated faults.

3.2.5 Summary and Conclusions

The experiments performed in this section have been oriented to verify the effect that injecting faults at Logic level produces at RTL level. The basic idea was to inject faults into combinational logic and study the fault manifestation in register elements.

To carry out these experiments, we have used a fault injection tool called VFIT (VHDL-based Fault Injection Tool) developed by the GSTF (Fault-Tolerant Systems Group) around a commercial VHDL simulator to run on an IBM-PC (or compatible) platform. We have simulated the VHDL model of a typical microcontroller. Faults deduced in Section 3.1 have been injected at combinational logic to study the propagation until the microcontroller registers.

The results obtained show that their effects are very harmful in the registers of the system. More in detail, we have found some facts that should be considered when injecting faults at RTL level:

- Most of faults manifested as bit-flip. It is possible to consider other models (especially in transient faults), such as delay and indetermination, although they manifested in a lower proportion.
- A single fault at Logic level can perturb several registers at a time. This fact, called *multiplicity* in this document, depends greatly on the type of injected fault and on the workload. We have found values from a few registers to some tens in critical cases.
- Workload has a notable influence on the results. So, it would be important to use workloads related with real applications running on each target system, or a complete set of artificial/synthetic workloads to sensitise as much faults and target elements as possible.
- Many *hidden* registers (not accessible by software) are affected. Not only user registers must be taken into account.
- When injecting transient faults, the working frequency of the target system has also a great influence on the percentage of propagated faults. We have found almost a linear dependency between them.

3.3 Impact of Faults at RTL Level in System Behaviour

In previous section, we have studied fault propagation in the microprocessor registers. We have seen that most of them are manifested as bit-flip. It is possible to consider other models (especially in transient faults), such as delay and indetermination, although they have lesser representativity.

The objective now is to inject faults in registers to verify the impact of these faults on system behaviour. We have counted the number of failures[5] when executing the system workload. The figures that we have considered are:

- $N_{failures}$ = Absolute number of failures occurred in the execution of the workload
- $N_{injectable}$ = Number of bits in registers that can be injected into, that is, the number of atomic fault targets
- $NR_{failures} = \frac{N_{failures}}{N_{injectable}}$ = Relative number of failures occurred in the execution of the workload

This last figure represents the number of failures per injectable bit of registers. It is a relative expression for the number of failures, taking into account the size of the potential fault targets. This global size has been calculated adding the sizes (in bits) of the selected registers to be injected.

We have also distinguished between *user* registers (registers reachable via software) and *hidden* (or non-user) registers. We are interested in checking the relative impact of faults

[5] A failure happens when the result of the workload is erroneous.

produced in registers that cannot be reachable by software. This question is important because these faults could not be reproduced applying some widely used fault injection techniques, such as SWIFI (Software Implemented Fault Injection).

The experiment conditions have been:

- **Injection tool**: VFIT
- **Injection technique**: Simulator commands
- **System**: PIC16X microcontroller
- **Number of faults**: 3000 per experiment
- **Workload**:
    - Arithmetic series of n integer numbers (n = 10)
    - Bubblesort (n = 10)
- **Fault type:** Single bit-flip in registers, as it is the most representative fault model according to the previous analysis (see Section 3.2)
- **Fault duration:** Generated randomly in the range [0.1T-1.0T]. It has been intended to inject short transient faults, with duration equal to a fraction of the clock cycle
- **Fault instant:** Generated randomly in the range [0-$t_{workload}$]
- **Fault targets:** All the registers of PIC16X microprocessor. Two classes of registers are considered:
    - *User* registers, reachable via software
    - *Hidden* registers, not reachable by software

3.3.1 Results

Table 3.1 shows the number of failures generated after injecting 3000 faults in system registers. The number of failures has been expressed in absolute and relative format. Two workloads are considered. The registers are classified as *user* registers and *hidden* registers.

Table 3.1: Number of failures. 3000 transient faults injected in registers. Fault model: bit-flip.

Workload	Register type	Size of fault targets (in bits)	Absolute number of failures	Relative number of failures (Number of failures/Size of fault targets)
Arithmetic series	User	91	186	2.04
	Hidden	343	197	0.57
Bubblesort	User	251	30	0.12
	Hidden	343	115	0.34

Figure 3.12 summarises previous results in a graphical way.

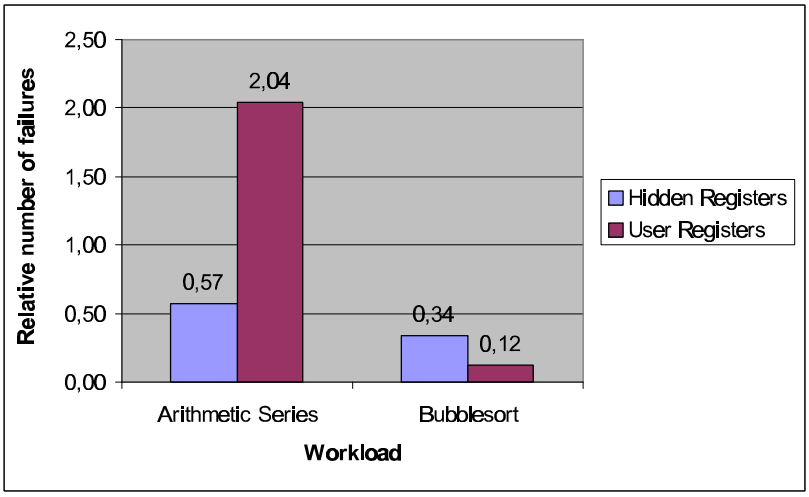


Figure 3.12: Relative number of failures in function of workload and register type. 3000 transient faults injected in registers. Fault model: bit-flip.

3.3.2 Conclusions

Some conclusions can be extracted from the analysis of the results:

- The lower (Fault duration/$T_{workload}$) rate can cause, respect to the differences between the two workloads, the lower failure percentages observed in bubblesort. This yields a lower incidence of the faults. In fact, $T_{workload}$(Bubblesort) is much higher than $T_{workload}$(Arithmetic series), while the fault duration is the same.
- In relation to the differences between the two classes of registers, it has been observed for arithmetic series workload that the impact of faults injected in *user* registers is higher than the impact of faults injected in *hidden* registers. In the case of bubblesort, which is a more complex workload, the higher impact corresponds to *hidden* registers. The cause of this last result can be related to the critical role of some *hidden* registers that are sensitised by the workload. For instance, the status register (which contains the ALU flags), the instruction register, the constant generator registers and some registers that store intermediate data of ALU calculations.
- In both cases, the absolute number of failures generated by injection in *hidden* registers can be considered as not negligible. To make a deep and realistic validation based on fault injection, we should be capable to inject into *hidden* registers (besides *user* registers, of course), as shown in previous example. In any case, it is obvious that the relative impact of *hidden* registers will depend on the workload and the specific system architecture.

3.4 Summary, Conclusions and Recommendations

Hardware faults encompass the Physical Device, Logic and RTL levels. In this section, the connections between the three abstraction levels have been established.

In Section 3.1, the connection between Physical Device and Logic level has been done using a theoretical study of the physical mechanisms related to faults in modern ICs. From this analysis, it is interesting to emphasise that some faults cannot be represented only by means

of stuck-at (in permanent faults) and bit-flip (in transient faults), being necessary the inclusion of new models: delay, indetermination, pulse and open-line. From the theoretical study, we have deduced also that in future technologies, transient faults in combinational logic will have a great importance.

The experiments described in Section 3.2 have been oriented to verify the effect that injecting faults (using VHDL-based fault injection) at Logic level produces at RTL level. The basic idea was to inject faults into combinational logic and study the fault manifestation in register elements. The results obtained have shown that their effects are very harmful in the registers of the system. More in detail, we have found some facts that should be considered when injecting faults at RTL level:

- Most of faults manifested as bit-flip. It is possible to consider other models (especially in transient faults), such as delay and indetermination, although they manifested in a lower proportion.
- A single fault at Logic level can perturb several registers at a time. This fact, called *multiplicity* in this document, depends greatly on the type of injected fault and on the workload. We have found values from a few registers to some tens in critical cases.
- Workload has a notable influence on the results. So, it would be important to use workloads related with real applications running on each target system, or a complete set of artificial/synthetic workloads to sensitise as much faults and target elements as possible.
- Many *hidden* registers (not accessible by software) are affected. Not only user registers must be taken into account.
- When injecting transient faults, the working frequency of the target system has also a great influence on the percentage of propagated faults. We have found almost a linear dependency between them.

As not all the registers of a system are identical (and, what is more important, not all the registers are accessible by software) a deeper study has been carried out in Section 3.3. Here, the registers of a microcontroller have been classified depending on their accessibility in *user* and *hidden* registers. Then, faults have been injected into the two types of registers, and their effects into the workload result have been analysed. When analysing the effects of injecting faults into both types of registers, we have found that the number of failures generated by injecting in *hidden* registers is not negligible. So, to make a deep and realistic validation based on fault injection, it should be possible to inject also into *hidden* registers. For this reason, we think that SWIFI techniques should be complemented with another injection technique to access such registers.

4 Software Fault Representativeness from OS Point of View

4.1 Introduction

This chapter proposes an approach to analyse the effects of real and injected faults. The three proposed fault injection techniques applied at operating system level are: i) provision of invalid values to the parameters of the kernel calls, ii) corruption by bit-flip of the kernel calls, and iii) corruption by bit-flip of the input parameters of the internal functions of the kernel.

The objectives of this analysis are as follows: i) study the possible equivalence between the fault injection techniques at API level, ii) compare the effects of API injected faults with internal injected faults, and iii) analyse the representativeness of the injected faults with respect to real faults, based on the nature of the produced errors.

The effects are analysed with respect to the kernel failure modes and by means of assertions placed in the kernel. The selection of these assertions is based on the analysis of error propagation of real faults observed on Linux device drivers and of the software source code.

The chapter is structured as follows. Section 4.2 presents the strategy we propose to carry out the analysis of the effects (errors) provoked by both real and injected faults. Section 4.3 describes the experimental framework. Some of the results obtained are then presented and discussed in Section 4.4. Finally, Section 4.5 summarises the chapter and draws some conclusions.

4.2 Proposed Strategy to Analyse the Effects of OS Software Faults

Many fault representativeness studies targeted the physical faults and they all agreed on the fact that the bit-flip is a representative fault model of physical faults (e.g., see [Cheynet *et al.* 2000]). However, besides a few studies [Christmansson & Chillarege 1996, Madeira *et al.* 2000], less attention has been paid to software faults, which are considered as the first causes of actual system outages [Lee & Iyer 1995].

The Orthogonal Defect Classification (ODC) reports a representative fault model of software faults [Chillarege *et al.* 1992]. It classifies the software faults that occurred during the development phase of OSs developed by IBM. Also, practice shows that despite the permanent nature of software faults, the errors they produce are very similar to those provoked by transient faults, since their activation is very dependent on the state of the system.

In this section, after the identification of the interfaces of an OS, we present successively the real and injected software faults that we are considering. The obtained results and the experimental observations to derive them are then described in detail.

4.2.1 Structure of an OS and of its Interfaces

As it orchestrates the execution of the different application processes, the kernel is the most critical part of an OS. The kernel gathers the functional components that provide various services to user applications or to manage the hardware. These services are provided through various interfaces. The Application Programming Interface (API) is the most important. This API supports all the system libraries.

The kernel uses device drivers to communicate with the surrounding physical environment. They can be considered as part of the kernel since they run in the same context (the kernel context). These components are also critical. The kernel provides the services necessary to implement them. However, for performance reasons some error detection mechanisms are omitted. Thus, a poorly written driver could corrupt the kernel internal state.

Finally, the hardware provides elementary services to the OS to execute. This interface is used for example at the system start-up to initialize the exception and interruption handlers.

Even a well-tested kernel contains some residual faults. However, their number is insignificant compared to faults contained in device drivers **¡Error!Marcador no definido.**. Also, as stated in [Murphy & Levidow 2000] OS failures are generally caused by errors propagating from device drivers.

In our study, we focus on *software faults*, which can be either internal or external to the kernel. We will consider the faults located at API level as *external* faults and those located in the internal functions and the drivers as *internal* faults.

4.2.2 Real Software Faults in Linux

The study of the real faults that are observed in the OS kernel source code permits us a better understanding of the erroneous behaviours that are provoked in reality. One way to collect such information is the analysis of the change logs provided with each new kernel version. They include comments about the various fixes and additions applied to the previous version. However, it is not always easy to associate a comment with its corresponding fix.

A meta-compiler developed at the Stanford University [Engler *et al.* 2000] permits the detection of real faults in OS kernels. It is based on system-specific checkers defined after a static analysis of the source code. The goal of a checker is to verify programming rules inside the kernel. The faults revealed by these checkers are published on the web. Table 1 presents some of the checkers used to reveal faults within the Linux kernel source code and gives the number of the real faults found in version 2.4.0 (the version we used in our experiments).

The developers of these checkers assume that the triggering of the first three checkers in Table 4.1 corresponds to real faults. The types of real faults that we will consider are those revealed by the BLOCK and the NULL checkers. The outcomes that are likely to be induced by these faults are respectively “Kernel hang” and “Exception”. The latter is usually followed by a “kernel panic” mode. As presented in Table 4.1, they correspond respectively to 42% and 25% of the faults revealed by the considered checkers. The analysis of their eventual provoked errors allows us to develop some assertions explained more in detail in Section 4.2.4.3.2.

Table 4.1 Linux kernel checkers and the number of their respective revealed faults

Checker	Description	Nb	%
BLOCK	Check if blocking functions can be called when interruptions are disabled or when spinlocks are held.	**206**	**42**
NULL	Check potentially NULL pointers returned from routines.	**122**	**25**
VAR	Check the allocated large stack variables (> 1K) because of the limited size of the kernel stack to 8K.	30	33
RANGE	Check the bounds of loops and array indices derived from user data.	47	
LOCK	Check if the locks are released and if they are not double acquired. This checker is generally useful for multi-processor systems.	26	
INTR	Check if disabled interrupts are restored.	27	
FREE	Check that freed memory is not reused.	17	
PARAM	Check if user pointers are dereferenced.	10	
SIZE	Check if the Allocated memory is enough to hold the type for which you are allocating.	3	

4.2.3 Models for Injected Software Faults

We present hereafter the considered model for the software system from which we derive the injected fault models. We distinguish between external and internal faults.

4.2.3.1 Functional decomposition and software system model

Four main entry points can be identified, through which Linux kernel functions are executed. Indeed, a switch to kernel mode can be triggered by: i) an interrupt issued to the CPU by a hardware device to indicate that it requires attention, ii) an exception signalled by a CPU because of an error, iii) a kernel call (or system call) issued by an application or iv) the execution of a kernel thread. The activation of kernel internal functions depends on these entry points, but also on the current state of the kernel. In this work, we concentrate on the third entry point: kernel calls issued via the API, which is the suitable interface to develop portable and easy to use dependability benchmarks.

Based on the work presented in [Bowman *et al.* 1999], and to facilitate the analysis of the Linux kernel, we decomposed it into five functional components: scheduling, memory management, synchronisation, file system(s) management and communication. This functional decomposition of Linux, which is a monolithic operating system, is only used to facilitate the analysis. Each functional component is composed of elementary functions.

It is worthwhile to distinguish the elementary functions that are reachable from the API (kernel calls) from those that are not (internal functions). Based on the work presented in [Devera 2001], by modifying the gcc compiler, we were able to generate at kernel compilation a call graph for each kernel call. A call graph identifies the elementary functions called by the considered kernel call. For each kernel call, we define depth levels. As an example, Figure 4.1 describes the call graph for the kernel call sched_setscheduler that has three depth levels. The "system_call" node is present in all call graphs associated with any kernel call. It represents the kernel call entry point.

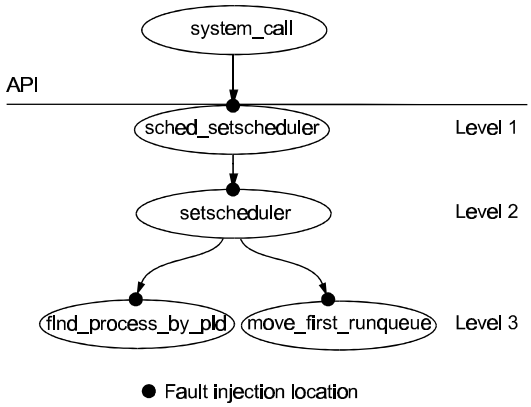


Figure 4.1: Call graph for sched_setscheduler kernel call

In this work, three fault models are considered. The goal is to analyse the degree of similarities of the erroneous behaviours reported for the kernel as a consequence of fault injection at the first level (API) and in lower levels. A fault model is defined with respect to the fault type and to the fault location. The fault types used are *bit-flips* and *invalid parameters*. We consider two locations, either the parameters of the targeted kernel call (i.e., *external faults*) or the parameters of the underlying kernel functions (i.e., *internal faults*).

Another important fault parameter is the trigger condition of the fault. The trigger condition is the event that leads to the injection of the fault. The considered trigger condition is the interception of the targeted system call. The fault is thus injected depending on the selected level (external or internal).

In the rest of this section, we address successively external and internal faults.

4.2.3.2 External faults

External faults mimic faults from the application level and they test the robustness of the kernel.

Bit-flip is a generic fault model. It primarily simulates hardware faults. The goal of the proposed comparison is to determine to what extent it can simulate the invalid argument injection technique, which is a more focused fault model for software faults.

We compare error sets caused by bit-flips (as for MAFALDA [Rodríguez *et al.* 1999]) into system call parameters with those caused by invalid parameters (as for Ballista [Koopman & DeVale 1999]). The parameter values are corrupted by i) issuing exhaustive bit-flips (32 per parameter) or, ii) replacing them with invalid values. Based on the work related to Ballista, and especially its online demonstration site, eight classes of invalid parameters are defined.

Table 4.2 summarises the set of invalid values for each data-type class used in the Linux API. These values are either invalid or close to the limit of the domain of validity. Indeed, the goal is to stress the system as much as possible.

Table 4.2 Data type classes

Permission flag	-1	0	ULONG_MAX (all bits = 1)			
Integer	INT_MIN	0	INT_MAX			
Unsigned Integer	INT_MIN	0	ULONG_MAX			
Process identifier	-1	0	INT_MAX			
File descriptor	Empty file	Deleted	Closed	Read pointer after EOF	Read pointer before begin	
Read pointer	Empty	-1	NULL	Non NULL	Freed	Random
Write pointer	Small (1 Byte)	Far (p + 4 MB)	NULL	Low (0x00000010)	Negative	Random
Time pointer	Negative	NULL	Random	INT_MAX		

4.2.3.3 Internal faults

Internal faults emulate various classes of faults such as those classified in the Orthogonal Defect Classification [Chillarege *et al.* 1992] (e.g., assignment, checking, interface, etc). We consider only the interface class. The set of experiments targets the kernel internal functions that are not reachable via the API.

4.2.4 Expected Results and Observation Levels

We present hereafter the expected results of this work and the proposed observations (i.e., experiment outcomes and internal assertions) necessary to obtain these results.

4.2.4.1 Expected results

The main objectives of the conducted experiments are to:

- Study the possible equivalence between the fault injection techniques at the API level,
- Compare the effects of API injected faults with internal injected faults,
- Analyse the representativeness of the injected faults with respect to real faults, based on the nature of the produced errors.

The comparison between two fault models is achieved through the comparison of their respective effects. The highest abstraction level consists in the quantification of the observed failure modes. From the point of view of the user of a dependability benchmark, if the observed failure modes, after the injection of two different fault models, are different we can state that these fault models are not equivalent. However, if the failure modes are similar, a refinement of the observations might be needed to be able to provide more affirmative conclusions. The first refinement we are considering consists in taking into account the error detection mechanisms built-in within the kernel to enhance the observability capabilities. The second refinement to further enhance the observability consists in implementing extra internal assertions within the kernel. These assertions are described in Section 4.1.4.3.3.

4.2.4.2 Experiment Outcomes

We distinguish two main types of outcomes as the result of a fault injection experiment: *reported* and *non-reported failures*. For the sake of conciseness, both types will be considered as *failures modes*. The considered outcomes are detailed hereafter.

Reported failures

- A hardware exception is raised. If the exception is raised in user mode, the kernel sends a signal to the process that caused the exception. However, if it is raised when running in kernel mode, either the running process is killed or the kernel enters the "panic" mode.
- An error code is returned. As the accuracy of the error reports is not the main objective of our study, we do not discriminate the cases when an incorrect error code is returned (also termed as "hindering" in [Koopman *et al.* 1997]).

Non-reported failures

- The kernel hangs. A kernel hang can be caused either by an infinite loop within the kernel or when it is waiting for an event that never occurs while interrupts are disabled.
- An application hangs. This can be due to an infinite loop that the application executes. Another possibility is when the application is waiting on a kernel wait queue for an event that will never occur. In the latter case, the application does either accept signals, in which case we can force it to quit, or it does not accept signals, in which case we need to reboot the kernel.

When none of the previous events is observed, then a "*no signaling*" outcome is assumed.

4.2.4.3 Internal assertions

This section presents the additional observation mechanisms that permit a finer tracing of the errors provoked by the injected faults. After the presentation of the considered error propagation model, we present the various assertions that are inserted within the kernel.

4.2.4.3.1 Error propagation model and assertions: general idea

An execution trace consists of a sequence of internal states. They provide fine grain observations for monitoring the internal errors.

The internal state of a program is defined by the set of variables of the program and their respective values and also by the value stored in the instruction pointer register. We consider that the internal state is erroneous if it contains at least an incorrect data value. An error is characterised by the couple (correct value, incorrect value) associated to a variable or to the instruction pointer. In addition, we distinguish between errors in the data flow and errors in the control flow.

The mechanisms that influence whether an error propagates are i) the creation, ii) the cancellation and iii) the hiding of the error. An error can be either an initial error (i.e., created as the result of the activation of a fault), or propagated by another error. The cancellation of an error is characterised by the flushing or overwriting of erroneous data. Finally, if a data

error remains unchanged and its value is not used, then we say that the error is hidden; it remains as a latent error. The cancellation and hiding mechanisms contribute to the "no signaling" outcome.

We define a kernel control flow as the sequence of instructions carried out by the kernel following a kernel call, an interruption or an exception. The code of Linux is reentrant, i.e., several control flows are carried out in parallel.

It is easy to analyse the state of a simple program, e.g., by comparing its state with a reference state [Daran & Thévenod-Fosse 1996]. However, it is impossible to realise such a task for a software executive kernel. The occurrence of random asynchronous events, such as hardware interruptions, and the fact that Linux code is partially reentrant, make it impossible to fully master its state. Due to the non-determinism attached to the behaviour of the target system and the wide range of the considered faults, looking for an exact matching of experiments would be irrelevant. Peripheral cards and the scheduling of kernel threads are the major causes of non-determinism. To minimise the system non-determinism, we execute the experiments just after the system ends booting. Also, we try to disable some drivers and kernel threads for each campaign, though this is not always possible in practice as this could divert the system from its nominal configuration. Practically, a specifically designed tracing tool inserts breakpoints into the kernel to monitor various events (system call trap, interrupt handling, context switch, etc.). This allowed us to reveal various causes of system non-determinism.

The approach we have used is to carry out comparisons with respect to the expected behaviour of the kernel and also by reducing the granularity of the considered state, i.e., by taking into consideration only the relevant part of the global state. The level of granularity that we consider is the kernel internal functions (Figure 1). The internal kernel-state is defined, at internal function entry or exit points, called *checkpoints*, by a data structure that contain the values associated to some critical variables. A checkpoint can belong to several kernel control flows. The combination of these values determines to some extent the consistency of the kernel-state. Examples of checkpoints are:

- activation of a fault, which corresponds in our context to the execution of the faulty instruction,
- raising of an exception,
- kernel-call entry and exit.

Additionally, checkpoints are implemented by means of assertions located at the level of internal functions. We have developed two types of assertions detailed in the following two subsections. The first type is based on the analysis of propagation paths of some real faults presented in Section 4.2.2. The second type consists in implementing extra observation and error detection mechanisms within the kernel. Such mechanisms could be easily implemented by kernel developers, but they are seldom included, essentially for performance reasons.

4.2.4.3.2 Assertions based on real faults

The Linux real faults database presented in Section 4.2.2 has shown that BLOCK and NULL fault classes are the most frequent in the Linux kernel. They are most of the time located in the Linux device drivers. For Linux developers, while some errors can cause the kernel to enter an endless loop and thus leading it to hang, most errors manifest either as null pointer dereferences or by the use of other incorrect pointer values (the usual outcome of such errors is an "oops" message). As a consequence, we put emphasis on BLOCK and NULL classes of faults.

We have analysed the propagation of the effects induced by these faults. We identified the errors that are likely to be provoked by these faults at kernel level. Then we implemented assertions that monitor such errors.

BLOCK faults correspond to calling blocking functions when interrupts are disabled or in the case of multiprocessor systems, when spinlocks are held. We identified all the internal functions present in the Stanford real fault database (see Section 4.2.2) that are considered as blocking functions. We implemented assertions at 8 of these functions entry points to check whether interrupts are disabled.

To implement the assertions associated with the NULL class of faults, we identify three basic functions (`__kmem_cache_alloc, __alloc_pages, __vmalloc`) that are used to allocate memory. The assertions consist in the test of the returned values.

4.2.4.3.3 Other specific assertions

The development of this type of assertions is based on the analysis of internal functions. They are inserted at the entry and exit points of internal functions, part of the call graph of a given kernel call.

As an example, the internal function `find_task_by_pid` that is present in the call graph of the `sched_setscheduler` kernel call, takes a process identifier as an input parameter and returns a pointer to the structure that characterises the process. A simple assertion is to verify that the process identifier associated to the returned structure is correct, i.e., equal to the input parameter.

In addition to these extra detection mechanisms, we implement assertions that monitor global kernel variables indicating the global kernel-state. They give us an internal view of the whole impact of the injected faults. We select indicators for each functional component. The memory pressure for example reveals the capacity of the kernel to serve user applications in term of memory allocation. Thus, in the main function of Linux dealing with memory allocation, we placed an assertion that gives us the value of such variable in order to post-analyse the evolution of memory pressure during an experience.

4.3 Experimental Framework

In this section, we present the main features of the experimental framework set up to conduct the experiments.

4.3.1 General Description

We associate a fault injection technique to each fault model. The three considered injection techniques are thus: i) provision of API invalid parameters, ii) bit-flip in API parameters, and iii) bit-flip in internal function parameters.

We recall that the goal is to inject various faults and to observe and compare their consequences. To ensure comparable results, we have developed a versatile tool supporting the application of the three injection techniques. Each technique requires four main steps:

- The kernel calls issued by the workload that the tool is tracing are intercepted. The tool uses Linux ptrace() interface and intercepts kernel calls in user mode as in [Akkerman 2001]. The kernel call that is targeted by the fault injection experiment is thus interrupted.
- A fault is injected according to the associated model (i.e., technique). The injection process ensures the synchronisation between the fault and the workload and thus allows for result comparison for the three techniques.
- The execution of the interrupted kernel call is resumed.
- The system behaviour is observed.

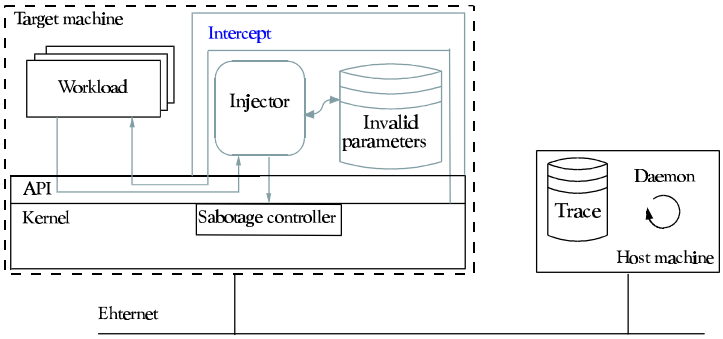


Figure 4.2: Framework general description

Figure 4.2 illustrates the experimental framework, based on two separate machines: target and host. The target kernel (version 2.4.0 of the Linux kernel) is installed on the *target machine* along with the injection tool. The hardware platform of the target machine is based on a Pentium III processor. Specific modules provide the capabilities that are specific to each kind of injection technique. The fault injection capabilities are detailed in Section 4.3.2.2. The aim of the *host machine* (that is connected to the target machine through an Ethernet link) is to monitor the target machine and to reboot it with the adequate options in case of a hang.

The location and the time of an injected fault depend on the executed workload. For each of the functional components presented in Section 4.2.3.1, we have developed an independent workload that activates the associated kernel calls.

We define a fault injection campaign per kernel call. During a campaign, we apply the three fault injection techniques to the kernel call.

4.3.2 Architecture

In addition to the modular workloads, two principal software modules characterise the proposed experimental framework i) the fault injection module, ii) the observation and experiment controller module.

4.3.2.1 Modular Workloads

The current framework applies to all kernel functional components. In this work, we focus the analysis on the **scheduling** and the **memory** components, since they provide the basic OS services.

The scheduling component performs three tasks: i) creation and destruction of processes, ii) scheduling of processes and iii) management of timers and interrupts. The developed workload activates the elementary functions associated with this component in a simple way. We have selected six kernel calls to be activated by this workload: i) `sched_setscheduler`, `setpriority` and `wait4` for the process scheduling task and ii) `setitimer`, `nanosleep` and `gettimeofday` for the timer management task. Other system calls associated with the scheduling component are used by the workload but are not relevant to our study since they have no parameters, such as the `fork` kernel call.

The workload is composed of three processes: a main one that creates two children. The main process changes its priority (`setpriority`) and creates two other processes before yielding the processor and waiting for the end of the other processes (`wait4`). One of the newly created processes sleeps for 5 ms (`nanosleep`), while the other one changes its scheduling policy to FIFO (`sched_setscheduler`). The sleeping process wakes up and issues various calls (`setitimer`) to update its timers. In fact, the kernel provides each process with three interval timers, each decrementing in a distinct time domain. When one of these timers expires, a signal is sent to the process, and it can restart. All the processes issue, along the execution of the workload, the `gettimeofday` call.

The memory component ensures the separation of process memory spaces and implements the virtual memory mechanism. It is also responsible for mapping files into memory. The selected workload is extracted from an existing performance benchmark [Finkel *et al.* 1992]. It solves a mathematical model of a jigsaw puzzle. It builds a puzzle, scrambles tiles and record the time required to solve the jumbled puzzle. This benchmark is useful to study memory allocation and paging behaviour.

It activates the following system calls i) `mmap` and `munmap` for mapping and unmapping files into memory, ii) `mprotect` to control allowable accesses to a region of memory and iii) `brk` to change the memory data segment size.

4.3.2.2 Fault injection

Depending on the considered fault model, the injector module controls the injection of the adequate fault.

As presented in Section 4.2.3.2, we consider two external fault models. A kernel call parameter is corrupted by a bit-flip or is substituted by an invalid parameter.

The injection of faults into internal function parameters is subtler. We distinguish the fault insertion phase from the fault-enabling phase. The fault insertion phase instruments the kernel code and is semi-automated. Code instrumentation is achieved in two steps:

- Since all internal functions of the kernel are not relevant to our study, the first step consists in choosing the target functions according to the call graph generated for each kernel call. These functions are delimited by inserting comments at the beginning and at the end.
- The second step consists in inserting, before compilation, blocks of code, called saboteur, at the input point of an elementary function, as illustrated by the black dots in Figure 4.1.

We have developed an injection controller module, called **sabotage controller** in Figure 4.2, to enable faults within the kernel. Although several saboteurs can be inserted, only one is activated per experiment. Each insertion is associated with a flag. The set of flags introduced permits the sabotage controller to control the injections. The **injector** in Figure 4.2 enables the activation of a fault by issuing an ioctl() to the sabotage controller.

Other possibilities for injection can be supported in the future including the insertion of invalid values (similar to the API level injections).

It is worth noting that such an injection technique is intrusive and can only be applied if the source code is available. But this is not at all a problem in the type of controlled experiments we are conducting here. This kind of injection provides very accurate corruptions [Christmansson *et al.* 1998].

4.3.2.3 Error and Failure Observation

Figure 4.3 presents the observation framework. The observations are collected on the target and the host machines. The execution trace that contains the assertions and the detection modes are stored on the target machine. Whereas, the failure modes such as the kernel or the application hang are stored on the host machine. These two sets form the results.

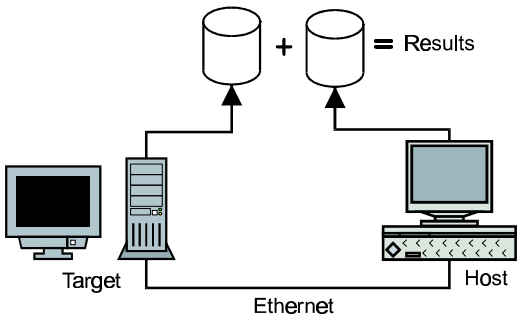


Figure 4.3: The observation framework

A module is inserted into the target kernel to retrieve the execution trace. The breakpoints along with their associated assertions are inserted within the kernel before compilation. The execution trace is saved on a file and analysed afterward.

We recall that the reported failures include returned error codes and exceptions. We have identified from the source code 20 error codes for the scheduling and the memory

components. These outcomes are deduced from the analysis of the application and the kernel execution traces.

The host machine notifies on the non-reported failures, i.e., the kernel and application hangs. The target machine signals to the host machine the beginning and the end of an experiment. After an experiment start, the host machine waits two minutes. If the target machine doesn't signal an experiment end within this interval, the host tries to establish a connection with the target. If it succeeds, we assume an application hang, and the host reboots the target automatically. Else, we assume a kernel hang, in which case the target needs to be rebooted manually. When the observed behaviour does not match any of the previous outcomes, a "no signaling" is reported.

4.4 Results

The conducted experiments targeted the scheduling and the memory components. We carried out 4470 experiments. Bit-flips in internal function parameters target 340 internal functions. These internal functions belong to the call graphs of six scheduling kernel calls and four memory kernel calls, target of the external injections. Table 4.3 summarises the experiment distribution per functional component.

The number of experiments when injecting invalid values is less than when injecting bit-flips. In fact, between 3 and 6 invalid values are associated to each kernel call parameter for the former injection technique. However, 32 bit-flips are issued to each kernel call parameter for the latter injection technique. That's why the injection of invalid parameters is eight times faster in retreiving all the results. However, it required more time for preparation.

Table 4.3 Experiment distribution per functional component

	Invalid argument	Bit-flip in API parameters	Bit-flip in internal function parameters
Scheduling	507	1890	552
Memory	101	1019	401
Total	608	2909	953

We present hereafter a top-down analysis of the obtained results. We first analyse the failure modes. Then, we refine these results by analysing the error codes returned. The next refinement concerns the enhanced monitoring allowed by the implemented specific assertions presented in Section 4.2.4.3.3. Finally, we show how such analyses concerning fault equivalence can be complemented by a fault representativeness study using the assertions based on real faults and presented in Section 4.2.4.3.2.

4.4.1 Analysis of the Failure Modes

In the following subsections we propose three kinds of analysis. The first analysis will permit to compare the provoked errors of the external fault models. Then we intend to compare the errors provoked by all the injected fault models. Finally, we present details about the influence of the target kernel functional component. The failure modes are given in Figure 4.4.

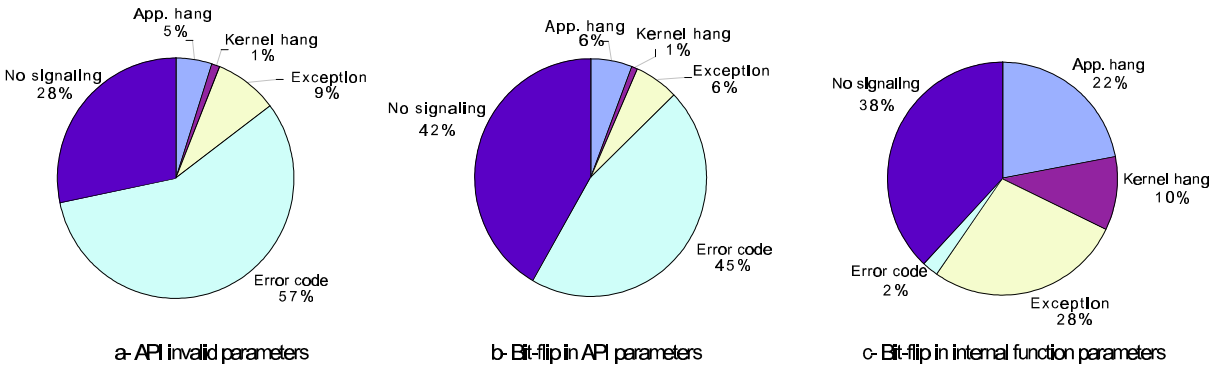


Figure 4.4: Failure mode distributions for the three injection techniques: global results

4.4.1.1 Comparison between external injected faults

The two external fault models provoke approximately the same failure modes in terms of nature and quantity. The dominant failure mode (Figure 4.4-a and Figure 4.4-b) is returning error codes (57% and 45% respectively). This shows the effectiveness of the checks implemented at the Linux kernel API level. A detailed analysis of the nature of the returned codes is presented in section 4.4.2.

API invalid parameters provoke less “no signaling” cases. Such an outcome is unusable and cannot be interpreted. However, it could lead to a failure in a different context than the one being considered in these experiments.

4.4.1.2 Comparison between all injected faults

We notice in Figure 4.4 the difference in the generated failure modes between the injections at the kernel API level (Figure 4.4-a and 4.4-b) and in its internal functions (Figure 4.4-c). The error code rate when injecting inside the kernel is low (2%). On the other hand, 28% of faults have been detected by hardware-generated exceptions, which means that 30% of faults have led to detected errors.

Let us analyse the reasons of the difference between the injections at the kernel API level and in its internal functions. Generally, the kernel calls in Linux consist in up calls to internal functions as illustrated in Figure 4.1 for `sched_setscheduler`. The latter calls one function (`setscheduler`), which fulfils the required service. One may assume that injections at the second depth level of this kind of kernel calls (`sched_setscheduler`, `gettimeofday` and `setitimer`) lead to the same error code. This is true for the `sched_setscheduler` kernel call where “Invalid Argument” and “Non existent process” error codes are generated even when injecting in the third level of the kernel function call graph. However, injections in the second level of the `setitimer` kernel call do not provoke “Bad Address” error code and provoke only an “Invalid argument” error code. This means that the error detection mechanisms for this function are implemented only at the first level. The analysis of the source code of the underlying function supports this statement. In fact only the value of the first parameter is checked in the underlying elementary function, which explains the presence of the “Invalid argument” error code alone in some experiments.

Figure 4.4-c shows that 10% of the injected faults in internal function parameters, leads to kernel hang, which is significantly higher than the 1% observed when injecting external faults. Also, 22% of the faults injected in internal function parameters lead to application hang, which is four times more than the observed percentage of external faults (5% and 6%).

4.4.1.3 Failure mode analysis according to functional components

Figure 4.5 presents the observed failure modes per functional component. The top three figures present the proportions associated with the scheduling component, and the bottom three are associated with the memory component.

External faults provoke less error codes for the memory component (39% for invalid parameters and 22% for bit-flips) than the scheduling component (66% for the two injection techniques). We also notice that when injecting external faults, "No signaling" is the dominant mode for the memory component (40% for invalid parameters and 60% for bit-flips). Another outstanding phenomenon is the presence of "Kernel hang" and "Exception" as outcomes in the memory component, which are absent for the scheduling component. This indicates that the error detection mechanisms implemented in scheduling kernel calls are more efficient than those implemented in memory kernel calls.

Internal faults provoke different behaviours between the two functional components too. We remark especially the rate of the "No signaling" mode, which is of 10% for the memory component and 61% for the scheduling component. Also the error detection rate ("Error code" and "Exception") is different: 50% for the memory component and only 14% for the scheduling. This shows the sensitivity of the memory component to the type of injected faults and its higher criticality compared to the scheduling component. The memory component is at a lower level and nearer the hardware.

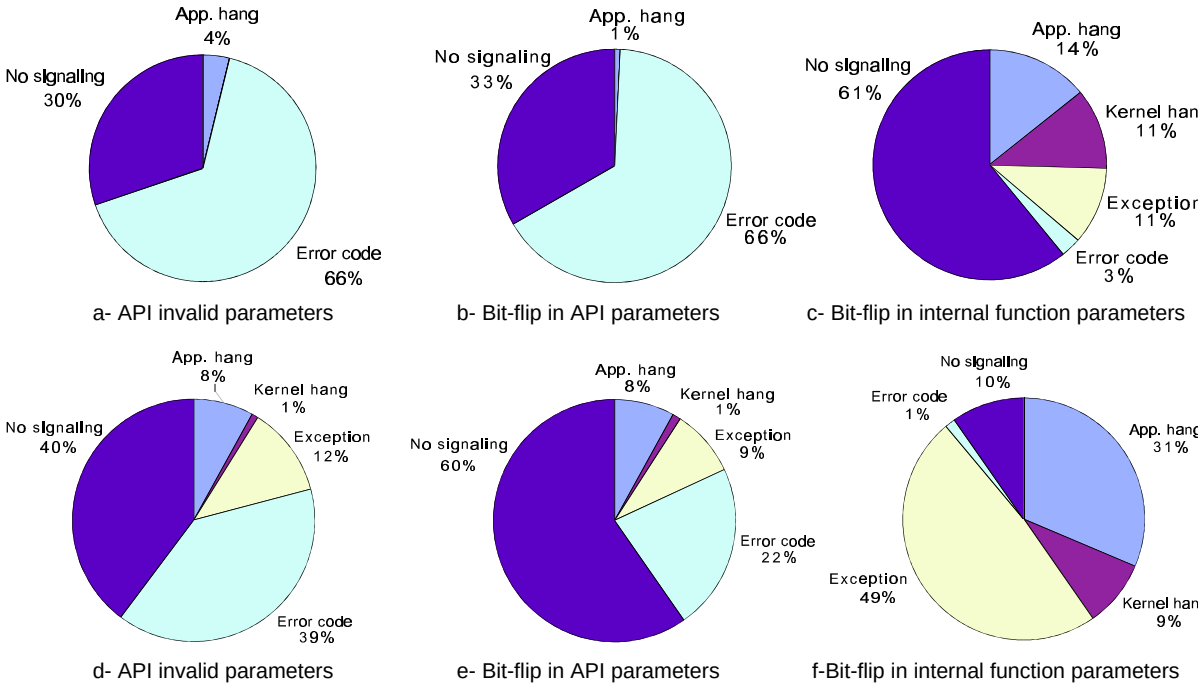


Figure 4.5: Failure mode rates of the scheduling and the memory components

All the raised exceptions in the memory and the scheduling components are Page Faults. This exception is raised when the addressed page is not present in memory, the corresponding page table entry is null, or a violation of the paging protection mechanism has occurred. It detects errors occurring within the memory. This is why their proportion is greater when injecting in the memory.

4.4.2 Comparative Analysis of the External Faults based on the Error Codes Returned

Based on the returned error codes, two kinds of analysis can be carried out. The first consists of the analysis of the nature of error codes, and the second consists of the analysis of whether other subsequent kernel calls return error codes.

4.4.2.1 Analysis of the nature of the error codes

Figures 4.4a and 4.4b show that the rate of error codes is greater when injecting invalid arguments than when injecting bit-flips. Figures 4.6-a and 4.6-b refine these results and show that generally**,** for a given kernel call, all error codes generated by the two injection techniques at API level are of the same nature. Yet, we notice a slight advantage for the bit-flip injection technique that provides more error codes than the invalid parameters. Also, the rate associated with each error code is not always equivalent, except for certain cases such as `wait4`.

The `mmap` kernel call provides a singular behaviour. As illustrated in Figure 4.6, five error codes were observed when injecting bit flips (“Out of memory”, “Invalid argument”, “No such device”, “Bad address” and “Bad file number”) while only one error code is returned when injecting invalid parameters (“Bad address”).

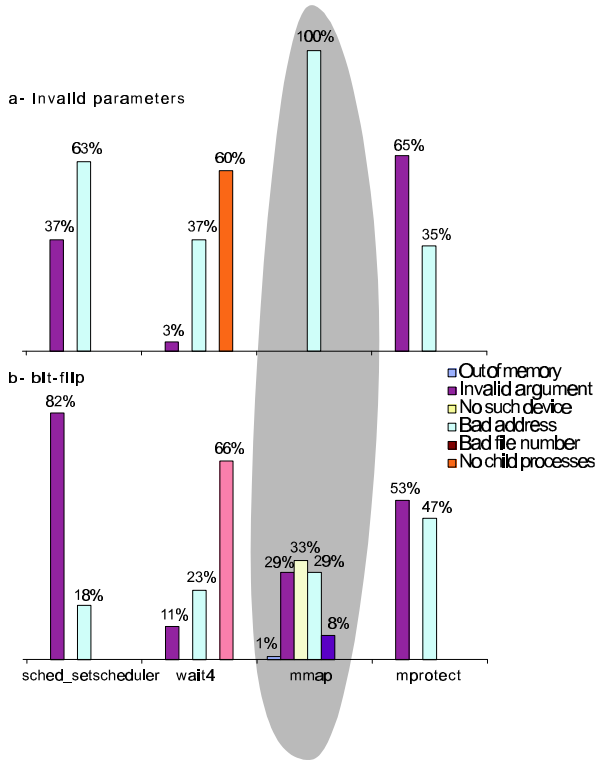


Figure 4.6: External fault analysis with respect to returned error codes

4.4.2.2 Error propagation analysis

When injecting within the kernel calls of the memory component issued by the memory workload, we observed error propagation to other functional components. We did not notice such propagation for the scheduling component.

Figure 4.7 illustrates error propagation rates when injecting faults in `brk`, `mprotect` and `munmap` kernel calls. The `mmap` kernel call does not provoke such propagation when injecting either invalid parameters or bit-flips. For each arc, the first percentage is associated with invalid argument technique and the second is associated to the bit-flip technique. We take into account all cases where an error code is observed. Error codes are returned either by `open` (file system component) or by both `open` and `wait4` (scheduling component). For example, corrupting the `brk` kernel call with invalid argument lead in 30% of the cases to `open` returning an error code and in 55% of the cases to both `open` and `wait4` returning error code. That does not mean that in 15% of the cases an error code is returned by `brk`. Figure 4.7 illustrates only the error propagation cases.

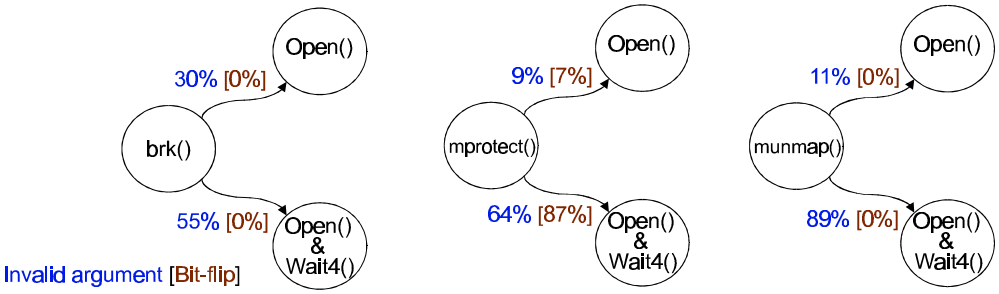


Figure 4.7: Error propagation: memory component

Injecting invalid parameters promotes error propagation for the three kernel calls among the four considered memory kernel calls. However, injecting bit-flips propagates errors only in the case of `mprotect`, with rates equivalent to those of invalid parameter technique (7% versus 9% propagate to `open` and 87% versus 64% propagate to both `open` and `wait4`).

4.4.3 Comparative Analysis Based on the Specific Assertions

We detail hereafter the observations made thanks to the implemented assertions introduced in Section 4.2.4.3.3. We recall that these assertions are implemented after the analysis of the source code. We are considering successively the two assertions that lead to relevant results.

The first assertion reports the `brk` kernel call activity. We have observed in normal operation (in the absence of faults) that the size of the memory data segment has not to be changed in 64% of the cases (`brk` does not carry out a specific treatment). Figure 4.8 illustrates the cases where we noticed a deviation from this normal invocation. Injected faults in the internal functions used by the `mmap` kernel call for example improve this rate by 0.5%. This is not the case for the bit-flip and the invalid parameter techniques at the API where the rate decrements respectively by 1.5% and 2.5%. This rate decreases by 7%, which is the worst case, when injecting faults in the parameters of the internal functions called by the `munmap` kernel call.

No such deviation has been observed for the scheduling component. There is no error propagation from the scheduling component to the memory component.

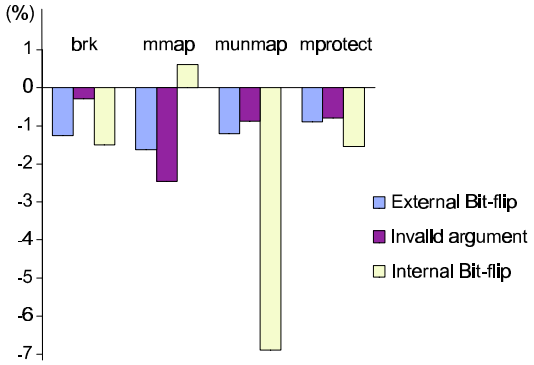


Figure 4.8: `brk` failure rate variations

The second assertion concerns the “memory pressure”. It represents the number of allocation requests that the kernel is trying to satisfy and thus the current memory load. The greater this value, the more the memory becomes a critical issue. Figure 4.9 presents the percentages of experiments where the memory pressure increases by more than 50% after a fault injection in the memory kernel calls. The average value of memory pressure is calculated before and after a fault injection. No such behaviour has been observed in the experiments targeting the scheduling component.

The influence of injecting faults in internal function parameters on the “memory pressure” is constant. For all the memory kernel calls, about 90% of the experiments lead to a memory pressure increase.

For the external faults, we remark an advantage when injecting invalid parameters since they tend to stress the kernel in more cases than the bit-flip technique. The cases where the bit-flip technique is equivalent to the invalid parameter technique is when targeting the `mprotect` and the `brk` kernel calls.

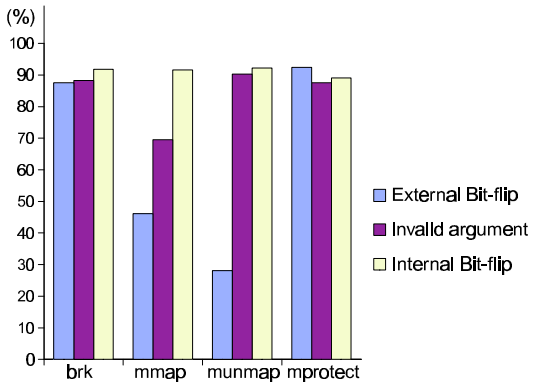


Figure 4.9: Memory pressure variation before and after fault injection

4.4.4 Comparison between Injected Faults and Real Faults

In this section we compare the consequences of the considered real faults (those revealed by the BLOCK and the NULL checkers) and of all the injected faults.

From Table 4.1, Figure 4.10-a shows that, if their provoked errors are not hidden 42% of real faults lead to “Kernel hang”, and 25% lead to “Exception”. But, as presented in Figure 4.4-a and Figure 4.4-b, only 1% of the external faults, lead to “Kernel hang” and only 9% and 6% lead to “Exception”.

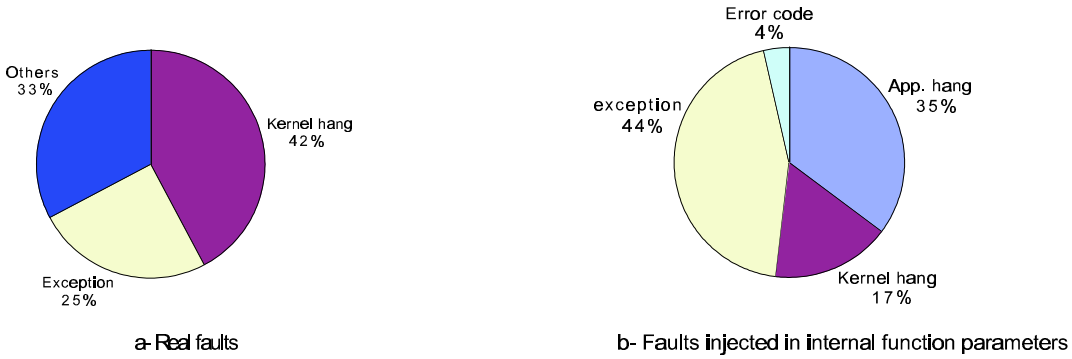


Figure 4.10: Comparison between real faults and faults injected in internal function parameters

The faults injected in internal function parameters are more or less representative of real faults. Figure 4.10-b presents the proportion of the observed failure modes without taking into account the “no signaling” mode. 17% (< 42%) of cases provoke “Kernel hang” and 44% (> 25%) provoke “Exception”. All we have to do is to adjust the distribution of the faults injected in the internal function parameters.

The goal of the implemented assertions is to observe the possible correlation between real errors (caused by real faults) and errors provoked by the three considered injection techniques. These assertions are designed to detect the targeted real errors. However, we were not able to activate these assertions with *any* of the three injection techniques. Consequently, none of the proposed injection techniques is able to reproduce the same real error.

This is due to the difference in the contexts in which the faults are activated. The real faults are revealed in the device drivers. Even though kernel calls and device drivers share some of the kernel internal functions, they are not activated in the same manner, i.e., the injected faults in the parameters of these internal functions are not activated in the same context. This is supported by a specific experiment in which we were able to activate a real fault based assertion. The experiment consisted in injecting faults in the parameters of internal functions called by the device driver functions. The considered workload inserted the network card device driver into the kernel. Although the activated assertion is designed to detect errors provoked by the faults revealed by the BLOCK checker, no kernel hang was observed. Accordingly, we can conclude that the error remains hidden.

4.5 Summary and conclusion

This work compares the impact of three types of SWIFI techniques on the Linux OS (version 2.4.0). Two of them target the kernel call parameters at the API (external faults) with two different fault models, namely: i) bit-flip corruption and ii) provision of invalid parameters. The third one applies bit-flips targeting the parameters of the internal functions of the kernel. We have developed an experimental framework that supports these three injection techniques and a comprehensive set of observations. The outcomes of the experiments include a wide range of failure modes either explicitly reported (e.g., exceptions) or not (e.g., hangs). In addition, specific assertions were implemented to provide a finer grain monitoring; in particular, some of them were deduced from the analysis of the effects caused by real faults.

The results presented in this paper refer to experiments focusing on the scheduling and memory components of the kernel. Depending on the target functional component, distinct failure modes are observed; in particular more exceptions are raised when targeting the memory component. Indeed, the scheduling component is at a higher abstraction level than the memory component, which is closely related to the mechanisms offered by the hardware. In addition, while we observed error propagation when injecting in the memory component, this was not the case for the scheduling component. Moreover, at the difference of the scheduling component, the faults injected into the memory component have a great influence on the level of stress of the kernel as revealed by the specific assertions. We can conclude that either the injected faults have not a great impact on the stability of the scheduling component, or more work is needed to develop improved assertions.

API-level fault injection is good candidate to assess kernel robustness. Flipping bits in kernel call parameters is easy to implement and does not need any *a priori* analysis of the parameter data types. However, it requires a lot of time, as it needs 32 injections per parameter for a 32-bit kernel and simple data types. Applying invalid parameters is eight times faster (for a complete campaign) compared to a bit-flip campaign, but it needs an *a priori* analysis of the kernel call parameters.

Although the provoked failure modes are comparable for both techniques independently from the functional component, the bit-flip injection technique provokes larger range of error code types than the invalid parameter injection technique. In particular, we have detailed the case of a kernel call (`mmap`) where out of the five error codes provoked by bit-flips only one could be provoked by the application of invalid parameters. Nevertheless, applying invalid parameters is proner to propagate faults than flipping bits, especially from the memory component to other kernel functional components as previously noted. Also, the proportion of experiments that lead to an increase of the memory pressure is more important when injecting invalid parameters.

Table 4.4 summarises the pro and cons for each technique. It shows that the invalid parameter technique provides more advantages than the bit-flip technique. In addition, it is worth noting that the *a priori* analysis could be done only once, as is the case for the Ballista-based POSIX test suite, which can be applied to all POSIX compliant systems.
Compared to the effects induced by external faults, flipping bits in internal function parameters provoked distinct erroneous behaviours. Indeed, many hardware exceptions were triggered by this technique, and the proportion of error codes observed was lower than for the

other two techniques. The implemented assertions exhibit further such behavioural differences.

Table 4.4 Comparative analysis between API-level fault injection techniques

	Experiment duration	Ease of application	Error codes provoked	Error propagation	Memory pressure	Significance of experiments
bit-flip	–	+	+	–	–	–
Invalid argument	+	–	–	+	+	+

Concerning the representativeness point of view, we have observed that external faults provoked very distinct behaviours compared to those induced by the real internal faults we considered (device driver faults). In particular, external faults were not able to activate the assertions based on real faults. This tends to indicate that it is unlikely that device driver faults could be easily emulated by injecting only at the API level, at least for the Linux kernel.

On the other hand, faults injected in internal function parameters were found, to some extent, representative of the considered real faults. Indeed, although the internal function parameters technique was not able to activate the assertions based on real faults, similar failure modes were provoked. This shows that the selected type (bit-flip) and location (interface) of the injected faults are not sufficient to provide a faultload matching the errors provoked by the considered real faults. Other fault types, such as invalid arguments and locations (e.g., assignation statements), need to be experimented. In addition, internal functions should be activated in the right way: from the driver interface, rather than from the kernel API. The experimental framework permits to easily implement those extensions.

The workloads used were selected to activate the kernel functional components in a typical way. The targeted kernel calls we have considered in this work are the most used in practice. However it would be interesting to target additional kernel calls for a each functional component.

Even though the two experimented kernel functional components are judged to be the most critical components, more work is needed to target the other kernel functional components.

5 Software Faults from Application (Language) Point of View

Software faults (also called software defects or bugs) have been consistently recognised as the major cause of computer outages. Several research studies show a clear predominance of software faults [Kalyanakrishnam *et al.* 1999, Lee & Iyer 1995, Sullivan & Chirallege 1992, Gray 1990] and given the huge complexity of today's software, the weight of software faults on overall system dependability will tend to increase. Regular readers of computer risks newsgroups, such as the ACM Forum on Risks to the Public in Computers and Related Systems [Newman 2001], can easily attest the relevance of software faults in many computer incidents.

The complete elimination of software defects during software development process is very difficult to attain in practice. In addition to well-known technical difficulties of the software development and testing process [Musa 1999, Lyu 1996], practical constraints such as the intense pressure to shrink time-to-market and cost of software contribute to the difficulties in assuring 100% defect free software. Therefore, the actual scenario in the computer industry is having systems in which software defects do exist but no one knows exactly where they are, when they will reveal themselves, and, above all, the possible consequences of the activation of the software faults. In a world where components of the shelf (COTS) are used more and more to build larger systems, the residual software faults represent a growing risk.

Many software reliability models and measurement procedures have been proposed for the prediction and estimation of measures of quality such as the number of faults remaining in a given software package. However, in addition to the difficulties in handling the extreme complexity of today's software and the limitations of the necessarily simple models with few parameters actually related to the project or software package at hand, the problem of these software quality measures is that they are mainly developer-oriented and do not give a measure of the possible impact of residual faults on the operation and on the end-user.

The use of fault injection to emulate the effects of real software faults has been recognised as potentially very useful [Voas *et al.* 1997, Christmansson & Chillarege 1996]. In practice, the injection of software faults consists of the introduction of small changes in the target program code, creating different versions of a program (each version has one injected software fault). The way faults are injected resembles the well-know mutation technique [DeMillo *et al.* 1988], but the injection of software faults has completely different goals. While mutation has been used for software testing to identify the best sets of test cases or to study the error propagation process, the injection of software faults in DBench is an important part of the faultload component of dependability benchmarks.

In general, the injection of software faults can be used as a complement to established software reliability engineering techniques [Kanoun *et al.* 1997, Musa 1999] and other traditional techniques meant to estimate measures of quality for software modules or products. Additionally, the injection of software faults is an effective way to validate fault-handling mechanisms and to evaluate the behaviour of a given system in the presence of the injected faults.

However, the injection of software faults has three main problems:

- Fault representativeness – The injected faults must be representative of real software faults or, at least, should cause similar erroneous behaviour.
- Intrusiveness – As the injection of software faults consists of small changes in the code, the injected module/program is not the same anymore, which may invalidate the observations made in the target system. The actual techniques used to inject the faults and to instrument the target program may cause additional intrusion.
- Lack of techniques and tools – In spite of the many fault injection tools proposed in the last two decades, very few proposals have addressed the injection of software faults (most of the tools emulate hardware faults) and in practice there are very few tools available for the injection of software faults. If we consider the specific context of DBench, the very nature of benchmarking implies the need of portable and easy to use techniques for the injection of software faults.

Although this deliverable is specifically devoted to fault representativeness, the fact that the problems mentioned above are intimately related one to each other means that is virtually impossible to separate representativeness from the other aspects. Thus, we propose to address these fundamental problems in the following way:

- Use of **educated mutations** to improve the representativeness of the injected faults as much as possible. The idea is to define a subset of selective faults based on available field data on bug reports and common programming language pitfalls. In this way, injected faults are not arbitrary mutations of the code but they do correspond to the most common programming mistakes.
- Use of a **new technique to emulate software faults** by selective mutations introduced at the machine-code level.
- Propose an operating scenario for the injection of software faults in which **faults are injected in one module to evaluate the behaviour of the rest of the system**. This solves the problem of intrusiveness, as the injected module is not under evaluation (see Figure 5.1). Some examples of concrete use of this scenario are: a) injection of faults in a device driver to evaluate the way the operating system (OS) behaves in the presence of a “mad” driver; b) injection of faults in the OS to evaluate a fault-tolerance layer at the application level; c) injection of faults in an application process (emulating a buggy application) to evaluate the probability of error propagation to other processes; etc, etc.

The technique proposed for the injection of software faults is called Generic Software Fault Injection Technique (G-SWFIT) and consists of finding key programming structures at the machine code-level where high-level software faults can be emulated. This way, it is possible to have a library of machine-code level structures and possible software faults that once introduced in such programming structures can emulate specific classes of high-level software faults. The accuracy of the injected faults and the generalisation of the proposed method (concerning high-level languages, compilers, compiler optimisation options, processor architecture) has been carefully studied and the available results suggest that the technique is very accurate and can be easily ported to practically all type of systems (we mainly need to define specific parts of the fault library to accommodate the emulation of faults in a new kind

of system). Finally, as the proposed technique works at the machine-code level, it is not required to have the source code of the target program, which makes it possible to apply the proposed technique to virtually any program (particularly relevant for the evaluation of COTS components or COTS based systems).

Although G-SWFIT has been specifically conceived for dependability benchmarking, we believe that this technique can be used in several other scenarios. The following points summarise the possible uses of G-SWFIT (obviously, the technique will be used in DBench for dependability benchmarking only):

- Dependability benchmarking. This is the central goal of G-SWFIT. Dependability benchmarking faultloads require generic, portable, and easy to use fault injection techniques and these are exactly the target features behind G-SWFIT design. In fact, the technique can be applied to any software module, even to software modules for which the source code is not available, and the technique does not require a complex fault injection tool to be used.
- Evaluation and validation of software fault-tolerance mechanisms. Knowing the difficulties in achieving 100% defect free software, it is vital to assure that software products can deal with residual faults in an acceptable way. This is particularly important for mission and business critical systems, as the required software fault tolerance techniques must be carefully tested and evaluated. G-SWFIT can be used as a generic technique to inject software faults for the evaluation and validation of software fault-tolerance mechanisms.
- Evaluation of the behaviour of software products in presence of faults and validation of wrappers. Even for application areas where the use of software fault-tolerance techniques are not indispensable, it is important to have means to evaluate the way software products behave in the presence of faults, either faults in their own code and faults in 3rd party components, in order to improve mechanisms used to wrap the effect of the faults and mechanisms meant to handle faults in an acceptable way (e.g., warning the user and terminating the application in a safe way instead of crashing as a consequence of a fault). From the application developer/integrator perspective this means protection against faults in 3rd party libraries, shrink-wrapped components, or even faulty operating systems.
- Software risk assessment and prediction of worst-case scenarios. The emulation of software faults can be used, as proposed in [Voas *et al.* 1997], to optimise the testing phase effort by performing risk assessment and prediction of worst-case scenarios. This way, it is possible to quantify the impact of software faults from the user point of view and get a quantitative idea of the potential risk represented by residual faults. G-SWFIT allows performing this evaluation in COTS software components, even when the source code is not available.

5.1 Software Faults Emulation at Low-Level Executable Code

G-SWFIT technique consists of modifying the ready-to-run binary code of software modules by introducing specific changes that correspond to the code that would have been generated

by the compiler if the software faults were in the high-level source code[6]. A library of mutations previously defined for the target platform guides the injection of code changes: the target application code is scanned for specific low-level instruction patterns and selective mutations are performed on those patterns to emulate related high-level faults. Depending on the size of the low-level mutation library and the settings specified by the user, a number of versions of the original target application are created, each one containing an emulated high-level software fault. Figure 5.1 depicts the key elements of this technique.. This technique was presented and its accuracy evaluated in [Durães & Madeira 2002]. Although not strictly necessary, a commercial disassembler can be used to produce assembly listing to assist guidance of the fault injection process.

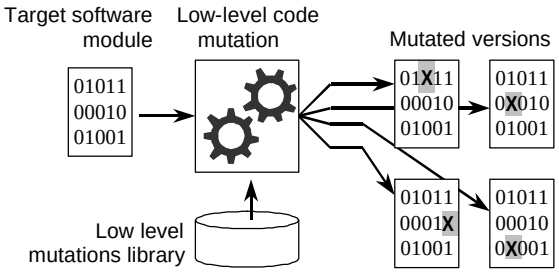


Figure 5.1 - Software fault emulation process

The definition of the low-level mutation library is based on two principles: the existence of a set of simple high-level programming errors that occur frequently and the knowledge of how high-level languages are translated into low-level code, in particular how high-level programming errors translate into specific low-level instruction sequences.

Many of the defects that remain in software modules after deployment are usually simple programming errors (at least if analysed independently from their context) or wrong usage of language constructs. Their complexity arises from the code that contains them [Madeira *et al.* 2000, Daran & Thévenod-Fosse 1996, Carter 2001a, Cramon 2001]. Such defects are classifiable into a few high-level fault classes, making possible a definition of a set of common errors. In order to accurately inject, at machine language level, faults that represent real high-level software defects, it becomes necessary to know in detail to what high-level constructs they do correspond to.

Orthogonal Defect Classification (ODC) is based on software faults found in real programs, and classifies software defects in a set of non-overlapping classes (the classification is based on the way faults have been corrected) [Chillarege 1995]. ODC classes of faults will be used as starting point, but as each class includes a very large number of possible faults a more detailed description is necessary.

Since the main purposes of this section is to show that high-level software faults can be emulated with a reasonably accuracy and that the technique to do that is feasible, a pragmatic approach was followed to identify the types of high-level software faults to emulate. We have

[6] Of course, it would be easier to apply the technique directly to the source code and recompile mutated versions. However, as the source code is not always available, we decided to target the technique to the generic case: the one in which is source code is not available.

analysed several sources, and built a list of possible software bugs that can reasonably be expected to occur frequently. We call these faults “educated mutations” to emphasise that they include inputs from experience and from field data on real software faults.

To assist definition of the fault library we used a synthetic application containing all the pertinent key constructs and structures, both with and without the considered high-level faults. The observation of the generated code for both cases (with and without faults) allow us to identify the specific machine code patterns where a given class of faults should be emulated by low-level mutations. Using this application with several compilers and different optimisation settings, we could understand how such variations affect low-level code generation and identify patterns for a given class of faults that remain constant. Additionally, existing coding standards are also considered to further refine the low-level instruction patterns related mutations. Those steps are depicted in Figure 5.2.

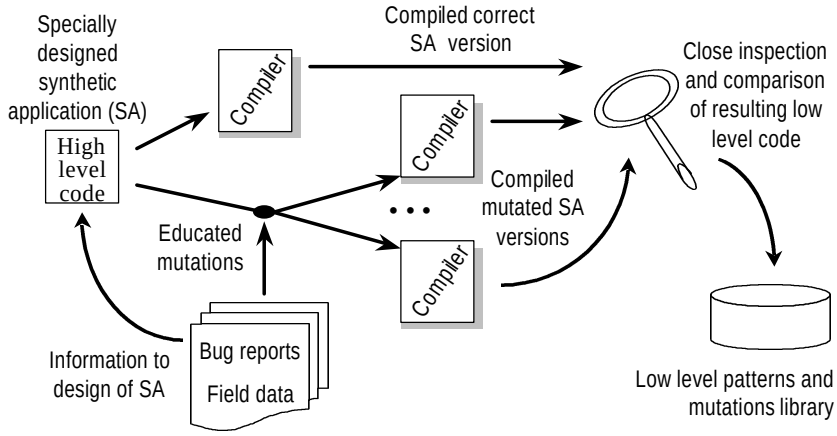


Figure 5.2 - Machine language instructions patterns and mutations definition

5.2 Fault Representativeness Evaluation

The accuracy of the proposed methodology for the emulation of software faults was evaluated through a case study, defined to represent the common dependability benchmarking scenario addressed in DBench. The following issues are relevant for the considered scenario:

- High-level language assumed for the source code: the language must be representative of the languages used for COTS software.
- Faultload definition: the faultload must emulate a set of high-level errors that are representative of common programming errors.
- Generalisation: the technique used for fault injection and its results should be portable to other platforms.

C programming language was selected for the case study to evaluate both the feasibility and accuracy of the emulated faults. This selection is based on two main reasons: a) the C language is one of the most widely used, therefore significant in the COTS scenario, and b) The C language has many features similar to other languages also commonly used (e.g., C++,

JAVA). However, the methodology described here is in no manner tied to the language itself, and can be easily ported to other languages (this issue is discussed in section 5.3.3).

The study of the G-SWFIT technique representativeness was carried out in three steps:

1) The set of high-level language software faults were identified. This step also comprised the elaboration of high-level faulty versions of the target applications through educated mutations.
2) The low-level (i.e., machine-language) instruction patterns, where mutations can be inserted for emulation of the high-level faults, were defined. The related mutations were also defined in this step.
3) Fault emulation was performed through the injection of mutations defined in the prior step into the low-level code. A number (from tens to thousands) of mutants were generated from the original executable version. The execution of each mutant with a representative input profile is a fault injection experiment in our context. The effects of the high-level language software faults and the equivalent low-level faults emulated by G-SWFIT were carefully compared.

Sections 5.2.1 and 5.2.2 present the details of the first two steps. In Section 5.2.3 the experimental environment is described, including the target applications used. The results are presented and discussed in section 5.2.4.

5.2.1 High-Level Software Faults Set Identification (Educated Mutations)

The characteristics of C programming language were analysed to obtain a set of errors that are prone to be made in this particular language, obtaining a high-level fault characterisation related to ODC but having a more detailed characterisation in what concerns its relationship with the syntactic rules of the language behind the faults. An extensive research on most common C programming bugs has been carried out, using various information sources ranging from programming manuals and best practice tutorials [Koenig 1989] to error reports [Carter 2001a, Cramon 2001]. Reference manuals of a tool specifically designed to assist programmers to avoid errors were also used to identification of the high-level fault set [Gimpel 2001].

Since the goal is the emulation of high-level faults that typically remain in software products after deployment, the identification of such faults is additionally guided by the following rules:

a) Faults must be relatively common (i.e., likely to appear in available software products).
b) Faults must not generate syntactic errors.
c) Faults must not be too obvious.

The first rule was enforced with the help of available bug reports and best practice tutorials mentioned earlier. Faults that generate syntactic errors have been excluded, because such faults cannot be compiled into executable code. At most, faults are allowed to generate warning messages, as this is very dependent on the compiler. Some compilers are smarter than others, raising the possibility of some errors being left undetected by less smarter

compilers (e.g., a function with an execution path that uses a variable not initialised is a situation that some compilers do not always detect). Finally, the high-level programming faults considered should not be so obvious that only novice programmers could/would make them.

The considered high-level faults were characterised according to the following items:

- Applicable ODC class: the characteristic relates the mentioned errors with the ODC classes. Some errors fit in more than one OCD class, depending on the way that those errors would be corrected.
- Example of possible cause: this helps to understand how the error may appear in the source code. This is a fundamental characteristic of the considered errors, as this work analyses the faults mainly why do faults appear instead by how they can be corrected. The possible causes are very important to identify all the different incarnations of each fault (this will be detailed further on).
- Compiler ability to detect the error: the ability of available (common) compilers to detect and warn the programmer of the possible existence of that error. For this evaluation, three different compilers were used (VC++, BorlandC and GCC). This characteristic is important, as it is directly related to the probability of leaving the error uncorrected in the program after deployment.
- Language specific degree: this characteristic relates each particular error with the possibility of that error appearing in all common languages, or instead, if it is something specific to one or few languages. This characteristic gives some insight on the generalisation of the technique to encompass common faults in other programming languages (generalisation issues will be detailed in Section 5.3).

The resulting set of high-level common errors in C programming language is presented in Table 5.1. Due to its nature, not all faults presented in Table 5.1 can be emulated with the same degree of accuracy. For example, the fault “missing statement due to non-terminated comment” is something that can never be accurately emulated with the same accuracy degree of the “assignment instead of equality comparison”. For the latter, locations inside the low-level code where an equality comparison exists can be identified, however there is no way, without the source code, to know where a comment originally was. This makes this particular type of fault fall outside the scope of the G-SWFIT technique. Others faults, although possible of emulation through this technique, will have lower emulation accuracy. For example, the locations suitable for emulation of the fault “wrong usage of parameter call” can be easily identified inside the low-level code, however, since the type of data of the parameter is lost in the low-level language, the majority of low-level mutations would correspond to high-level faults causing syntactic errors when compiling.

The types of high-level faults emulated in the test-case are presented in Table 5.2. Some of the previously presented high-level faults were grouped into the same type since they can be emulated trough the same mutations applied to the same machine-code instruction patterns. It is worth noting that a total of four different ODC classes are covered (note that ODC has six classes directly related to the code), and these four classes correspond to 79.2% of faults found in the field according to data form the actual use of ODC at IBM [Christmansson & Chillarege 1996].

Table 5.1 – High-level fault case study for C language

ODC class	Fault type description	Example of possible causes	Examples of educated mutations	ODC also	Compiler ability to detect	Language dependency degree
Assignment	Missing or wrong local variable initialisation	Initialisation dependent of execution path	Omission of first assignment to local variable		Some	Low
Checking	Assignment where comparison was intended	If (a = b) instead of if (a = = b)	Change expressions "==" to assignment	Alg.	Some / High	High
	Miss by one in decisions	Lack of attention;	*If (a < b)* instead of *if (a <= b)*	Assig.	None	None
	One iteration too few or too many	Lack of attention; Misunderstanding of variable meaning	*while (a <= b)* instead *while (a < b)*	Assig.	None	None / low
Interface	Missing return statement	Non-trivial execution paths inside functions	Omit return statement were syntactically possible		Some	None
	Wrong *return* statement	*return var* with *var* not properly set	Omit previous assignment to variable used with *return*.		Low / Some	None
	Wrong usage of parameters in function call	*Func(x,y)* instead of *func(y,x)* and *x* and *y* with compatible data types	Change order of values in the function call (only if compatible data type)		None	None
Algorithm	Miss aligned *else*	Bad indenting practice plus single statements mixed with multiple-statement portions of code inside the if or else	Omission of curled brackets inside complex *if-else* chains	Alg.	None / very low	Some / low
	Binary operator usage where logical operators were intended	Typo; Lack of attention	Change of *a && b* to *a & b*	Check. Assign.	Some (low)	High
	Wrong logical expression due to wrong usage of operator precedence	Usage of complex expressions Example: *(a == b && c == d || d == e)*	Insertion, omission or change of parenthesis in complex expressions		None / very low	High / Low
	Missing statement	Non-terminated comment (terminated one line latter, for instance) A/*p (p is a pointer)	Omission of one statement following a comment	(Assign)	None	None / Low
	Missing function call	Lack of attention	Removal of the function call if that function appears by itself, i.e., its not part of a bigger statement	Alg.	None	None

Table 5.2 - High-level faults emulated in the case study

Type of high-level fault emulated in the test-case	Encompassed faults:	ODC applicable class
Assignment instead of equality comparison	Assignment where comparison was intended	Algorithm, Checking
Miss by one	One iteration too many or too few	***Checking, Assignment, Algorithm***
	Miss-by-one comparison error	
Missing or wrong return statement	Missing return statement	Interface, (Assignment)
	Returning a variable not properly set	
Missing local variable initialisation	Missing local variable initialisation	Assignment
Missing function call	Missing function call	Algorithm

5.2.2 Low-Level Instruction Patterns and Mutations

As mentioned in 5.1, a synthetic program containing all the relevant high-level code constructs was used to assist the task of identifying the machine-code patterns and the corresponding low-level mutations. Additionally, the low-level instructions patterns have

been enhanced with the observations from standards on C compilation [Carter 2001b] (e.g., the passing of parameters to functions in strictly regulated by standards).

The low-level instruction patterns must be generic enough to match the different ways that the same construct can be translated into low-level code, and specific enough to left out (not matching) other constructs that have similar patterns but do not match the intended high-level fault. For most of the identified high-level bugs it was possible to find patterns that satisfy this condition. However, some patterns also match some other high-level instructions or constructs that are not related to the intended high-level fault. This leads to the possibility that some injected mutations at the low-level code do not correspond totally to the intended high-level error.

5.2.2.1 Assignment instead of an equality comparison

"Assignment statements instead of an equality comparison" can take several forms in the high-level source code: a comparison inside an `if` construct or a comparison inside a `while` or `do-while` condition; in both cases, a comparison can be made between two variables or between a variable and a value. Therefore, several patterns in low-level language are associated with this bug. All of them can be seen in Table 5.3, as well as their meaning and related mutation.

Table 5.3 Assignment instead of comparison patterns

Description	Search pattern	Constr	Mutation
Comparison of a variable to another variable			

if (var1 = = var2) { }	MOV *reg, mem1*
	CMP *reg, mem2*
	JNE *ahead*
		C1	MOV *reg, mem2*
			MOV *mem1, reg*
			CMP *reg*, 0
			JE *ahed*

While (var1 = = var2) { }	MOV *reg, mem1*
	CMP *reg, mem2*
	JE *back*
		C1	MOV *mem1, reg*
			CMP *reg*, 0
			JNE *back*

do { } while (var1 = = var2)			
for (…; var1 == var2; …) { }			
Comparison of a variable to a value			

if (var = = value) { }	CMP *mem, imed*
	JNE *ahead*
		C2	MOV *mem, imed*
			CMP *mem*, 0
			JE *ahed*

While (var = = value) { }	CMP *mem, imed*
	JE *back*
		C2	MOV *mem, imed*
			CMP *mem*,0
			JNE *back*

do { } while (var1 = = value)			
for (…; var1 == value; …) {}			

Unfortunately the patterns used to find equality comparisons between a variable and a value also match comparisons between *non-lvalues* (*rvalues*) and variables. Although highly uncommon, such things as "*if (4 == a)*" can occur. No pattern was found that could match "*lvalue = = rvalue*" without matching "rvalue = = *lvalue*", therefore, some deviations are expected from a perfect match in the accuracy evaluation results. Another form of use of a *rvalue* at the left of a comparison is when the result of a function call is used at the left side of the comparison operator. To avoid matching this specific situation, the patterns must verify two constrains, shown in Table 5.4.

Table 5.4 Assignment patterns constraints

Assignment instead of comparison pattern constraints			
Constraint	**Description**		
C1	Pattern must not be preceded by:		

	CALL *address*
	ADD ESP, *value*
	MOV *mem1*, EAX
		nor	CALL *address*
			MOV *mem1*, EAX

C2	Pattern must not be preceded by:		

	CALL *address*
	ADD ESP, *value*
	MOV *mem*, EAX
		nor	CALL *address*
			MOV *mem*, EAX

5.2.2.2 Missing or wrong return statement

"Missing or wrong return" errors can appear in different forms: a `return` statement may be left out in the source code by mistake (e.g., when a function have several execution paths, one of them without the explicit `return` expression) or using an auxiliary variable to maintain the value that will be returned and there is one possible way (execution path) of that variable not being properly set. The patterns used to match suitable locations, shown in Table 5.5, depend on the identification of the starting and ending points of a function. Fortunately, that is a relatively easy task, since the beginning and ending points of a function or procedure translate to very specific patterns of code, as can be seen in Table 5.6.

Table 5.5 Missing or wrong return statement patterns

Description	Search pattern	Mutation
return *expression* (other than just a value)	MOV EAX, *mem* *zero or more instructions except:* MOV EAX, … *and* XOR EAX, EAX EAX *modifying instruction* *stack cleanup and return*	All instructions are omitted except the stack frame cleanup and return
return *value*	MOV EAX, *value* *Zero or more instructions except:* MOV EAX, … *and* XOR EAX, EAX EAX *modifying instruction* *Stack cleanup and return*	All instructions are omitted except the stack frame cleanup and return
return 0 (typical value)	XOR EAX, EAX *Stack cleanup and return*	XOR EAX, EAX removed
return *not-properly--set-variable* (usage of a variable in the return which was not correctly set)	MOV EAX, *mem* *Zero or more instructions except:* MOV EAX, … *and* XOR EAX, EAX EAX *modifying instruction* *Stack cleanup and return*	MOV *mem*, … instructions in the same function omitted (each one counts as a different fault)

Table 5.6 Function start and end patterns

	Machine code sequence	Meaning
Function start	PUSH EBP MOV EBP, ESP	Stack frame initialisation
Function end	MOV ESP, EBP POP EBP RET	Stack frame cleanup and return

It is quite clear at this stage that some patterns are not trivial, which means that conventional fault injectors would have problems in inserting the low-level code modification. The approach proposed in the G-SWFIT does not have any special difficulty regarding pattern complexity since mutations are done prior to execution of the target application, when complex pre-processing is possible without causing intrusion to the target application.

5.2.2.3 Miss-by-one boundary checking

The low-level instruction patterns generated by the compiler needed to identify "miss-by-one" type faults are relatively unique and easy to identify, as can be seen in Table 5.7. Mainly, the patterns consist of conditional jump instruction. The mutations themselves are jump instructions (to the same target address) but with the "equal" condition reversed, i.e., if the condition is *jump if greater(less)*, then it will be mutated to *jump if greater(less) or equal*; if the instruction is *jump if greater(less) or equal*, then the mutated instruction will be *jump if greater(less)*. All patterns and related mutations presented apply to high-level expressions involving both variables and literals.

Table 5.7 Miss-by-one patterns

Description + example	Search pattern	Mutation
Greater/Above test		
Original expression: …> … Mutated expression: … >= …	JG *address*	JGE *address*
	JA *address*	JAE *address*
Greater/Above or equal test		
Original expression: …>= … Mutated expression: … > …	JGE *address*	JG *address*
	JAE *address*	JA *address*
Less/Below test		
Original expression: …< … Mutated expression … <= …	JL *address*	JLE *address*
	JB *address*	JBE *address*
Less/Below or equal test		
Original expression: …<= … Mutated expression: … < …	JLE *address*	JL *address*
	JBE *address*	JBE *address*

5.2.2.4 Missing or wrong local variable initialisation

The patterns used to identify locations suitable for the emulation of faults related to missing or wrong initialisation in local variable just find the first occurrence of an assignment to a location inside the stack space in each function. These patterns are shown in Table 5.8. To work properly, the identification of function starting and ending points is required, which is done using the patterns presented in Table 5.6.

The pattern for wrong initialisation is basically the same, with the difference that the mutation introduced is different (assignment of incorrect value, or variable, or expression). As expected, some of the patterns presented can be used to emulate different high-level faults, although they are quite similar. This fact helps maintaining the set of low-level patterns and mutations in a manageable size.

Table 5.8 Missing local variable initialisation pattern and mutation

Description	Search pattern	Mutation
First occurrence of value assignment to a location in the stack memory space (in a given procedure code)		
local_var = value	First occurrence of MOV *offset*[EBP], … for each given *offset*	MOV instruction removed
local_var = some_var		
local_var = expression		

5.2.2.5 Missing function call

The patterns used to identify locations suitable for the emulation of faults related to a missing function call are shown in Table 5.9. Such patterns and its related mutations are those on. In order to avoid removing a function call which return value is used, i.e., the call itself is located inside an expression, the eligible locations for fault emulation must comply to the constrains shown in Table 5.10. The rationale for avoiding removing functions calls located inside expressions is that such high-level mistakes are more difficult to make, as the compiler can easily detect that the resulting expression is incomplete. On the other hand, the simple omission of a function call that appears by itself in a statement is something that can hardly be detected by any compiler.

Table 5.9 Missing function call pattern and mutation

Description	Search pattern	Mutation
Occurrence of a function call by itself (i.e., the whole statement where it appears is the call itself)		
Correct: some*_function(…..)* Bug: function call omitted	CALL *target-address*	CALL instruction removed

Table 5.10 Missing function call pattern constraints

Missing function call pattern constraints		
Description	Rationale	Examples of high-level code being avoided

Pattern must not be followed by any usage of the value of the AX/EAX register before another *call*, *jump*, *ret* instruction or assignment of a value to that register	Usage of AX/EAX register in that manner following a function call would mean that the return value of that called function is being used	var = function();
		if (function() == ….) {}
		var1 = var2 + function();

One important aspect is the use of macros in the high-level source code, for such macros may expand to a sequence of instructions without any function call whatsoever. This may cause a slight difference in the observed behaviour of target applications regarding executions of high-level faults and low-level faults.

5.2.3 Experimental Set-Up for Evaluation of the Accuracy of the Proposed Technique

To evaluate the accuracy of the emulation of the high-level faults we compared the effects (program failure modes) of high-level educated mutants with the ones obtained with low-level mutations inserted using G-SWFIT. The first step was, of course, to build the library with

low-level patterns and the corresponding high-level faults, using the strategy explained in previous sections.

The correctness of the outputs generated in each run (when the program terminates and generates results) was evaluated using an oracle that knows the correct results for all the input vectors used in the experiments. The list of correct results has been obtained by running each application with no faults for all the input vectors. In addition to the correctness of the results, other aspects of the application behaviour have been used, leading to the following failure modes:

- Correct behaviour ("***correct***"): the application produced the expected result in the allotted time and did not cause any abnormal event during its execution. Either the injected fault was not activated or it was tolerated by the inherent program redundancy. To minimise the possibility of the injected fault not being activated, a large number of different inputs were used with each mutated version.
- Uneventful execution but with incorrect results ("***error***"): the application terminated but the results were incorrect.
- Erratic behaviour ("***erratic***"): the application behaved in an unpredicted manner. The injected fault was activated and produced consequences that were not within the normal parameters of execution (for instance, the application produced an incoherent error message, that was not either an error message or success message), which is of no use to the surrounding environment. Thus, there is no feedback to enabling the determination of the success of the requested task.
- The application hanged ("***timeout***"): the situation is detected by allowing a *more-than-enough* time interval for its completion. If the application does no terminate within that interval, then it is assumed that it hanged and is terminated by an automated tool.

The target applications were chosen according to the following three rules:

a) The application should not be too simple, in order to be representative of commonly used applications.

b) The selected applications must produce a deterministic output from a specific input, to enable the automation of the equivalence testing procedure.

c) Source code should be available. Although this is not necessary for the actual use of the proposed technique (on the contrary, our main goal is to avoid the need of the source code to emulate software faults) the evaluation of the accuracy presented in the paper needs the source code to compare the effects of low-level mutations with known high-level faults.

The selected target applications are the following:

- **Lzari**: This application compresses a file using an arithmetic compression algorithm, producing two new files, being the first the compressed version of the original file and the second the result of the decompression in order to obtain the original file again. In order to fully exercise the program algorithm a variety of input files have been chosen including both hard to compress data (such as jpeg files) and easy to compress (such as text files).

- **Camelot**: This application produces the minimum number of moves necessary to gather a king and one to 63 knights in the same position of a chessboard. The usage of this application provides the additional advantage of having a number of versions each containing a real programming error, which have been used to help the definition of the high-level faults [Madeira *et al.* 2000].
- **Gzip**: This application is a widely used compression tool available in most Unix boxes. A port for the win32 environment was used in this work. As Gzip is a real-world commonly used application, its inclusion in this work provides a good degree of confidence of the deployment of this technique to COTS components. The fault injection was performed over the compression modules.

The host machine is an Intel Pentium III machine running Windows2000. Low-level fault injection was carried out with the aid of tools written in perl language. The injection itself is done into the assembly language file, which is produced by the compiler. The decision to inject into the assembly language instead of the executable file was made since it is in the context of an experimental work. The final injection tool for practical low-level fault injection works directly in the binary code.

A specific tool was implemented to launch the target applications and collect information about the way it terminates and if it terminates at all. This tool launches the target application and waits for its termination. If the timeout period has expired and the target application did not terminated, then it is forcibly terminated. When the application terminates normally, the return code is collected.

5.2.4 Results and Discussion

Table 11 summarises the experiments performed. For each fault and fault level (i.e. original high-level faults and equivalent low-level faults injected by G-SWFIT), a different executable file was produced and run against all the inputs for that application (70 for Lzari and Gzip, 120 for Camelot). Figures 5.3 through 5.7 show the behaviour observed after each run. From a global point of view, the behaviour of the target applications was mainly the same for both high-level faults and its emulated low-level counterparts, which suggests that educated mutations at the source code level can be emulated by G-SWFIT at the machine code level with a good precision. The following paragraphs discuss the results in more detail.

Table 5.11 Injected faults and executed runs

Fault		**Application (# faults)**		
Type	**Level**	**Lzari**	**Camelot**	**Gzip**
Assignment where equality comparison was intended	High	9	7	17
	Low	12	8	20
Miss by one in inequality checking	High	51	29	34
	Low	61	32	33
Missing return or return variable not initialised	High	5	8	3
	Low	5	8	3
Local variable not properly initialised	High	16	1	6
	Low	15	1	8
Missing function call	High	33	17	14
	Low	29	17	10
Total runs		16520	15360	10360

Considering “missing or wrong return statement” faults (Figure 5.5), all applications revealed a perfect match between the behaviour of the high-level faulty executables and the low-level faulty ones. This suggests that the patterns and mutations used to emulate the missing or wrong return statement can indeed emulate this fault with very good accuracy.

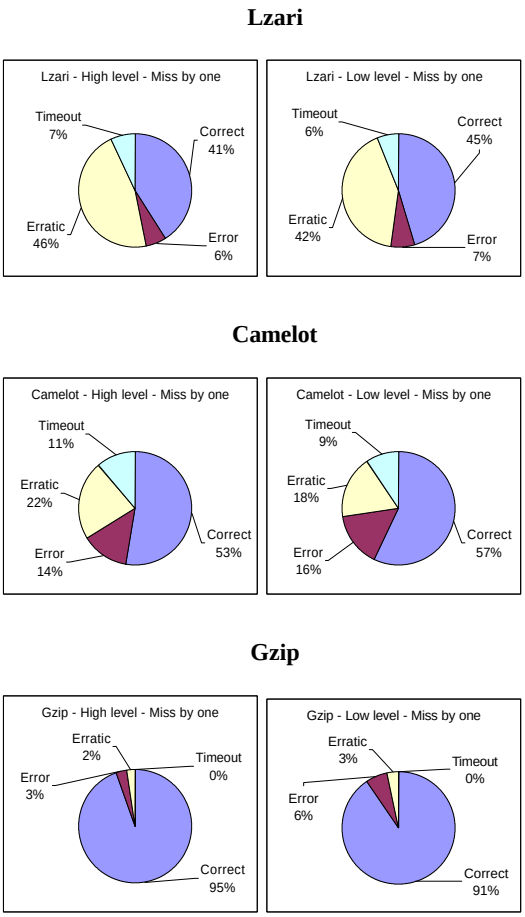


Figure 5.3 – Results for miss-by-one boundary check.

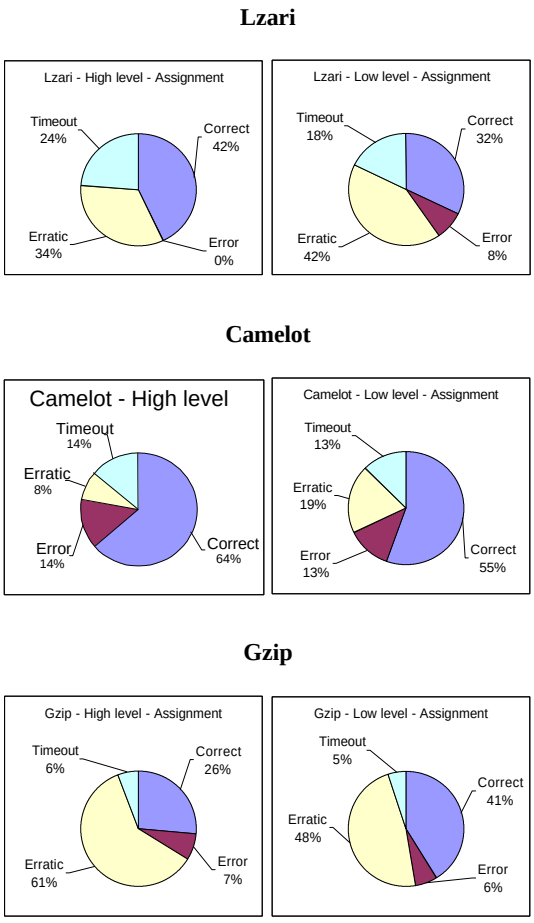


Figure 5.4 – Results for assignment instead of equality comparison

Regarding “missing function call” faults (Figure 5.7) and the “miss-by-one” faults (Figure 5.3), a very good correspondence was observed for all applications. Minor differences were noted however, especially for Gzip. This small difference of behaviour is explained by the use of macros within the high-level source code. At high-level, such macros resemble function calls, but the corresponding machine-code instruction sequence does not necessarily contain a CALL instruction. This has the consequence of some high-level missing function call not being emulated at low-level. Also, because the code corresponding to the macro definition is replicated whenever such macro is used, if the macro definition contains a fault location, then a single high-level fault location will translate to several locations within the low-level code. This is indeed the cause for the difference of behaviour observed for the Lzari application regarding the “miss by one” fault. Due to the small differences observed in the behaviour, it can be said that both missing function calls and “miss by one” faults can be accurately emulated by the patterns and mutations used in the case study. It is worth noting that there is a

large percentage of correct results (especially regarding Gzip), which suggests that these faults could be remain undetected, as the application worked correctly for many test cases.

Regarding “assignment instead of equality comparison” faults (Figure 5.4) and “missing local variable initialisation” faults (Figure 5.6), similar behaviour of the target applications was still observed. However, some differences were noted for Lzari and Gzip. Considering “missing local variable initialisation” faults, the application behaviour difference can be explained by the use of register variables in the source code. Regarding “assignment instead of equality comparison” faults, the difference is caused by the fact that the patterns used to find suitable locations for the injection of this fault also matched some locations were such faults could not be emulated, namely, locations that were originated by high-level expressions of the type “rvalue == something”, which cannot be mutated to “rvalue = expression” without generating a syntactic error. Although a perfect match in application behaviour was not attained for these two kinds of faults, the reasonable match observed suggests that these faults can still be emulated in this way.

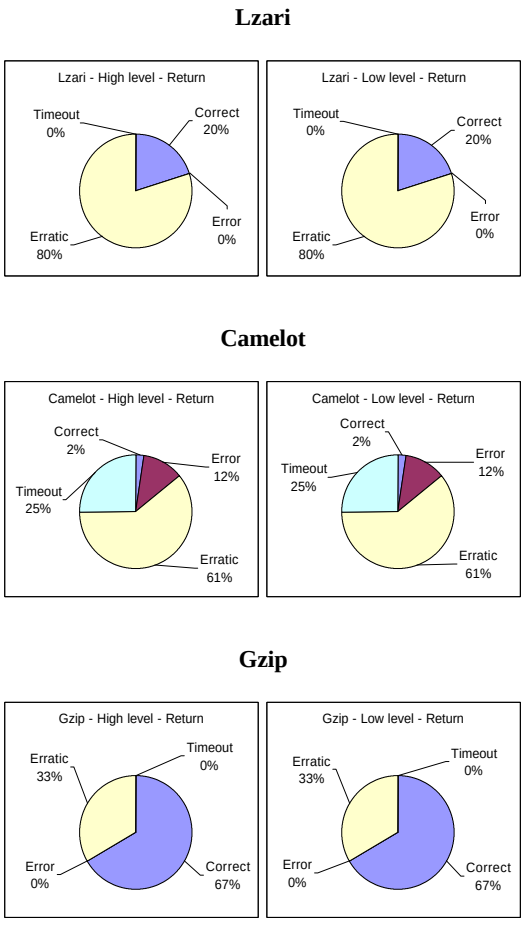


Figure 5.5 – Results for missing or wrong return statement.

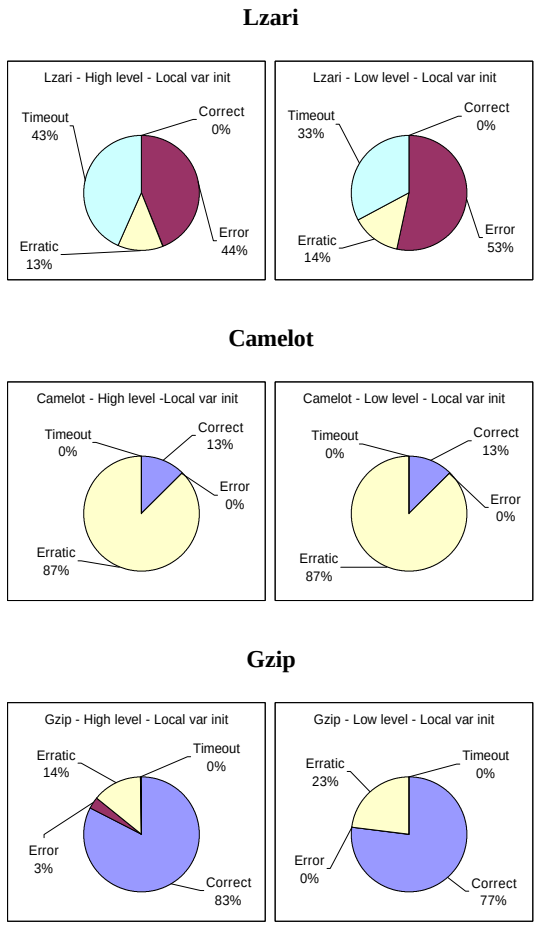
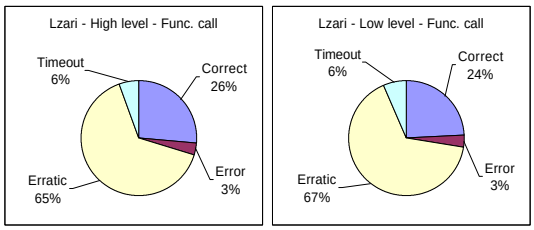
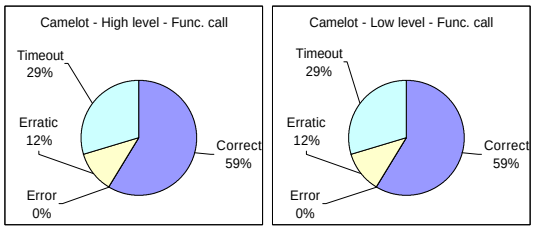


Figure 5.6 – Results for missing local variable initialisation

Lzari



Camelot



Gzip

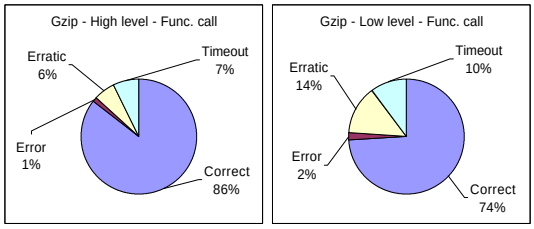


Figure 5.7 – Results for missing function call.

5.3 Generalisation of G-SWFIT

As one of the goals is to apply the technique to COTS software modules, we may not even know which compiler or language has been used to produce the executable code. Thus, it is very important to investigate how the proposed technique can be generalised and ported. The portability of the proposed technique is dependent on several factors:

a) Use of different compiler optimisation settings

b) Use of different compilers

c) Use of different high-level languages

d) Different host architectures

In practice, what is needed to generalise the technique is to investigate how the factors mentioned above influence the library of low-level instruction patterns and corresponding high-level faults. Obviously, once we have a library of faults, the actual use of the technique is straightforward, as what is needed is to find the pattern and insert the mutations, which is largely independent on the set-up details.

In order to study the generalisation of G-SWFIT we compiled the synthetic application (the same of section 5.1) using different optimisation settings, different compilers, target

architectures, and a different language (i.e., the same application in Pascal) and analyse the resulting low-level code to check if the instruction patterns initially identified were still present or if new patterns have to be added to the library. The following subsections discuss these results.

5.3.1 Different Optimisation Techniques

With respect to different optimisation settings using the same compiler, all possible settings for the tested compilers (Visual C++, Borland C++, GNU C++, Turbo C++) were evaluated, including optimising code for speed and for size. It has been observed that different patterns are generated for a few high-level constructs, depending on the optimising settings used. However, the conclusion regarding different optimising techniques is that, although some new low-level instructions patterns have to be considered, none of the new patterns collides with the existing ones, i.e., no ambiguity raises regarding to which high-level language construct does a given sequence of low-level instruction relates to. Thus, this issue has the only consequence of augmenting the library of low-level patterns and corresponding mutations.

5.3.2 Different Compilers

To address the issue of different compiler usage, the low-level code generation of the following compilers was observed for the C language in the Intel / Windows platform: Borland C++, Visual C++, Turbo C++, and GNU C++. It is worth noting that the set of observed compilers encompass both recent and old compilers. The inclusion of older compiler addresses the usage of legacy COTS and the lack of knowledge about which compiler was used to produce a given COTS. For all the observed compilers, the code generation is essentially the same. This is due to the existence of compilation standards.

5.3.3 Different Languages

Both C++ and Pascal languages were considered and experimented. The C++ language generates essentially the same low-level code as C, when the same kind of high-level constructs is used. Although Object-oriented constructs were not directly compared, no ambiguity with the existing patterns was detected. To encompass the new high-level faults made possible by the expanded syntax of C++, it only needs to augment the library of patterns and mutations. In the case of Pascal, the resulting low-level code is essentially the same as the resulting from C language. Some minor differences do exist, for instance in the way that parameter are passed to functions. However, concerning the patterns used in the case-study, no interference was noted.

5.3.4 Different Host Architectures

Regarding the issue of different host architecture, the following compilers/platforms were used: GCC for Linux on a IA32 machine and GCC for OSF Unix over an Alpha AXP.

In the GCC for Linux over IA32 case, the code output is essentially the same as compilers for Windows over IA32, including the variations caused by different optimisation settings. This was somewhat expected since the type of the underlying machine is the same. Concerning the

GCC for OSF Unix over Alpha AXP, the code is totally different. This was to be expected since there are very few similarities between an Intel 80x86 and an Alpha AXP processor. Therefore, low-level code must be also different. However, patterns can still be identified and related with the original high-level constructs. In the case of the Alpha processor there are specific coding standards, which greatly assist the task of defining low-level instruction patterns. The generalisation to different architectures is mainly a question of porting to a different processor, maintaining all the essential aspects of the technique: a library of patterns and mutations must be defined prior to the injection of emulated faults; the injection itself is carried out with ported versions of the same tools used in the case study. Furthermore, it can also be noted that the existing differences between different processors pose no difficulty for the deployment of this technique since support for different processors families at the same time (i.e., by the same suite of tools) is not expected to be necessary or even particular useful.

5.3.5 Generalisation Discussion

As a conclusion, the generation of the library of faults (and the technique itself) is mainly dependent on the target architecture. This means that we need as many libraries of faults as architectures we want to cover. All the other factors evaluated have also some influence on the libraries of faults, but this influence consists of having a more or less complete library. That is, the more compilers and optimisation settings are used, the more complete the library is.

Another important conclusion concerning DBench goals is that the faultload equivalence (considering both high-level versus low-level faults and equivalence across different systems and platforms) has probably to be drawn in statistical terms. The common ground for the definition of faultloads based on software faults is the high-level (i.e., source code level) fault classification (assumed to be generic, in spite of the different programming languages) and the fact that G-SWFIT can be ported across different systems and platforms.

5.4 Conclusions

G-SWFIT technique emulates software faults by selective mutations introduced at machine-code level. The idea is to find suitable locations for the accurate emulation of specific high-level languages programming errors that are usually responsible for common software faults. The key aspects of this technique are a library of low-level instructions patterns and mutations that relate to specific high-level faults in specific constructs and structures, and a pre-processing step of the target application to generate a (large) number of mutants. The execution of each mutant represents the injection of a fault.

Experimental results show that most of the patterns provide good accuracy, while some provide a not so good but still acceptable accuracy. This suggests that this technique is in fact a good way to emulate software faults when no source code is available, which is a crucial goal for DBench. Another strong point of this technique is the fact that it can be fully automated, which is essential for the definition of dependability benchmark faultloads.

The generalisation and portability of G-SWFIT is mainly dependent on the target architecture (i.e., the target processor programming model), while aspects such as the compiler

optimisation settings, different compilers, and language used to program the target application only influences the size of the library of faults and the effort needed to generate this library.

6 Operator Faults in Transactional Systems

Considering the three major types of faults addressed in DBench (hardware, software, or operator faults), available studies clearly point operator faults and software faults as the most frequent causes for computer failures [Gray 1990, Sullivan & Chirallege 1991, Sullivan & Chirallege 1992, Lee & Iyer 1995, Velpuri 1995, Kalyanakrishnam *et al.* 1999, Sunbelt 1999]. This is particularly true for very complex systems with many thousands of lines of code, as it is notably the case of typical transactional systems.

Previous chapters addressed the emulation of software faults. This chapter discusses the emulation of operator faults in transactional systems. The great complexity of the administration of transactional systems (which is clearly dominated by the underlying database management systems administration) and the need of tuning and administrating the system in a daily basis, clearly explains why operator faults (i.e., wrong system administrator actions) are considered the prevalent type of fault in complex transactional systems (even more important than software faults). Several interviews with database administrators (DBAs) of real databases installations conducted on behalf of our research work also confirmed the prevalence of operator faults.

The structure of this chapter is the following: next section presents background on DBMSs, and Section 6.2 discusses DBMS administration. Section 6.3 discusses the problem of operator faults in DBMS and presents a comparative analysis of administrator faults in three different DBMSs. Section 6.3 also proposes some guidelines for the definition of faultloads based on operator faults. Section 6.4 presents an example of DBMS recovery benchmarking and Section 6.5 concludes the chapter.

6.1 Background on DBMSs

Most of the transactional systems available today use a database management system as transactional engine. For that reason, transactional systems are strongly influenced by database technology, and then it is important to summarise the key concepts on database systems before discussing the problem of operator faults in transactional systems.

A database is a collection of data describing the activities of one or more related organisations [Ramakrishnan 1999][7]. The software designed to assist in maintaining and using databases is called database management system, or DBMS. A DBMS allows users to define the data to be stored in terms of a data model, which is a collection of high-level metadata that hide many low-level storage details. Most DBMS available today are based on the relational data model, which was proposed by E. F. Codd in 1970 [Codd 1970, Codd 1990]. The relational data model is very simple and elegant, and defines a database as a collection of one or more relations, where each relation is a table with rows and columns. DBMSs based on the

[7] This first paragraph is a condensed view of key definitions presented in chapter 1 of the book "Database Management Systems", by R. Ramakrishnan, second edition, McGraw Hill, ISBN 0-07-232206-3.

relational data model are frequently called relational database management systems. In the rest of the chapter we will use the term DBMS to refer to relational database management systems.

In practice, a typical database application (e.g., banking, insurance companies, telecommunications, etc) is a client-server system (either a traditional client-server or a three tier system) where a number of users are connected to a database server via a terminal or a desktop computer (today the trend is to access database servers through the internet using a browser). The user's actions are translated into SQL commands (Structured Query Language: the relational language used by DBMS [Date & Darwen 1993]) by the client application and sent to the database server. The results are sent back to the client to be displayed in the adequate format by the client application.

A very important notion is the concept of transaction [Gray 1990]. In a simplified view, a transaction is a set of commands that perform a given action and take the database from a consistent state to another consistent state. Transaction management is an important functionality of modern DBMSs, and it is directly related to dependability aspects, particularly in what concerns concurrency control and recovery. Concurrency control is the activity of coordinating the actions of processes that operate in parallel and access shared data, and therefore potentially interfere with each other. Recovery assures that faults (hardware, software, or operator faults) do not corrupt persistent data stored in the database tables.

In order to correctly deal with concurrency control and recovery, DBMS transactions must fulfil the following properties: **atomicity** (either all actions in the transaction are executed or none are), **consistency** (the execution of a transaction results in a consistent database state), **isolation** (the effects of a transaction must be understood without considering other concurrently executing transactions), and **durability** (the effects of a transaction that has been successfully completed must persist, even when the system has a failure after the transaction is finished). These properties are known as the ACID properties.

6.2 DBMS Administration

A database administrator (DBA) is responsible for developing and maintaining a database installation. Typically, a DBA is someone with many years of practice and has a vast experience with one or more of the major DBMS products, such as Oracle Database Server, Sybase Adaptive Server, Informix Dynamic Server, and Microsoft SQL Server.

The major DBMS available today are extremely complex and require a regular and demanding administration. The DBA is responsible for managing all the aspects concerning the DBMS environment, which makes the list of tasks performed by a database administrator quite extensive. Although different DBMS have different implementation and functionalities, administration tasks can be grouped in core areas common to all of them. The following points summarise those administration areas (each area corresponds to many administration tasks) and give an idea of the huge complexity of the administration of a database system.

- *Memory and processes administration*: The main goal is to optimise I/O operations through the correct definition of the optimal size of the different memory areas,

particularly the cache and buffer policies. Additionally, database administrator has to manage a number of different process types and the communications between the database clients and the server.

- *Security management*: Security is an important issue in database administration and requires continuous attention from the DBA. Some typical tasks include adding and removing users, managing privileges, and monitoring resources utilisation.
- *Storage administration*: The main goal is to minimise the contention when accessing the data and the fragmentation in the physical storage of the database objects. Several aspects have to be managed, such as the replication of some types of files, the distribution of the files by several disks, and other parameters related to physical space allocation.
- *Database objects administration*: The database administrator must frequently monitor and manage the database objects (tables, indexes, etc.) in order to maximise the system performance. For example, create the adequate indexes, cluster or partition some objects, and configure space allocation parameters, are some of the important tasks that a DBA has to execute frequently.
- *Recovery mechanisms administration*: The tuning of the recovery mechanism is clearly the most difficult area in database administration. Correctly configure the recovery mechanisms in order to minimise the recovery time and the lost transactions in a failure situation is one of the most important database administration aspects. However, it is also important to conciliate the aspects related to recovery with system performance. The main goal is to achieve the right balance between system recoverability and performance.

These major areas of administration can be found in any commercial DBMS, as they are related to core functions available in all DBMS [Ramakrishnan 1999]. Table 6.1 presents the main administration tasks for each one of these areas (each task can be divided in several subtasks) for three of the leading DBMS in the market: Oracle 8i, Sybase Adaptive Server 12.5, and Informix Dynamic Server 9.3. The tasks presented are grouped using the administration areas presented before and have been identified based on field experience on database administration, the comparative analysis of the administration manuals of the different products considered, and through discussions and interviews with several database administrators of real databases installations. Each column shows the administration tasks for a given DBMS and tasks in the same row are considered equivalent.

Although the details of some administration tasks are specific to each DBMS, the standard SQL used by the vast majority of DBMS greatly simplifies the establishment of equivalent tasks in different DBMS implementation. However, some of the tasks are intimately related to specific features of a given DBMS and do not have counterparts in other DBMS. There are also some tasks in a DBMS that may correspond to two or more tasks in another DBMS.

Table 6.1: Administration tasks in different DBMS

Class	Oracle 8i	Sybase Adaptive Server 12.5	Informix Dynamic Server 9.3
Memory and Processes Administration	Install and config. the database server	Install and config. the database server	Install and config. the database server
	Create Oracle databases		
	Startup and shutdown	Startup and shutdown	Startup and shutdown
	Manage Oracle processes	Manage multiprocessor servers	Manage virtual processors and threads
	Manage SGA allocation	Configure memory	Manage shared memory
		Configure data caches	
	Manage job queues		
	Set database configuration parameters	Set database configuration parameters	Set database configuration parameters
	Manage client/server communications	Manage client/server communications	Manage client/server communications
Security Management	Establish security policies	Establish security policies	Establish security policies
	Manage users and resources	Manage adaptive server logins and database users	Manage users
		Limit Access to Server Resources	Monitor resources and user activity
	Manage user privileges and roles	Manage user permissions and roles	Grant and revoke privileges
			Create and use roles
		Manage remote servers	
	Audit database use	Audit	Audit
Storage Administration		Create and manage user databases	
		Set database options	
	Manage tablespaces	Create and use segments	Manage databases
			Manage tblspaces
	Manage datafiles	Initialise and mirror database devices	Manage dbspaces, blobspaces, sbspaces, and extspaces
	Manage rollback segments		
	Manage temporary segments		Manage temporary dbspaces and temporary sbspaces
		Reorganise space in tables	
DB Objects	Manage tables	Manage tables	Manage tables
	Manage performance optimisation objects	Manage performance optimisation objects	Manage performance optimisation objects
	Manage other objects	Manage other objects	Manage other objects
	Manage objects replication		
Recovery Mechanisms Administration	Develop backup and recovery strategies	Develop backup and recovery plans	Develop backup and recovery plans
	Manage control files		
	Manage the online redo log	Manage transaction Log	Manage logical-log files
	Manage archived redo logs	Restore system databases	Manage the physical log
	Perform backups and recovery	Backup and restore user databases	Perform manual recovery
			Manage mirroring
			Use high-availability data replication
	Check data block corruption	Check database consistency	Check consistency

6.3 Operator Faults in DBMSs

Operator faults in database systems are database administrator mistakes. End-user errors are not considered, as the end-user actions do not affect directly the dependability of the system. In fact, the end-users do not have direct access to the DBMS (e.g., do not execute SQL

commands, even in user accounts), and they are only allowed to use the database through well-defined interface applications that isolate them form the DBMS (e.g., an ATM interface allows the users to perform high-level operations such as withdraw money or check account balance, and there is no room for user mistakes that affect the system dependability).

As we can see from the list of administration tasks presented in Table 6.1, database administrators manage all aspects of DBMSs. In spite of constant efforts to introduce self-maintaining and self-administering features in DBMS, database administration still is a job heavily based on human operators (and this picture is not likely to change in the near future). Obviously, administrator mistakes easily hurt DBMS availability, which shows the interest of benchmarking the behaviour of DBMSs in the presence of these faults.

6.3.1 Classes of Operator Faults

As mentioned before, the list of tasks performed by a DBA is very long. Although some of those tasks are common to several DBMSs, different database systems have different possibilities for administration and consequently different sets of possible operator faults. Thus, we have decided to analyse the main database administration areas and tasks and try to map them in classes of operator faults common to all DBMSs. Table 6.2 shows the proposed classes of operator faults (each class of faults corresponds to many types of administrator mistakes) and gives some examples of faults for each class.

Table 6.2: Classes of DBMS operator faults

Classes of DBMS operator faults	Description
Memory and processes administration	Mistakes in the administration of processes and memory structures. Deleting or corrupting the database initialisation files, defining incorrectly the memory allocation and processes initialisation parameters are typical faults related to processes and memory administration. Another typical fault is the accidental database shutdown that causes the loss of service.
Security management	Mistakes in the attribution of passwords, access privileges, and disk space to users. These are very problematic faults in database administration, as their effects are difficult to detect.
Storage administration	Mistakes in the administration of the physical and logical storage structures. Common examples of this class of faults are: the removal or corruption of database files, the incorrect distribution of files by several disks, and letting the storage structures run out of space.
Database object administration	Errors related to the management of the user objects. The removal of a user object (e.g., table, index, cluster, etc.), the incorrect configuration of the user objects storage parameters, and the incorrect use of the optimisation structures (e.g., indexes, clusters, etc.) are common faults related to the database schema administration.
Recovery mechanisms administration	Mistakes in the configuration and administration of the database recovery mechanisms. Some typical examples are: the inexistence of backups, the removal or corruption of a log file, and the inexistence of archive logs.

6.3.2 Comparative Analysis of Administration Faults in Several DBMSs

Operator faults are mistakes made by human operators when executing administration tasks. Different DBMSs include different sets of administration tasks and consequently have different sets of possible operator faults. In order to identify the differences and similarities among different DBMSs concerning operator faults, a comparative analysis has been made to identify the operator faults associated to each administration task. Tables 6.3, 6.4, and 6.5

present the list of the administration tasks and respective operator faults types identified for the three DBMSs shown in Table 6.1, respectively Oracle 8i, Sybase Adaptive Server 12.5, and Informix Dynamic Server 9.3. Note that a given fault type actually represents many possible faults (e.g., as a typical database has hundreds of tables there are many possibilities for deleting a table by mistake).

Table 6.3: Administration tasks and operator faults in Oracle 8*i* DBMS

Class	Administration tasks	Operator Fault Types
Memory and Processes Admin.	Create Oracle database	Create an Oracle database during peek workload
	Startup and shutdown	Make a database shutdown inadvertently
	Manage Oracle processes	Incorrect configuration of the processes parameters
	Manage SGA allocation	Incorrect configuration of the SGA parameters
	Manage job queues	Incorrectly set a job to a peek workload time
	Set database configuration parameters	Incorrect config. the DB parameters; Remove or corrupt the initialisation file
	Manage client/ server comm.	Incorrect config. of the maximum number of user sessions; Kill a user session
Security Manag.	Manage Users and Resources	Database access level faults (passwords); Delete a database user; Assign incorrect profiles to users; Grant incorrect disk space to users
	Manage User Privileges and Roles	Incorrect attribution of system privileges to users; Incorrect attribution of object privileges to users; Incorrect attribution of roles to users
Storage Admin.	Manage tablespaces	Delete a tablespace; Set a tablespace offline; Allow a tablespace to run out of space
	Manage datafiles	Delete or corrupt a datafile; Set a datafile offline; Incorrect distribution of datafiles through disk
	Manage rollback segments	Delete a rollback segment; Set a rollback segment offline; Insufficient number of rollback segments; Allow a rollback segment to run out of space
	Manage temporary segments	Delete a temporary segment; Allow temporary segments to run out of space
DB Objects Admin.	Manage tables	Delete a table; Incorrect configuration of table's storage parameters; Setting NOLOGING option in tables
	Manage performance optimisation objects	Incorrect use of performance optimisation objects
	Manage other objects	Delete any database object
	Manage objects replication	Incorrectly replicate objects
Recovery Mechan. Admin.	Develop backup and recovery strategies	Develop a wrong backup and recovery strategy
	Manage control files	Delete a control file
	Manage the online redo log	Delete a redo log file or group; Store all redo log group members in the same location; Insufficient redo log groups to support archive logs
	Manage archived redo logs	Inexistence of archive logs; Delete an archive log file Store archive log files in the same disk as datafiles
	Perform backups and recovery	Backups missing to allow recovery; Make a hot backup during peek workload
	Address data block corruption	Correct data block corruption during peek workload

As we can see, it is possible to establish equivalence among many operator faults in different DBMSs. A very important aspect is that very sophisticated DBMSs with many things to administrate seem to be more prone to operator faults than simpler DBMSs, with only a few things to administrate (obviously, DBMS with less administration possibilities have limited tuning possibilities).

The interface of the administration tools is also very important to identify weaknesses in DBMSs concerning operator faults. Normally a DBMS includes several administration tools. These tools have two types of interface: graphical interface or SQL command line interface. In a graphical interface the database administrator executes the administration tasks by clicking on information boxes and buttons in a graphical environment. The administration tool translates those actions into SQL commands and submits them to the DBMS. In a SQL command line interface, the database administrator writes the SQL commands directly.

Table 6.4: Administration tasks and operator faults in Sybase Adaptive Server 12.5 DBMS

Class	Administration Tasks	Operator Faults Types
Memory and Processes Admin.	Startup and Shutdown	Make a database shutdown inadvertently
	Manage Multiprocessors Servers	Kill a process; Incorrect configuration of the processes parameters
	Configure Memory	Incorrect configuration of Memory parameters
	Conf. Data Caches	Incorrect configuration of Data Caches parameters
	Set Database Configuration Parameters	Incorrect config. of DB parameters; Remove or corrupt the config. file
	Manage Client/ Server Comm.	Kill a user session
Security Manag.	Manage Logins and Database Users	Database access level faults (passwords); Delete a database user
	Limit Access to Server Resources	Incorrect distribution of resources
	Manage User Permissions and Roles	Incorrect attribution of permissions to users; Incorrect attribution of roles to users
Storage Admin.	Create and Manage User Databases	Delete a user database
	Set Database Options	Incorrect configuration of database options
	Create and Use Segments	Delete a segment
	Initialise and Mirror Database Devices	Delete or corrupt a database device
	Reorganise Space in Tables	Reorganise space in table during peek workload
DB Objects Admin.	Manage Tables	Delete a table
	Manage Performance Optimisation Objects	Incorrect use of performance optimisation objects
	Manage Other Objects	Delete any database object
Recovery Mechan. Admin.	Develop Backup and Recovery Strategies	Develop a wrong backup and recovery strategy
	Manage Transaction Log	
	Restore System Databases	Inexistence of system databases backups
	Backup and Restore User Databases	Inexistence of user databases backups; Make a user database backup during peek workload
	Check Database Consistency	Check database consistency during peek workload

Table 6.5: Administration tasks and operator faults in Informix Dynamic Server 9.3 DBMS

Class	Administration Tasks	Operator Faults Types
Memory & Processes Admin.	Startup and Shutdown	Make a database shutdown inadvertently
	Manage Virtual Processes and Threads	Kill a process; Incorrect configuration of the processes parameters
	Manage Shared Memory	Incorrect configuration of Shared Memory parameters
	Set Database Configuration Parameters	Incorrect config. of DB parameters; Remove or corrupt the config. file
	Manage Client/ Server Comm.	Kill a user session
Security Manag.	Manage Users	Database access level faults (passwords); Delete a user
	Monitor Resources and User Activity	Incorrect distribution of resources
	Grant and Revoke Privileges	Incorrect attribution of privileges to users
	Create and Use Roles	Incorrect attribution of roles to users
Storage Admin.	Manage Databases	Delete a Database
	Manage Tblspaces	Delete or corrupt a Tblspace
	Manage Dbspaces, Blobspaces, Sbspaces, Extspaces	Delete or corrupt a Dbspace, Blobspace, Sbspace, or a Extspace
	Manage Temporary Dbspaces and Temp. Sbspaces	Delete a Temporary dbspace or a temporary Temporary subspace
DB Objects Admin.	Manage Tables	Delete a table
	Manage Performance Optimisation Objects	Incorrect use of performance optimisation objects
	Manage Other Objects	Delete any database object
Recovery Mechan. Admin.	Develop Backup/ and Recovery Strategies	Develop a wrong backup and recovery strategy
	Manage Logical-Log Files	Delete a logical-log file
	Manage the Physical Log	Delete a physical log file
	Manage Mirroring	Incorrectly mirroring of files
	Use High-Availability Data Replication	Incorrectly replication of files
	Check Consistency	Check consistency during peek workload

A very important aspect concerning administration tools is the existence or not of some kind of mechanisms to confirm and undo operator tasks [Brown & Patterson 2001]. In tools with a confirmation mechanism, the operator has to validate each operation (through a commit command or clicking in a button), which reduces the probability of occurrence of faults. The undo mechanism is essential and allows to rollback an operation after its execution (nullifying the fault if it exists). Obviously, it is very difficult to rollback some types of complex operations, such as deleting a file or deleting relational objects, without using extremely complex recovery procedures. That is the reason why most of the DBMS do not include undo facilities as a standard administration mechanism.

6.3.3 Operator Faults Emulation

The injection of operator faults in a DBMS can be easily achieved by reproducing common database administrator mistakes. That is, operator faults can be injected in the system by using exactly the same means used in the field by the real database administrator. Using the concept of "emulation distance" presented in the introduction of this deliverable, our proposal to emulate operator faults in DBMS is to use "distance zero emulation". In other words, we do not emulate faults as it is usual in traditional fault injection: we really reproduce operator faults.

In order to reproduce an operator fault in a way similar to what happens in real world the set of steps represented in Figure 6.1 must be followed. A very important aspect concerning the fault emulation is the instant of activation of the faults (fault trigger). The same fault activated in different moments may cause different behaviour according to the system state, which means that different instants for the injection of faults must be chosen (faults can be uniformly distributed over time or can be synchronised with a specific event or command of the workload).

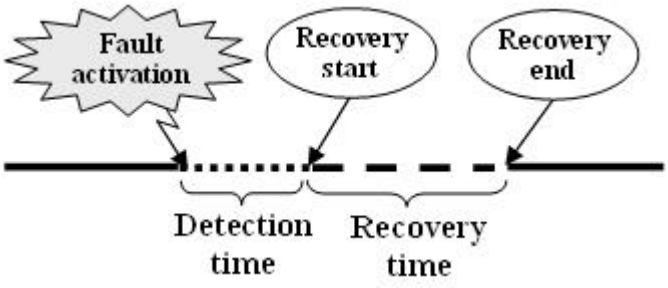


Figure 6.1: Steps to emulate an operator fault

The recovery start depends on the time needed to detect the error. However, the operation faults do not start the recovery process in an automatic way in most of the cases. Thus, the detection that some administration mistake has been made is normally carried out by the DBA himself, either because the DBA realise his own mistakes or because it is notified by the end-users that some functionality has been affected. As the detection time is highly human dependent, a typical detection time has to be established for benchmark purposes for each type of operator fault. In practice, the detection time is estimated for each fault based on previous experience and on the analysis of the possible fault effects. It is worth noting that the

detection time as shown in Figure 1 also includes the time required for the DBA to understand the effects of the fault and start the adequate recovery procedure.

Another relevant aspect to automate the experiments is that it is necessary to evaluate the type of recovery procedure that may be required after each fault. This means that the scripts used to inject each type of fault must also include all the steps required to start the adequate recovery procedure, which is much more complex than the actual reproduction of the operator faults.

6.4 Definition of Faultloads Based on Operator Faults

A faultload is a set of faults or stressful conditions that may affect the system. Two important aspects have to be considered in the definition of a faultload: fault representativeness and fault portability. To be useful, a faultload must be representative of real faults and, at the same time, must define the faults in a way that make them portable (i.e., valid) among the possible target systems. Concerning operator faults, although some faults are highly system dependent, the analysis made in the previous sections showed that most of the operator faults can be found in several DBMSs.

From the functional, the DBMS administration can be defined as a set of core functionalities, namely: memory and processes, security, storage, database objects, and recovery mechanisms. These functionalities are equivalent to the administration areas presented in sub-section 6.2. A possible solution to define a portable faultload is to focus on the high abstraction level that corresponds to the core functionalities of DBMS administration. This functional abstraction level corresponds to the set of administration functionalities common to most DBMSs.

To define a faultload based on operator faults, the following set of steps must be followed:

1) Identify the administration tasks for each core administration functionality.
2) Identify the operator fault types that may occur when executing each one of those administration tasks (in a similar way to tables 6.3, 6.4, and 6.5).
3) Define weights to each fault type according to the number of times the correspondent administration task is executed (reasonable estimation of the frequency of each fault can be obtained by field data, using for example real database logs).
4) Define the faultload as the exhaustive list of possible operator faults for all the types identified. The number of times a given fault type will appear in the faultload depends on the weights defined and each type must appear at least once.

The reason why we propose an exhaustive list of possible faults (taking into account the list of administration tasks for the core administration functionalities) is that different systems can be fairly compared in terms of recoverability if all the possible causes for DBMS recovery are evaluated in the benchmark. It is worth noting that the limited number of administration tasks assures that even the exhaustive list of operator faults is within acceptable bounds considering the number of faults.

It is important to note that some types of faults do not affect the system in a visible way concerning recovery. An example of these faults is the security class faults. When a DBA

introduces inadvertently a security fault the system continues to work normally until another person maliciously take advantage from that to break into the system (this is a second event). Fault types that need a second event to show up must be treated in a different way.

6.5 Conclusion

This chapter focuses the problem of emulating operator faults for dependability benchmarking of transactional systems based on a DBMS. A comparative analysis concerning administration tasks and operator faults in different DBMS is presented and a general classification for operator faults is proposed. A set of guidelines for the definition of faultloads based on operator faults for DMBS recovery benchmarking is then proposed based on the core DBMS administration functionalities.

7 Conclusion

In this deliverable it has been made an in depth study of fault representativeness from the SW, HW and operator faults point of view. Such study has been carried out developing conceptual frameworks to identify the main notions governing the problem of fault representativeness.

The conclusions can be summarised taking into account the context under which they were obtained:

- **Context and classes of fault Addressed**: The presence of a specific FTM on a target system will alter the error propagation path. Assuming a perfect coverage of the FTM the representativeness of the benchmark using the fault load characterised by the injected faults could be zero. Because of that fact it has been necessary to introduce a new distance parameter: the distance separating the level where the faults are injected from the level where the FTM is acting.
- **Fault propagation between logic and RTL abstraction level**: From the study of fault modelling at logic and RTL levels, three main conclusions can be considered:
    - Considering transient faults due to cosmic ray radiation, it is necessary to distinguish between the faults occurred in combinational circuits and in storage (memory and latches). This differentiation leads to the specification of **pulse** fault model, related to combinational circuits (while for storage circuits it is used the bit-flip model).
    - Recent studies point out that in future ICs the likelihood of apparition of transient faults will increase greatly. Moreover, some fault types that usually have been neglected (and subsequently not used in fault injection campaigns) will have an important impact in future technologies. This is the case of faults in combinational logic (**pulse** fault model). At higher working frequencies, the likelihood to latch a combinational fault is going to raise considerably. Other unconsidered fault models like **delay** and **indetermination**, are going to increase their influence also due to problems related to high speed working. At high frequencies, skin and Miller effects will make that delay in ICs will not be constant, originating even time violations that can lead to indeterminate outputs.
    - When studying the propagation of faults in combinational circuits to RTL level, it has been observed that the number of failures generated by injecting in hidden (not accessible via software) registers is not negligible. For that reason, SWIFI techniques should be complemented with another fault injection technique to access to such registers.
- **SW fault representativeness from OS point of view:** It has been compared the impact of three types of SWIFI techniques on the Linux OS (version 2.4.0.) Two of them target the kernel call parameters at the API (external faults) and the third one targets the parameters of the internal functions of the kernel. The results obtained refer to experiments focusing

on the scheduling and memory components of the kernel. Based on these experiments it can be stated:

- API-level fault injection is a good candidate to assess kernel robustness. Flipping bits in kernel call parameters is easy to implement and does not need any *a priori* analysis of the parameter data types. However, it requires a lot of time, as it needs 32 injections per parameter for a 32-bit kernel and simple data types.
- Applying invalid parameters is eight times faster (for a complete campaign) compared to a bit-flip campaign, but it needs an *a priori* analysis of the kernel call parameters. Also, it presents more advantages compared to bit-flip technique. As an example, it ensures more error propagation cases.

Concerning the representativeness point of view, we observed that external faults provoked very distinct behaviours when comparing to those induced by the real internal faults we considered (device driver faults). In particular, external faults were not able to activate the assertions based on real faults. This tends to indicate that it is unlikely that device driver faults could be easily emulated by injecting only at the API level, at least for the Linux kernel.

On the other hand, faults injected in internal function parameters were found, to some extent, representative of the considered real faults. Indeed, although the internal function parameters technique was not able to activate the assertions based on real faults, similar failure modes were provoked. This shows that the selected type (bit-flip) and location (interface) of the injected faults are not sufficient to provide a faultload matching the errors provoked by the considered real faults.

Accordingly, to study the OS robustness with regard to driver faults, in WP3 we will extend the prototype Benchmark to include the driver interface in addition to the API.

- **Software Faults from Application (Language) Point of View**. Because of the inherent problems of SW faults injection it has been proposed:
  - Use of **educated mutations** to improve the representativeness of the injected faults as much as possible.
  - Use of a **new technique to emulate software faults** by selective mutations introduced at the machine-code level.
  - Propose of an operating scenario for the injection of software faults in **which faults are injected in one module to evaluate the behaviour of the rest of the system**.

  In summary, it has been proposed a new technique for the injection of software faults called Generic Software Fault Injection Technique (G-SWFIT). G-SWFIT consists of finding key programming structures at the machine code-level where high-level software faults can be emulated.
- **Operator Faults in Transactional Systems**. The great complexity of the administration of transactional systems and the need of tuning and administrating the system in a daily basis, clearly explains why operator faults are considered the prevalent type of fault in complex transactional systems (even more important than software faults). Several

interviews with database administrators (DBA) of real databases installations conducted on behalf of our research work also confirmed the prevalence of operator faults. A comparative analysis concerning administration tasks and operator faults in different DBMS is presented and a general classification for operator faults is proposed.

References

[Aidemark *et al.* 2001] J. L. Aidemark, J. P. Vinter, P. Folkesson and J. Karlsson, "GOOFI: A Generic Fault Injection Tool", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), pp.83-88, IEEE CS Press, 2001.

[Akkerman 2001] W. Akkerman, Strace Home Page http://www.wi.leidenuniv.nl/~wichert/strace, 2001.

[Amerasekera & Najm 1997] E.A. Amerasekera and F.N. Najm, "Failure mechanisms in semiconductor devices", John Whiley & Sons, 1997.

[Arlat 1992] J. Arlat, "Fault Injection for the Experimental Validation of Fault-Tolerant Systems", in *Proc. Workshop Fault-Tolerant Systems*, (Kyoto, Japan), pp.33-40, IEICE, Tokyo, Japan, 1992.

[Arlat *et al.* 1990] 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*, 16 (2), pp.166-182, February 1990.

[Arlat & Crouzet 2002] J. Arlat and Y. Crouzet, "Faultload Representativeness for Dependability Benchmarking", in *Supplement Int. Conference on Dependable Systems and Networks (DSN-2002),* (Washington, DC, USA), IEEE CS Press, 2002.

[Arlat *et al.* 2002] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Transactions on Computers*, 51 (2), pp.138-163, February 2002.

[Avizienis *et al.* 2001] A. Avizienis, J.-C. Laprie and B. Randell, "Fundamental Concepts of Dependability", LAAS-CNRS, France, Research Report, N°01-145, April 2001.

[Baraza *et al.* 2002] J.C. Baraza. J. Gracia, D. Gil, P.J. Gil, "A prototype of a VHDL-Based Fault Injection Tool. Description and Application". *Journal of Systems Architecture*, ISSN. 1383-7621, 47(10), pp. 847-867, April 2002.

[Barton *et al.* 1990] J. H. Barton, E. W. Czeck, Z. Z. Segall and D. P. Siewiorek, "Fault Injection Experiments Using FIAT", *IEEE Transactions on Computers*, 39 (4), pp.575-582, April 1990.

[Bowman *et al.* 1999] I. T. Bowman, R. C. Holt & N. V. Brewster, "Linux as a case study: Its Extracted Software Architecture", in *Proc. 21st Int. Conf. on Software Engineering* (Los Angeles, CA, USA), 1999.

[Brown & Patterson 2001] A. Brown and D. Patterson, "To Err is Human", *First Workshop on Evaluating and Architecting System Dependability (EASY), Joint organized with IEEE/ACM 28th Inte. Symp. on Computer Architecture (ISCA) and the IEEE Int. Conf. on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), July 1st, 2001.

[Carreira *et al.* 1998] 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*, 24 (2), pp.125-136, February 1998.

[Carreira *et al.* 1999] J. V. Carreira, D. Costa and J. G. Silva, "Fault Injection Spot-checks Computer System Dependability", *IEEE Spectrum*, 36, pp.50-55, August 1999.

[Carter 2001a] P. Carter, "Common C Errors", 2001, available at http://www.drpaulcarter.com/cs/common-c-errors.php.

[Carter 2001b] P. Carter, "PC Assembly Language", 2001, available at http://www.drpaulcarter.com/pcasm.

[Cha *et al.* 1993] H. Cha, E.M. Rudnick, G.S. Choi, J.H. Patel, R.K. Iyer, "A fast and accurate gate-level transient fault simulation environment". in *Procs. 23rd Symp. On Fault-Tolerant Computing Systems (FTCS-23)*, Toulouse (France), pp. 310-319, June 1993.

[Cheynet *et al.* 2000] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda and M. Violante, "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with respect to Transient Errors", *IEEE Transactions on Nuclear Science*, 47 (6), pp.2231-2236, December 2000.

[Chillarege & Bowen 1989] R. Chillarege and N. S. Bowen, "Understanding Large System Failures — A Fault Injection Experiment", in *Proc. 19th Int. Symp. on Fault-Tolerant Computing (FTCS-19)*, (Chicago, IL, USA), pp.356-363, IEEE Computer Society Press, 1989.

[Chillarege *et al.* 1992] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray & M. Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements", *IEEE Transactions of software engineering* 18(11): 943-956, 1992.

[Chillarege 1995] R. Chillarege, "Orthogonal Defect Classification", Chapter 9 of *Handbook of Software Reliability Engineering*, Michael R. Lyu Ed., IEEE Computer Society Press, McGraw-Hill, 1995.

[Choi & Iyer 1992] G. S. Choi and R. K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis", *IEEE Transactions on Computers*, 41 (12), pp.1515-1526, December 1992.

[Christmansson & Chillarege 1996] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults-Based on Field Data", in *Proc. 26th Int. Symp. on Fault Tolerant Computing (FTCS-26)*, (Sendai, Japan), pp. 304-313, 1996.

[Christmansson *et al.* 1998] J. Christmansson, M. Hiller & M. Rimen, "An Experimental Comparison of Fault and Error Injection", in *Proc. 9th Int. Symp. on Software Reliability Engineering*, (Paderborn, Germany), pp.369-378, 1998.

[Codd 1970] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, 1970.

[Codd 1990] E.F. Codd, "The Relational Model for Database Management", Addison-Wesley Publishing Company, 1990, ISBN 0-201-14192-2.

[Constantinescu 2001] C. Constantinescu, "Dependability Analysis of a Fault-Tolerant Processor", in *Proc. 2001 Pacific Rim International Symposium on Dependable Computing*, pp. 63–67, 2001.

[Constantinescu 2002] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2002)*, (Washington, DC, USA), Accepted.

[Cramon 2001] J. Cramon, "12 C Common Errors", available at http://www.edm2.com/0504/12cerr.html.

[Daran & Thévenod-Fosse 1996] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations", in *Proc. Int. Symp. on Software Testing and Analysis (ISSTA'96)*, (S. J. Zeil, Ed.), (San Diego, CA, USA), pp.158-171, ACM Press, 1996.

[Date & Darwen 1993] C.J. Date and H. Darwen, "The SQL Standard", Third Edition, Addison-Wesley Publishing Company, 1993, ISBN 0-201-55822-X.

[DeMillo *et al.* 1988] R. DeMillo, D. Guindi, W. McCracken, A. Offut, and K. King, "An Extended Overview of the Mothra Software Testing Environment", in *Proc. of ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Verification, and Analysis*, pp. 142-151, July 1988.

[Devera 2001] Devik's MM Gallery *http://luxik.cdi.cz/~devik/mm.htm*, 2001.

[Durães & Madeira 2002] J. Durães and H. Madeira, "Emulation of software faults by selective mutations at machine-code level", submitted.

[Engler *et al.* 2000] D. Engler, B. Chelf, A. Chou & S. Hallem, "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions", in *Proc. 4th Symp. on Operating Systems Design and Implementation (OSDI)*, (San Diego, CA, USA), USENIX, 2000.

[Favalli *et al.* 1991] M. Favalli, P. Olivo, and B. Riccò. "Fault simulation for general FCMOS ICs", *Journal of electronic testing: theory and applications*, 2, 181-190, 1991.

[Finkel *et al.* 1992] D. Finkel, R. Kinicki, J. Lehmann & J. CaraDonna, "Comparisons Of Distributed Operating System Performance Using The Wpi Benchmark Suite", Worcester Polytechnic Institute, (MA, USA), Technical Report, N°CS-TR92-2, 1992.

[Folkesson *et al.* 1998] P. Folkesson, S. Svensson and J. Karlsson, "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection", in *Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28)*, (Munich, Germany), pp.284-293, IEEE CS Press, 1998.

[Freeman 1996] L.B. Freeman, "Critical charge calculations for a bipolar SRAM array.", *IBM Journal of Research and Development*, Vol. 40, No. 1, Jan. 1996.

[Fuchs 1998] E. Fuchs, "Validating the Fail-Silence of the MARS Architecture", in *Dependable Computing for Critical Applications (Proc. 6th IFIP Int. Working Conference on Dependable Computing for Critical Applications: DCCA-6,* (Grainau, Germany), W. H. Sanders, Ed., Dependable Computing and Fault-Tolerant Systems, 11, (J.-C. Laprie, Ed.), pp.225-247, IEEE CS Press, Los Vaqueros, CA, USA, 1998.

[Ghate 1981] P.B. Ghate, "Aluminum alloy metallization for integrated circuits", *Thin Solid Films*, 83, p.195-205, 1981.

[Gil 1999] D. Gil, "Validación de Sistemas Tolerantes a Fallos mediante inyección de fallos en modelos VHDL", *Tesis Doctoral*, Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia (Spain), 1999.

[Gimpel 2001] "PC Lint Reference manual", Gimpel Software, available at http://www.gimpel.com.

[Gray 1990] J. Gray, "A Census of Tandem Systems Availability Between 1985 and 1990", *IEEE Transactions on Reliability*, Vol. 39, No. 4, pp. 409-418, October 1990.

[Hawkins 2000] C. Hawkins. "CMOS IC failure mechanism and defect based testing", *2nd summer course on selected microelectronic design & test topics*, (Palma de Mallorca, Spain) 7th july 2000.

[Hazucha & Svennson 2000] P. Hazucha and C. Svennson. "Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate", *IEEE Transactions on Nuclear Science*, Vol. 47, No. 6, Dec. 2000.

[Hsueh *et al.* 1997] M.-C. Hsueh, T. K. Tsai and R. K. Iyer, "Fault Injection Techniques and Tools", *Computer*, 30 (4), pp.75-82, April 1997.

[Hu & Lu 1999] C. Hu and Q. Lu, "A unified gate oxide reliability model". Proc. 39th IRPS, pp. 47-52, 1999.

[IEEE 1993] "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1993.

[Jones 1987] R.E. Jones, "Line width dependence of stresses in aluminum interconnect", in *Proc. 25th IRPS*, pp. 9-14, 1987.

[Juhnke & Klar 1995] T. Juhnke and H Klar, "Calculation of the soft error rate of submicron CMOS logic circuits", *IEEE Journal of Solid-State Circuits*, 30:830–834, July 1995.

[Kalyanakrishnam *et al.* 1999] M. Kalyanakrishnam, Z. Kalbarczyk, R. Iyer, "Failure Data Analysis of a LAN of Windows NT Based Computers", in *Proc. Symp. on Reliable Distributed Database Systems (SRDS18),* Switzerland, pp. 178-187, October 1999.

[Kanawati *et al.* 1995] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System", *IEEE Transactions on Computers*, 44 (2), pp.248-260, February 1995.

[Kanoun *et al.* 1997] K. Kanoun, M. Kaâniche and J.-C. Laprie, "Qualitative and Quantitative Reliability Assessment", *IEEE Software*, 14 (2), pp.77-86, March 1997.

[King *et al.* 1989] R. King, C. van Schaick, J. Lusk, "Electrical overstresses of non-encapsulated bond wires", in *Proc. 27th IRPS*, p.141-151, 1989.

[Koenig 1989] A. Koenig, "C Traps and pitfalls", Addison-Wesley, 1989.

[Koopman & DeVale 1999] P. Koopman & J. DeVale, "Comparing the Robustness of POSIX Operating Systems", in *Proc. 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29)*, (Madison, WI, USA), pp.30-37, IEEE Computer Society Press, 1999.

[Koopman *et al.* 1997] P. Koopman, J. Sung, C. Dingman, D. P. Siewiorek & T. Marz, "Comparing Operating Systems Using Robustness Benchmarks", in *Proc. 16th Int. Symp. on Reliable Distributed Systems (SRDS-16)*, (Durham, NC, USA), pp.72-79, IEEE Computer Society Press, 1997.

[Krishnamurthy *et al.* 1998] N. Krishnamurthy, V. Jhaveri and J. A. Abraham, "A Design Methodology for Software Fault Injection in Embedded Systems", in *Proc. IFIP Int. Workshop on Dependable Computing and Its Applications (DCIA-98)*, (Y. Chen, Ed.), (Johannesburg, South Africa), pp. 237-248, Wits University, 1998.

[Lee & Iyer 1995] I. Lee & R.K. Iyer, "Software Dependability in the Tandem GUARDIAN System", *IEEE Transactions of software engineering*, 21(5): 455-467, 1995.

[Lee *et al.* 2001] W.H. Lee *et al.* "Data retention failure in NOR flash memory cells", in *Proc. 39th IRPS*, pp. 57-61, 2001.

[Liden *et al.* 1994] P. Liden *et al.*, "On latching probability of particle induced transients in combinatorial networks", in *Proc. 24th Int. Symp. on Fault Tolerant Computer Systems (FTCS–24)*, pp. 821-824, 1994.

[Lin *et al.* 1996] H.C. Lin *et al.*, "Plasma charging induced gate oxide damage during metal etching and ashing", in *Proc. 1st Int. Symp. On Plasma Process-Induced Damage*, pp. 113-116, 1996.

[Lyu 1996] M.R. Lyu, "Handbook of Software Reliability Engineering", IEEE Computer Society Press, McGraw-Hill, 1996.

[Madeira *et al.* 2000] H. Madeira, D. Costa and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000)*, (New York, NY, USA), pp.417-426, IEEE Computer Society Press, 2000.

[Madeira *et al.* 2001] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johanson and R. Lindström, "Preliminary Dependability Benchmark Framework", Available at http://www.laas.fr/dbench/delivrables.html, DBench Project IST 2000-25425 Deliverable, N°CF2, September 2001.

[McVittie 1996] J. McVittie, "Plasma Charging damage: an overview", in *Proc. 1st Int. Symp. On Plasma Process-Induced Damage*, pp. 7-20, 1996.

[Microchip 2000] Microchip, "PIC16C5X", Available at http://www.microchip.com, Microchip Technology Inc., 2000

[Model Technology 2001] Model Technology, "Modelsim SE User's Manual. Version 5.5e", 2001.

[Murphy & Levidow 2000] B. Murphy & B. Levidow, "Windows 2000 Dependability", in *Workshop on Dependable Networks and Operating Systems, at the Int. Conf. on Dependable Systems and Networks (DSN 2000)*, (New York, NY, USA), pp.D20-D28, 2000.

[Musa 1999] J. Musa, "Software Reliability Engineering", McGraw-Hill, 1999.

[Newman 2001] P. Newman (moderator), "The Risks Digest, Forum on Risks to the Public in Computers and Related Systems", *ACM Committee on Computers and Public Policy*, Vol. 1 (1986) to Vol. 21 (2001), available at http://catless.ncl.ac.uk/Risks.

[Oates 1993] A. Oates, "Electromigration in Stress-Voided Al Alloy Conductors", in Proc. 31st IRPS, pp. 20-30, 1990.

[Ogawa *et al.* 2001] E.T. Ogawa *et al.*, "Statistics of electromigration early failures in Cu/oxide dual-damascene interconnects", in *Proc. 39th IRPS*, pp. 341-350, 2001.

[Pagaduan *et al.* 2001] F.E. Pagaduan *et al.*, "The effects of plasma induced damage on transistor degradation and the relationship to field programmable gate array performance", in *Proc. 39th IRPS*, pp. 315-319, 2001.

[Pradhan 1986] D.K. Pradhan, "Fault-Tolerant Computer System Design", I.S.B.N.: 0-13-057887-8, Prentice-Hall, 1996.

[Pucknell & Eshraghian 1994] D.A. Pucknell, K. Eshraghian. "Basic VLSI Design". Prentice Hall.1994.

[Ramakrishnan 1999] R. Ramakrishnan, "Database Management Systems", Second edition, McGraw Hill, ISBN 0-07-232206-3, 1999.

[Rangan *et al.* 1999] S. Rangan *et al.*, "A model for channel hot carrier reliability degradation due to plasma damage in MOS devices", in *Proc. 37th IRPS*, pp. 370-374, 1999.

[Rodder *et al.* 1995] M. Rodder, S. Aur, C. Chen, "A scaled 1.8V, 0.18µm gate lenght CMOS technology: device design and reliability considerations", *Tech. Dig. IEDM*, pp. 415-418, 1995.

[Rodríguez *et al.* 1999] M. Rodríguez, F. Salles, J. C. Fabre and J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid", in *Proc. 3rd European Dependable Computing Conf. (EDCC-3)*, (E. M. J. Hlavicka, A. Pataricza, Ed.), (Prague, Czech Republic), LNCS, 1667, pp.143-160, Springer, 1999.

[Rodríguez *et al.* 2002] M. Rodríguez, J.-C. Fabre and J. Arlat, "Assessment of Real-Time Systems by Fault-Injection", in *Proc. European Safety and Reliability Conference (ESREL-2002)*, (Lyon, France), pp. 101-108, 2002.

[Shin *et al.* 1993] H. Shin *et al.*, "Plasma-etching charge-up damage to thin oxides", *Sol. St. Tech.*, pp. 29-34, August 1993.

[Shirley & Blish 1987] C.G. Shirley and R. Blish, "Thin film cracking and wire ball shear in plastic DIPs due to temperature cycle and thermal shock", in *Proc. 25th IRPS*, pp. 238-249, 1987.

[Shivakumar *et al.* 2002] P. Shimakumar *et al.* "Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2002)*, (Washington, DC, USA), Accepted.

[Siewiorek 1994] D.P. Siewiorek. "Reliable Computer Systems. Design and Evaluation", Digital Press, 1994.

[Srinivasan *et al.* 1994] G.R. Srinivasan, P.C. Murley, H.K. Tang. "Accurate, predictive modeling of soft error rate due to cosmic rays and chip alpha radiation", in *Proc. 32nd IRPS*, pp. 12-16, 1994.

[Stathis 2001] J.H. Stathis, "Physical and predictive models of ultra thin oxide reliability in CMOS devices and circuits", in *Proc. 39th IRPS*, pp. 132-150, 2001.

[Stott *et al.* 1998] D. T. Stott, G. Ries, M.-C. Hsueh and R. K. Iyer, "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection", *IEEE Transactions on Computers*, 47 (1), pp.108-119, January 1998.

[Strong *et al.* 1993] A.W. Strong *et al.*, "Gate dielectric integrity and reliability", in *Proc. 31st IRPS*, pp. 18-21, 1993.

[Sullivan & Chirallege 1991] M. Sullivan and R. Chillarege, "Software defects and their impact on systems availability – A study of field failures on operating systems", in *Proc. 21st IEEE Fault Tolerant Computing Symp., (FTCS-21)*, pp. 2-9, June 1991.

[Sullivan & Chirallege 1992] M. Sullivan and R. Chillarege, "Comparison of Software Defects in Database Management Systems and Operating Systems", in *Proc. 22nd IEEE Fault Tolerant Computing Symposium (FTCS-22)*, pp. 475-484, July 1992.

[Sunbelt 1999] Sunbelt International, "NT Reliability Survey Results", available at http://www.sunbelt-software.com/ntrelres3.htm, published on March, 23, 1999.

[Sylvester & Keutzer 1999] D. Sylvester and K. Keutzer, "Rethinking Deep-Submicron Circuit Design". *IEEE Computer*, Nov. 1999.

[Tammaro 2000] M.J. Tammaro, "The role of copper in electromigration: the effect of Cu-vanacy binding energy", in *Proc. 38th IRPS*, pp. 311-320, 1987.

[Tezaki *et al.* 1990] A. Tezaki, T. Mineta, H. Egawa, "Measurement of Three dimensional stress and modeling of stress-induced migration failure in aluminum interconnects", in *Proc. 28th IRPS*, pp. 209-215, 1990.

[Turner & Parsons 1982] T. Turner and R.D. Parsons, "A New Failure Mechanism: Al-Si Bond Pad Whisker Growth During Lifetest", *IEEE Trans. Comp. Hyb. Man. Tech.*, Vol. CHMT-5, pp. 431-435, 1982.

[Velpuri 1995] Velpuri, Rama, Oracle Backup and Recovery Handbook, Oracle Press / Osborne McGraw-Hill, Berkeley, California, 1995.

[Voas *et al.* 1997] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting how Badly 'Good' Software can Behave", *IEEE Software*, 1997.

[Walker 2000] M.G. Walker, "Modelling the wiring of deep submicron ICs", *IEEE Spectrum*, Vol. 27, No. 3, pp. 65-71, March 2000.

[Yount & Siewiorek 1996] C. R. Yount and D. P. Siewiorek, "A Methodology for the Rapid Injection of Transient Hardware Errors", *IEEE Transactions on Computers*, 45 (8), pp.881-891, August 1996.

[Ziegler 1998] J. Ziegler, "Terrestrial cosmic ray intensities", *IBM Journal of Research and Development*, Vol. 42, No. 1, Jan. 1998.