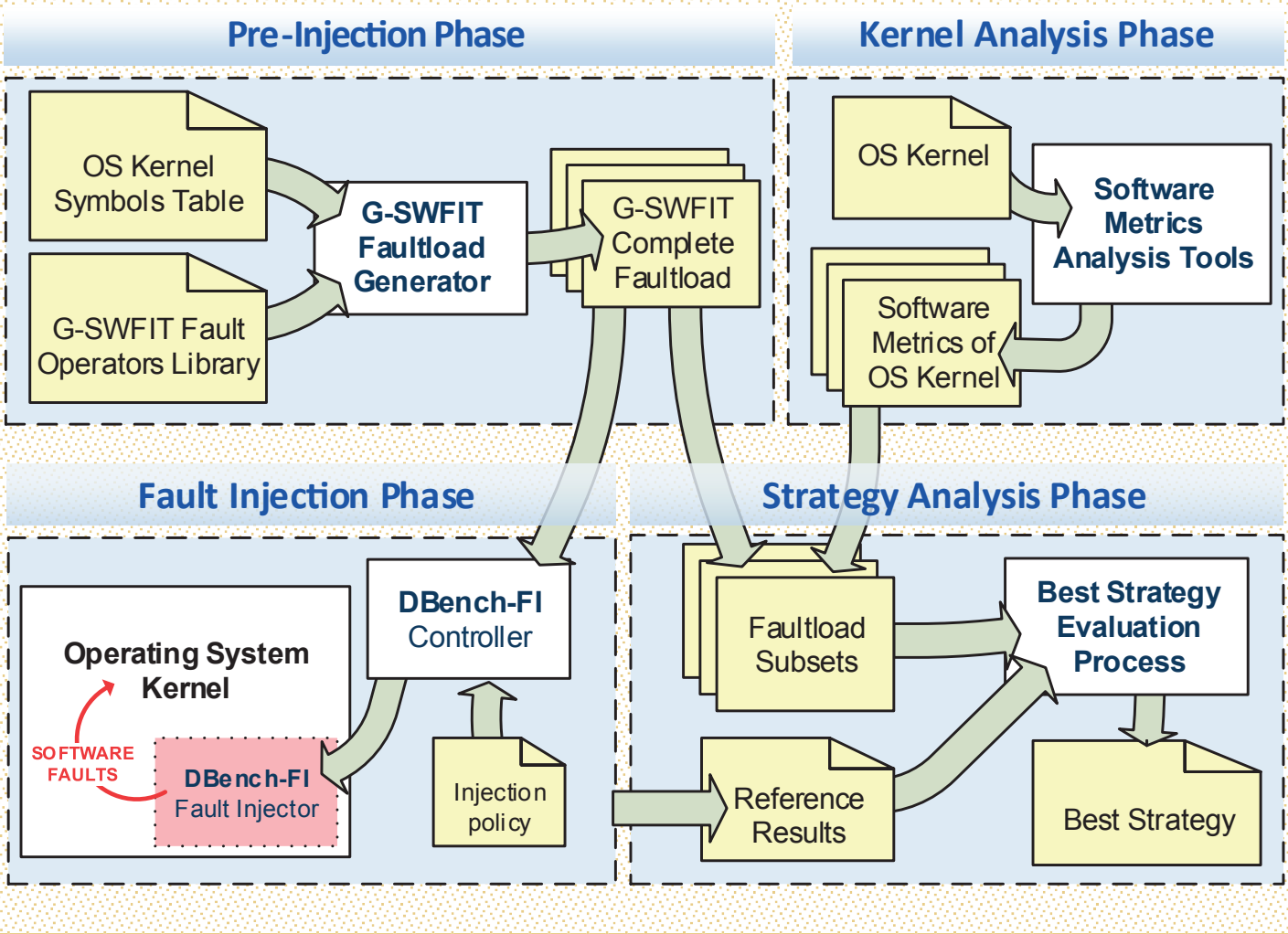


Pedro Miguel Lopes Nunes da Costa

DEPENDABILITY BENCHMARKING FOR LARGE AND COMPLEX SYSTEMS

Tese de doutoramento na área científica de Engenharia Informática, orientada pelo Senhor Professor Doutor João Gabriel Silva e pelo Senhor Professor Doutor Henrique Madeira, e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2013

· U C ·

UNIVERSIDADE DE COIMBRA

Thesis submitted to the
UNIVERSITY OF COIMBRA
for the degree of Doctor of Philosophy
in Informatics Engineering

Dependability Benchmarking for Large and Complex Systems

Pedro Miguel Lopes Nunes da Costa

Under supervision of

Prof. João Gabriel Silva
Dep. de Engenharia Informática
Universidade de Coimbra
Portugal

Prof. Henrique Madeira
Dep. de Engenharia Informática
Universidade de Coimbra
Portugal

Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra
Portugal

September 2013

Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Coimbra, Portugal – September 2013

- To my Wife, with love –

Abstract

The spread of computer-based systems and the growing number of its applications in critical tasks has increased the dependence of modern societies on that kind of systems. As a consequence, dependability benchmarking of computer systems, as a way to assess and compare the dependability of components and systems, has caught the attention of researchers and practitioners in recent years.

One crucial component of dependability benchmarks is the fault injector. Dependability benchmarks must include fault injectors with very specific features: (i) they should be very easy to install and use, without the need for any complex setup or installation procedure;(ii) have high level of portability; (iii) have very low intrusiveness, in order to mitigate the performance loss; (iv) be capable of injecting faults in both user and system spaces; (v) and in code and data segments of any process, irrespective of their complexity; (vi) be independent of the availability of the source code of any system component or user process; (vii) be dynamically linked into a target system; and (viii) be compatible with the latest and most advanced software fault models. Since existing fault injectors do not fulfill these requirements, this thesis presents a pioneering SWIFI tool named DBench-FI (Dependability Benchmarking Fault Injector), specially developed for dependability benchmarking. Their unique characteristics make it one of the most versatile fault injectors available.

Among the main components of a dependability benchmark suite, the most critical one is undoubtedly the faultload. It should embody a repeatable, portable, representative and generally accepted fault set. Concerning software faults, the definition of that kind of faultloads is particularly difficult, as it requires a much more complex emulation

method than the traditional stuck-at or bit-flip used for hardware faults. Moreover, a faultload based on software faults requires a clear separation between the software components which are selected as fault injection target and the benchmark target (i.e., the system under evaluation), as the injection of software faults actually changes the code of the target component. This way, the faults should be injected in one component (the fault injection target) in order to evaluate their impact in the other components or in the overall system, guaranteeing the inviolability of the benchmark target and the credibility of the dependability benchmark.

Although faultloads based on software faults had already been proposed, the choice of adequate fault injection targets (i.e., actual software components where the faults are injected) is still an open and crucial issue. Knowing that the number of possible software faults that can be injected in a given system is potentially very large (especially for large and complex systems), the problem of defining a faultload made of a small number of representative faults is of utmost importance. This thesis presents a comprehensive fault injection study and proposes a strategy to guide the fault injection target selection to reduce the number of faults required for the faultload. Furthermore, it exemplifies the proposed approach with a real web-server dependability benchmark and a large-scale integer vector sort application.

Acknowledgments

This thesis constitutes an important milestone in my life, for which I am indebted to all the people who made it possible.

First and foremost, I would like to thank my advisors for all the inspiration and support, despite the multiple tasks in which they are committed as a consequence of the important positions they hold in the University of Coimbra. I thank Professor João Gabriel Silva for the encouragement and for integrating me into the, now called, Software and Systems Engineering Group of the University of Coimbra. I am also truly thankful to Professor Henrique Madeira for his constant guidance, availability, valuable comments and stimulating discussions over the last years.

I would also like to express my gratitude to all who assisted me at different stages of this research. I am grateful to João Durães for the initial incentive and for providing me the G-SWFIT analysis tool. I thank José Luís Silva for his friendship and unsparing kindness aid on issues regarding operating systems administration. I would also like to thank MSquared Technologies and Testwell for providing me the versions of RSM and CMT++, respectively, and to ISCAC Coimbra Business School for all the facilities conceded in the context of the PROTEC/FCT doctoral grant.

I am grateful to my colleagues Manuel Castelo-Branco and António Gonçalves for the friendship and encouragement.

To all the friends that I did not mentioned, but who have been around these last years, encouraging and helping me, I would also like to express my gratitude.

Finally, this list would not be complete without mentioning my family. I want to thank my lovely wife Paula and my son Diogo, on whose constant encouragement, understanding and love I have relied on to overcome the hard times of this program. I am also deeply thankful to my mother and brother Carlos for their support and care. I would also like to give a posthumous thank to my father for all the good times we spent together.

Pedro Nunes da Costa

Coimbra, September 2013

Table of Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
BT	Benchmark Target
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DBench-FI	Dependability Benchmarking Fault Injector
DBMS	Database Management System
FIT	Fault Injection Target
GHz	Gigahertz
GiB	Gibibyte
G-SWFIT	Generic Software Fault Injection Technique
HWFI	Hardware Implemented Fault Injection
I/O	Input/Output
Kbps	Kilobits per second
LKM	Loadable Kernel Module
ODC	Orthogonal Defect Classification
OLTP	On-Line Transaction Processing
OS	Operating System
RAID	Redundant Array of Independent Disks
SUB	System Under Benchmark

SWIFI	Software Implemented Fault Injection
TPC	Transaction Processing Performance Council
TPC-C	Transaction Processing Performance Council Benchmark C
WMC	Weighted Methods in a Class
WS	Web-Server

Resumo em Língua Portuguesa

O aumento da utilização dos sistemas informáticos e o número crescente das suas aplicações em tarefas críticas das sociedades modernas tem aumentado a dependência desse tipo de sistemas. Em consequência, nos últimos anos, as *benchmarks* de confiabilidade têm sido objeto de enorme interesse, quer por parte de investigadores, quer por parte da indústria.

Um dos elementos fundamentais que integram as *benchmarks* de confiabilidade é o injetor de falhas. As *benchmarks* de confiabilidade devem incluir injetores de falhas com características muito específicas: (i) devem ser fáceis de instalar e de utilizar, não exigindo qualquer procedimento especial de instalação ou execução; (ii) devem possuir um elevado nível de portabilidade; (iii) devem possuir um baixo nível de intrusividade no sistema alvo, de forma a minorar a perda de desempenho; (iv) devem oferecer a capacidade de injetar falhas em todo o sistema alvo (quer no espaço do utilizador, quer no espaço do sistema); (v) assim como nos segmentos de código e de dados de qualquer processo, independentemente da sua complexidade; (vi) devem ser independentes da disponibilidade ou conhecimento do código fonte de qualquer componente do sistema ou processo de utilizador; (vii) ser dinamicamente integrados no sistema alvo; e (viii) ser compatíveis com os mais avançados e recentes modelos de falhas de software. Uma vez que os atuais injetores de falhas não satisfazem todos os requisitos mencionados, esta tese apresenta uma ferramenta de injeção de falhas pioneira, implementada por software (*Software Implemented Fault Injection - SWIFI*), denominada DBench-FI, especialmente desenvolvida para *benchmarks* de confiabilidade. As suas características únicas fazem dele um dos mais versáteis injetores de falhas atualmente existentes.

De entre os componentes fundamentais das benchmarks de confiabilidade (*workload*, *faultload*, medidas, e configuração experimental e procedimentos), a faultload é, sem dúvida, um dos mais críticos. Ela deve incorporar um conjunto de falhas repetível, portável, representativo e aceite pela comunidade e pela indústria. No que concerne a falhas de software, a definição desse tipo de *fautloads* é particularmente difícil, uma vez que exige métodos bastante mais complexos do que o tradicional *stuck-at* ou *bit-flip* utilizado nas falhas de hardware. Adicionalmente, as faultload baseadas em falhas de software exigem uma clara separação entre os componentes de software que são selecionados como alvo da injeção de falhas e o alvo da benchmark (i.e., o sistema sob avaliação), uma vez que a injeção de falhas de software altera efetivamente o código do componente alvo. Desta forma, as falhas devem ser injetadas num componente (o alvo da injeção de falhas) a fim de se avaliar o seu impacto nos outros componentes ou no sistema como um todo, garantindo a inviolabilidade do alvo da benchmark e a credibilidade das benchmarks de confiabilidade.

Apesar de terem já sido propostas faultloads baseadas em falhas de software, a escolha dos alvos da injeção de falhas (ou seja, os componentes de software onde as falhas são injetadas) continua a ser um tópico em aberto, apesar de fundamental. Sabendo-se que o número de falhas de software que podem ser injetadas num dado sistema é potencialmente muito grande, o problema da definição de uma faultload composta por um número pequeno de falhas representativas é de extrema importância. Esta tese apresenta igualmente um estudo exaustivo de injeção de falhas e propõe uma estratégia de orientação da seleção dos alvos da injeção de falhas para a redução o número de falhas necessárias numa faultload. Além disso, exemplifica a abordagem proposta com a utilização de uma *benchmark* de confiabilidade, real, para *web-servers* e de uma aplicação de ordenação de vetores de números inteiros de larga dimensão.

Chapter 1

Introduction

This thesis is the result of several years of research in the field of dependability benchmarking at the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra.

This opening chapter presents the motivation and the research goals for this work, providing a basis for the discussion that follows. The structure of the thesis is also presented in the final section of this chapter (Section 1.3).

1.1 Goal and Motivation

With the spread of computing systems and the growing number of its applications in our everyday life, modern societies are becoming increasingly dependent on computer-based systems. System failures are a serious risk and cause more damages than ever before. Although more serious consequences arise from failures in safety critical applications, such as medical, aircraft, and nuclear power systems, there are other areas where such system failures cause important damage, like financial losses.

There are many examples of system failures with consequent high costs in a wide range of areas. For example, in 1991, software problems in the Patriot missile-defense system used during the Gulf War prevented intercepting an Iraqi Scud missile killing 28 American soldiers and injuring around 100 other people [Blair *et al.* 1992]. Between June 1985 and January 1987, a race condition bug led to what became tragically known as the Therac-25 accident - a computer controlled radiation therapy machine that massively overdosed six people, with resultant deaths and injuries [Leveson *et al.* 1995]. On 26th and 27th November 1992, design fatal flaws caused the failure of the London Ambulance Computer Aided Dispatch system [THRA 1993]. The economic impact that a bug can have in a nation-wide money-critical system was fully shown in the credit card denial of authorization occurred in France, on 26th-27th June 1993 [Laprie 1995]. On 4th June 1996, a software problem caused the maiden flight explosion of Ariane 5 [Lions 1996], resulting in a direct loss of at least 370 million dollars to the European Space Agency (ESA) [Dowson 1997]. On 7th August 1996, inadequate redundancy [Garber 1996] led to the blackout of America Online (AOL) computer network, preventing the service provider's network for 19 hours, affecting 6 million users. On 14th August 2003, approximately 50 million people in the northeastern United States and southeastern Canada were impacted by the blackout of the General Electric energy management system [PSOTF 2004]. The outage was due to a software fault, triggered by a unique combination of events that led to a cascade of system failures and to an estimated total loss of 13 billion dollars [Wong *et al.* 2010]. On 7th March 2008, the reactor number 2 of the Edwin Irby Hatch nuclear power plant, in United States, was forced into an emergency shutdown for 48 hours after the installation of a software update on a computer operating on the plant's business network [Krebs 2008]. The resulting loss was estimated in 5 million dollars [Wong *et al.* 2010]. More recently, on June 2012, a software fault originated by a bad software upgrade caused the collapse of the Royal Bank of Scotland/Natwest computer banking system. As a consequence, several

million of costumers were unable to access their accounts for several days [Masters *et al.* 2012, Scott 2012]. The cost of this system failure was estimated in more than 100 million pounds [Treanor 2012].

The use of formal methods for software validation is many times rejected [DeMillo *et al.* 1979], since they encompass a too complex and time-consuming process that cannot be managed and used, in practice, in software development. Instead, many software engineers and designers argue the use of more elaborate testing methods in order to ensure the correctness of software. However, a counter-argument to this view is the fact that, as stated by [Dijkstra 1972], testing could only prove the presence of bugs, but not their absence.

In fact, it has been obvious over the last years that the high level of dependability, essential for modern computer systems, cannot generally be achieved using only a rigorous development process accepted by many of the actual certification schemes. The evaluation of dependability of computer systems is absolutely essential in an increasingly dependent society on that kind of systems. However, the intrinsic complexity of such an assessment is further aggravated by the growing complexity of both hardware and software [Silva *et al.* 2005]. Several research studies also show not only a clear predominance of software faults [Gray *et al.* 1991, Sullivan *et al.* 1992, Lee *et al.* 1995, Chou 1997, Kalyanakrishnam *et al.* 1999] when compared to other types of system faults, but also that its weight on the overall system dependability will tend to increase. As a consequence, it is nowadays generally accepted that most of the software components have residual faults, also known as software defects or bugs, which escape the traditional testing phases of software development process. Among the main causes for those circumstances, besides the well-known technical difficulties intrinsic to the software development and testing process [Lyu 1996, Musa 1996], one can mention the huge complexity of today's software and the increasing pressure to reduce time to market. This scenario emphasizes the importance of system dependability assessment as

a measure of confidence that can be relied on a given system. This includes the evaluation of attributes like availability, reliability, safety, integrity, among others. More than ever, practical approaches for the evaluation of the dependability of computer systems are very much needed, especially standardized dependability benchmarks that allow comparing dependability attributes of analogous and alternative software products or components. However, the experimental evaluation of the dependability of computer systems is very difficult [Carreira *et al.* 1995] as it depends on fault activation probability, which in turn depends on either internal or external system factors like the different layers of the software, the actual hardware where the software is running, environment issues, and human interaction.

After the success of the performance benchmarking initiatives that caught the attention of the industry in the last decades and have driven the creation of organizations like TPC (Transaction Processing Performance Evaluation Corporation) [TPC] and SPEC (Standard Performance Evaluation Corporation) [SPEC], dependability benchmarking has been the focus of attention of researchers and practitioners in the recent years [Kanoun *et al.* 2008, Brown *et al.* 2000, Vieira *et al.* 2003, Zhu *et al.* 2003a, Lightstone *et al.* 2003, Kanoun *et al.* 2001, Christmansson *et al.* 1996a, Durães *et al.* 2002a]. To many business critical systems and applications, dependability attributes like availability, integrity and reliability, among others, are as important as performance. The goal of dependability benchmarking is thus to provide a standard procedure specification to characterize a computer system or component, providing the assessment of dependability related measures.

The main components of a dependability benchmark suite are [Kanoun *et al.* 2008, Koopman *et al.* 1999a]:

- **Workload** – representing the work to be done by the system during the benchmark run and used to create a realistic operating

scenario. It should represent a typical operational profile for a particular application area.

- **Faultload** – representing a repeatable, portable, representative and generally accepted set of faults and stressful conditions that could lead to undependability, if not properly handled by the system.
- **Measures** - characterizing the performance and dependability of a system executing the workload in the presence of the faultload.
- **Experimental setup and benchmark procedures** - describing the setup required to run the benchmark and the set of procedures and rules that must be followed during the benchmark execution in order to ensure uniform conditions for measurement.

Among these components, one of the most critical and difficult to define is, doubtlessly, the faultload [Durães *et al.* 2004a], since it should represent a repeatable, portable, representative and generally accepted fault set. That difficulty is even higher in what concerns software faults, since they required a much more complex emulation method than the usual bit-flip fault injection approach used to emulate real hardware faults [Voas *et al.* 1997a]. Furthermore, a faultload based on software faults requires a clear separation between the software components that are selected as fault injection target and the benchmark target (i.e., system under evaluation), as the injection of software faults actually changes the code of the target component. This way, the faults should be injected in one component (the target) in order to evaluate their impact in the other components or in the overall system. In fact, the software faults injected in the target component actually allow answering the question of what would happen to the system if a residual fault in such component became activated.

A representative faultload must be one that contains faults that represents the common programming bugs that escape the traditional software testing phases and still persist in existent software products

[Durães *et al.* 2004b]. Although the faultload definition of that kind of faults had already been proposed [Durães *et al.* 2006], a problem still persists when that model is applied in very large and complex systems. Commonly, there is a large number of possible targets components for fault injection and, consequently, that represents a huge number of possible software faults to be injected.

In fact, the use of dependability benchmarks driven by software faultloads (e.g., such as the ones proposed in [Kanoun *et al.* 2008]) has a major problem: it could take years to inject the faultload, which means that, in practice, it is not possible to run such dependability benchmarks. This is the case when the target system is a large piece of software, such as an operating system. Reducing the size of the faultload (but keeping it representative enough to obtain valid results) is essential to show industry and the research community that it is possible to use dependability benchmarks in large-scale systems. It should be noticed that among the mentioned faultload properties (repeatability, portability and representativeness), the representativeness is the one that needs special attention when reducing the faultload. In fact, properties such as repeatability and portability of the faultload are either not affected by the reduction of the number of faults or it is even easier to satisfy those properties with a reduced faultload.

This thesis presents the results of more than two years of continuous fault injection experiments in real systems and proposes a strategy to answer a still open and crucial question: **how to choose adequate fault injection targets, and thus reducing the total software fault injection experiments, without restricting the benchmark scope and representativeness?**

This study is an attempt to answer this question. The presented work is based on an experimental study and incorporates the results of a three-year research effort focused on showing that it is possible to obtain

accurate fault injection using a faultload that contains only a small fraction of all the possible faults that can be injected in a target system.

1.2 Contributions

As mentioned, the aim of this thesis is to propose an approach to guide the fault injection target selection of dependability benchmarks, decreasing the execution time of the benchmark, maintaining, simultaneously, their usefulness and representativeness. This is especially useful in large and complex systems where the experimentation time can be highly reduced without compromising the dependability benchmark results. This method will open the possibility to extend the dependability benchmarks to those kinds of systems, making them feasible and applicable (such benchmarks usually take several months or even years to execute due to its large faultload size).

Within this context, the main contributions of the thesis are the following:

1. To provide a software fault injector compatible with the demanding requirements of dependability benchmarks. Namely, it should be very easy to install and use, have high level of portability and very low intrusiveness, be capable of injecting faults in both user and system spaces, and in code and data segments of any process, irrespective of their complexity, be independent of the availability of any source code of any system component or user process, be dynamically linked into a target system and be compatible with the latest and most advanced software fault models. Concerning this last requirement, it was considered essential the compatibility of the fault injector with the Generic Software Fault Injection Technique (G-SWFIT) [Durães *et al.* 2006] – the state-of-the-art in software faults model.

G-SWFIT is based on a set of operators for software fault emulation through low-level code changes in the target executable code, mimicking the most common types of real software faults. These operators resulted from a field study based on the analysis and classification of more than 600 software faults found in real software applications. The developed tool is one of the most versatile software fault injectors currently available.

2. To define and evaluate different hypothesis for the reduction of the number of software fault injection experiments. The evaluation is based on the analysis of the error obtained in consequence of the reduction of the fault injection experiments. This study uses the results obtained with a comprehensive faultload that includes all possible software target locations (the complete set of the kernel OS functions, referred in kernel symbols table), resulting in one of the most extensive fault injection studies ever reported.
3. To present a strategy to guide the fault injection target selection of dependability benchmarks and to reduce the required number of software faults, thus decreasing the execution time of the benchmark, maintaining, simultaneously, their usefulness and representativeness. The proposed methodology is especially useful in large and complex systems, where the experimentation time can be severely reduced without compromising the dependability benchmark results. Conducted experiments showed that the fault injection experiments can be reduced by more than 75%, maintaining the induced error below 1%. This method will open the possibility to extend the dependability benchmarks to large and complex systems, making them feasible and practicably applied (such benchmarks would take several months or even years to execute due to its large faultload size).

1.3 Thesis organization

The thesis is organized in seven chapters, as follows:

- **Chapter 1**, this chapter, presents the motivation for the undergone investigation, the research objectives and the contributions of the thesis.
- **Chapter 2** contains some terminology and the state of the art in dependable computing area that are relevant to this study. More specifically, it surveys previous relevant work in the dependability benchmarking, fault injection and software faults to the assessment and improvement of dependable systems. This chapter is especially oriented to the reader who is not familiar with the dependable computing area, so it can be skipped by knowledgeable readers.
- **Chapter 3** provides an overview of dependability benchmarking of software systems, its goals, components, general framework and challenges currently raised in this area.
- **Chapter 4** presents a software fault injector specially developed for dependability benchmarking – the *DBench-FI* (*Dependability Benchmarking Fault Injector*). It describes in detail its architecture, the corresponding modules and the way they interact with each other and with the user, besides a detailed presentation of its implementation.
- **Chapter 5** describes the problem that arises in assessing the dependability of large and complex systems, particularly with regard to software faultloads. It also presents and provides an early assessment of the experimental strategy followed in this work for the definition of compact and representative faultloads based on software faults.

- **Chapter 6** describes the experimental setup used to demonstrate the effectiveness of the proposed approach with two real and different systems: a web-server dependability benchmark and a large-scale integer vector sort application extended with performance and quality measures.
- **Chapter 7** concludes the thesis and indicates suggestions for future improvements and future research directions.

Chapter 2

Background and Related Work

This chapter introduces some basic concepts used in dependable computing systems and surveys the previous research works that are relevant to this study. This presentation of the pertinent terminology and of the state of the art includes the areas of dependability benchmarking and fault injection, with special emphasis to software systems, software fault injection and software faults.

This chapter is especially oriented to the reader who is not familiar with the dependable computing area. As a consequence, it can be skipped by knowledgeable readers.

2.1 Introduction

The increasingly dependency of modern societies on computer systems has brought a greater awareness of the importance of the dependability concept. Several examples of computer failures, like the ones mentioned in the previous chapter, show the catastrophic consequences of that dependence. Computer systems may result in costs to the society, in addition to the expected benefits [CASDCST 1992], for which they were developed. In this context, a new role of questions is raised: Can

we rely on computer systems? Are the computer systems dependable? What are the limits of that dependability?

The accuracy of the computational results has preoccupied systems programmers and their users since the first generation of computers (from the late 1940's to mid-1950). At that time, the use of unreliable components required the use of special techniques that allow the improvement of systems dependability. Among the used techniques, the error detection and correction, duplexing with comparison, triplication with voting and the diagnostics to locate failed components can be mentioned [Avizienis *et al.* 2000].

The growing use of computer systems in critical tasks of our society has increased the interest to develop systems that provide the expected service even in the presence of faults, known as fault tolerant systems. That need is even more obvious if we consider the adversity of the environment in which those systems sometimes operate and the fact that there are no perfect systems, that is, systems without any project or implementation defect. Moreover, the more complex a system is and the higher the number of its components, the higher is the probability of the occurrence of a failure in that system.

The level of confidence that can be relied on a service of a system is a determining factor in the characterization of that system, being fundamental in systems where human lives or substantial economic values are at risk. Dependability, together with functionality, performance, cost and security establishes the fundamental properties of computing and communication systems [Avizienis *et al.* 2004].

2.2 Basic concepts and definitions

2.2.1 Dependability

Dependability is defined in [Laprie 1985, Laprie 1995] as that property of a computer system such that reliance can justifiably be placed on the service it delivers. In this context, the delivered service is the behavior of the system, as it is perceived by its user - another system that interacts with the provider and receives the service [Avizienis *et al.* 2004].

However, to assess whether a system satisfies the requirements of dependability is not an easy task, especially when complex and large systems are involved. Moreover, that assessment is further hampered by the fact that dependability is a global concept which embraces a set of different attributes, whose emphasis and importance depends on the characteristics of the system or application being analyzed.

2.2.2 Attributes of dependability

As mentioned, dependability is an integrating concept which embraces a number of different, but complementary, attributes [Laprie 1995, Avizienis *et al.* 2004], that corresponds to different viewpoints of the system:

- **Availability** – concerning the readiness for correct service;
- **Reliability** – regarding the continuity of correct service;
- **Safety** – related to the absence of catastrophic consequences on the user(s) and the environment:
- **Confidentiality** – regarding to the non-occurrence of unauthorized disclosure of information;
- **Integrity** – related to the absence of improper system alterations;

- **Maintainability** – concerning the ability to undergo modifications and repairs;

Accordingly to the usual definitions, which consider it as a composite notion, security is not included as a single attribute of dependability [Avizienis *et al.* 2000]. Instead, **security** is considered as a combination of the mentioned attributes of confidentiality, integrity (concerning the absence of unauthorized system alterations) and availability (for authorized users only) [Avizienis *et al.* 2004]. Furthermore, dependability regarding to erroneous inputs is sometimes referred as robustness.

2.2.3 Impairments to dependability

According to [Jalote 1994, Clark *et al.* 1995], the first two attributes are, among all, the most relevant, given their importance on the fault tolerance capabilities of a system. However, the mentioned attributes of dependability may be emphasized in a greater or smaller extent, according to their importance on the application being analyzed. That importance should be considered in a relative or probabilistic rather than in an absolute or deterministic way, as the unavoidable presence or occurrence of faults prevents the existence of totally available, reliable, safe or secure systems [Avizienis *et al.* 2000].

In [Laprie 1995], faults, errors and failures are defined as the impairments to dependability. A **fault** is a defect that potentially causes an error. That is, the cause of an error is a fault. Although a fault has the potential to generate errors, those errors may not occur during the observation period. In other words, the presence of faults does not guarantee the occurrence of an error. However, the reverse is true: an error in a system state always involves the presence of a fault in that system. A fault that produces an error is said to be **active**. Otherwise, it is **dormant**. An **Error** is the part of the system state (altered by a fault) that is liable to

cause a subsequent failure. An error is a manifestation of a fault. Undetected errors in a system are said to be **latent**. A system **failure** occurs when the system does not comply with its specification, that is, when the system does not provide the expected service. Failures are caused by errors. If an error exists in a system state, then, unless some corrective measures are taken, there is a sequence of actions that can be performed and that could lead to a failure. A failure in a system does not always reveal the same way. Different forms of failures that can occur in a system are called types of failures or failure modes. The mentioned cause-effect relationship among these impairments, as described in [Jonhson 1989, Avizienis *et al.* 2004], can be represented as depicted in Figure 2-1.

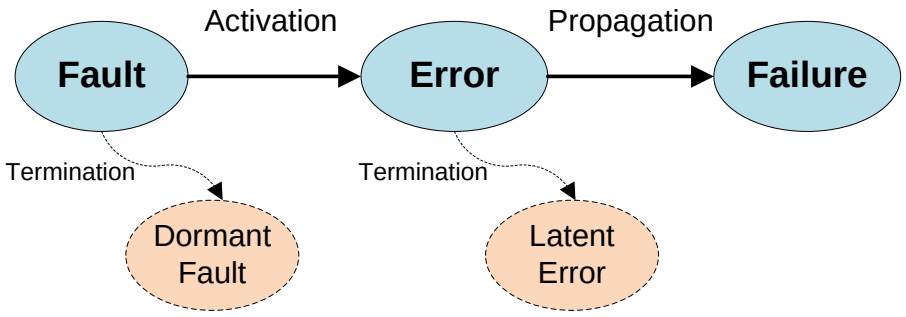


Figure 2-1 – Relationship between fault, error and failure.

Faults can be classified according to several factors or viewpoints [Laprie 1992, Laprie 1995, Laprie 1998, Avizienis 2004]. In the context of this work, two viewpoints deserve a special emphasis, among all other: phenomenological cause and persistence. Concerning phenomenological cause, the faults can be classified in:

- **Physical faults** – faults caused by physical phenomena, internal or external to the system.
- **Human-made faults** – faults that result from human action, either design faults, when committed during the system design and development phases, or operational faults, when due to input or operating conditions violation.

Relating to persistence, a fault can be considered in one of the following categories [Carreira *et al.* 1999, Koren *et al.* 2007]:

- **Permanent Faults** – occur in a continuous and stable mode in time. Concerning hardware, a permanent fault means an irreversible damage that can only be recovered through the repair or the replacement of the faulty component;
- **Intermittent Faults** – faults whose presence is limited in time, caused by unstable hardware, or varying hardware or software states. This kind of faults can be repaired by replacement or redesign of the hardware or software;
- **Transient Faults** – faults that are caused by temporal environmental conditions like, for example, electromagnetic interference, or radiations.

The main difference between intermittent and transient faults[1] is that the latter cannot be repaired, since neither the hardware nor the software is

---

[1] In the literature, the transient and intermittent bugs are sometimes referred as Heisenbugs, because they disappear when reexamined (in analogy to the Heisenberg Uncertainty Principle). By contrast, the permanent faults are referred to as Bohrbugs, as they represent good solid bugs, which are easy to diagnose upon detection (in analogy to the Bohr Atom Model). In recent taxonomies of software faults [Grottke *et al.* 2007], the Heisenbugs are

damaged [Siewiorek *et al.* 1992]. According to several studies, the transient faults occur much more frequently than the permanent faults and are also much more difficult to detect [Carreira *et al.* 1998a, Carreira *et al.* 1999, Clark *et al.* 1995].

2.2.4 Improving dependability

The dependability of a system is defined by the dependability of hardware and software that constitutes it. The development of dependable systems requires, according to [Avizienis *et al.* 2000, Avizienis *et al.* 2004], the combined use of four techniques: fault prevention, fault tolerance, fault removal and fault forecasting.

Fault prevention is the ability of avoiding the occurrence or introduction of faults in a system. Thus, it can be considered as the initial defensive mechanism towards dependability. It is attained by applying quality control techniques during the system design and development phases. General approaches include formal methods in requirement

---

classified as a type of Mandelbugs (alluding to Benoît Mandelbrot, a leading researcher in fractal geometry) - a more general class of bugs, characterized by having complex and obscure causes, making their behavior appear chaotic or even non-deterministic.

specifications and rigorous testing of all system components and their interactions. Regarding software, it consists in good programming principles and environments (structural programming, modularization and formal verification techniques), whereas for hardware, it involves rigorous design rules (design reviews, component screening and testing). External faults such as lightning or radiation can be prevented by shielding, radiation hardening, etc. User and operation faults can be reduced by training and regular maintenance procedures.

Fault tolerance aims to provide the systems the capability to deliver the correct service in the presence of faults (as represented in Figure 2-2). Obviously, fault tolerance assumes that fault prevention is not enough to eliminate all the possible faults in a system and, consequently, any system has some probability to have or is likely to develop a fault. That probability is even increased if we consider that it is impossible to eliminate all the environment aspects susceptible to change the system proper operation. Fault tolerance mechanisms are implemented using redundancy, error detection and subsequent system recovery mechanisms. A redundant system can mask a failed component with a redundant one and continue to operate without any service interruption, or at least, with the minimal interference on the external behavior, since the recovery mechanisms may cause some performance degradation. It should be noticed that fault tolerance is a recursive concept. That is, it is essential that the mechanisms which implement fault tolerance are themself protected against the faults that may affect them.

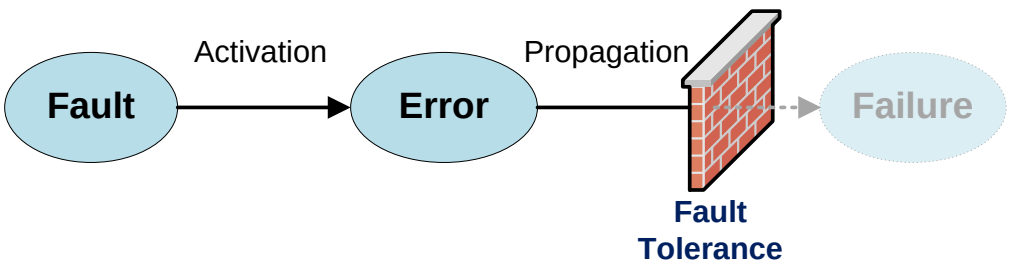


Figure 2-2 – Fault tolerance mechanisms.

The fault tolerant architectures are presently used in a wide range of applications, from safety critical to commercial ones. For example, concerning safety critical systems, fault tolerant architectures are used in the flight control computers of the fly-by-wire systems of the Boeing 777 and AIRBUS A320/A330/A340 airplanes [Torres 2000].

Fault removal aims to reduce the number or the severity of the faults and may be performed during both the development and operational phases of a system. During the development phase, fault removal consists in verification, diagnosis and correction [Avizienis *et al.* 2004], usually done by debugging, and/or simulation of hardware and software. Fault removal during the operational is conducted by maintenance techniques, corrective or preventive. At this phase, faults can be removed replacing the faulty system components or by software updates.

Fault forecasting predicts possible faults in order to prevent or avoid them or to limit their effects. This is accomplished by performing an evaluation, qualitative and quantitative, of the system behavior, with respect to fault occurrence or activation. This evaluation is commonly achieved using modeling and simulation of the system and faults. Qualitatively, it comprises the probabilistic evaluation of some attributes of dependability, interpreted as dependability measures. Quantitatively, it consists on the identification, classification and ranking of failure modes or event combinations that are liable to lead to system failures.

The inclusion of all these four techniques should be analyzed earlier in the project phase of the systems, since it is very difficult to apply them in systems where dependability issues were not taken into consideration. Moreover, depending on the emphasis assigned to each dependability attribute, according to the specificities of each application, there must be a balanced use of these techniques. This trade-off is even more difficult as conflicts may exist between some dependability attributes, such as availability and security [Avizienis *et al.* 2000].

The relation between dependability, its attributes, impairments and means can be represented in a single schema as exposed in Figure 2-3.

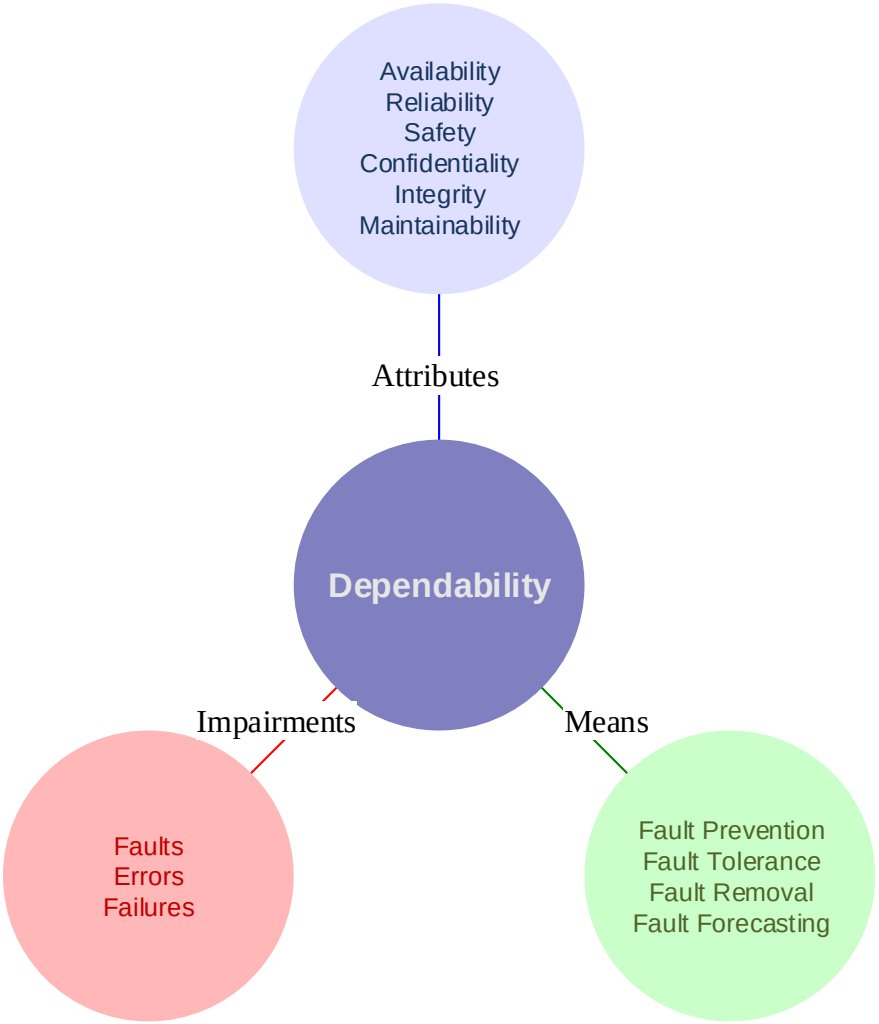


Figure 2-3 – The taxonomy of dependability.

2.3 Dependability benchmarking

The direct evaluation and comparison of performance of systems, concerning some of its characteristics like performance and functionality, have long driven the computer industry to improve their products.

However, very often, the systems and configurations are optimized in order to achieve the best performance and do not represent the real systems used in field [Vieira *et al.* 2009]. This way, these performance-oriented configurations tend to characterize unrealistic scenarios, as they disregard dependability-related aspects that are required by many modern computer systems. In fact, recently, factors like dependability and maintainability of systems are also seen as very important. However, unfortunately, while there are different ways to evaluate and compare different systems and components, regarding its performance and functionality, the evaluation of the dependability attributes of a system turns out to be much more difficult. One of the main difficulties is related to the existence of a wider spectrum of measures in dependability benchmarks, when compared to performance benchmarks.

The need of tools to evaluate and compare the dependability of systems is nowadays reinforced by the current trend of using commercial off-the-shelf (COTS) components and of COTS-based systems with high dependability requirements, as a way to reduce costs and shorten the development and deployment times. In fact, it is important to note that the increase of confidence in the general dependability of COTS, induced by its large-scale use, may not constitute a sufficient condition for its use in critical applications. In addition to the faults that those components may have, the COTS software components are developed without the knowledge of the specific context in which they will be used and are usually provided as a black box, mostly without a rigorous written specification [Guerra *et al.* 2004]. The integration of such components into computer systems creates additional dependability challenges that demands tools capable of evaluating and comparing the dependability attributes between systems.

According to [Madeira *et al.* 2001], dependability benchmarks should provide a generic, cost-effective and reproducible way for evaluating the behavior of components and computer systems in the presence of faults,

allowing the quantification of dependability attributes, seen as measures, or the characterization of systems into well-defined dependability classes. It is important to note that some fault tolerance mechanisms may inflict a performance overhead in the systems, which is also interesting to evaluate. Indeed, a timely and correct service delivery, concerning the system specification, is of utmost importance, mainly in hard real-time systems.

Furthermore, in addition to the characteristics of the dependability evaluation and validation techniques, a dependability benchmark should represent an agreement that is accepted by the computer industry and/or by the user community. Dependability benchmarks are, obviously, of utmost importance to complex, mission critical systems and for high-end business-critical applications. Moreover, they may also play a broader key role in the computer systems area, driving the industry to produce better systems, similarly to what happened before, in the performance and database areas.

2.3.1 Reference model

Dependability benchmarks are generally based in modeling or experimentation, or both. The modeling approaches include analytical [Trivedi *et al.* 1994] and simulation models [Rimén *et al.* 1993], and are generally used to support architectural decisions at design phase. They require the knowledge of the system functions and architecture, in terms of system components and their interactions, namely in what concerns to the fault tolerance and recovery mechanisms used to increase the system dependability. This knowledge is used to build a representation of the system, in order to model the system behavior and to analyze events and activities like failure occurrences, error detection and propagation, system recovery, etc. Those events and activities, characterized by event rates and conditional probabilities of success or failure, known as model parameters, are then used to analyze the system dependability. Block diagrams, faults

trees, Markov chains or stochastic Petri nets are examples of modeling techniques used for dependability modeling of computing systems. The required allocation of numerical values to the model parameters, such as coverage factor and restart times, are usually based on experimental measurement, field data or past experience related to similar systems. It is worth noting that the modeling approach may be unfeasible for large and complex systems, since systems made of many components with several dependencies usually lead to high complex models [Kanoun *et al.* 1996]. However, for some COTS-based systems, in particular to those systems whose architecture is not known in detail, the modeling approach can be used to produce, with a reduced effort, simple high-level models.

On the other hand, experimental approaches are used in computer prototypes or actual systems in order to evaluate the effectiveness of the fault tolerance mechanisms and to characterize the system in the presence of faults. They are usually obtained from observation of the system in real field operation [Gray 1990], also known as field measurement, or through the execution of benchmark controlled experiments, based on fault injection techniques [Hsueh *et al.* 1997, Carreira *et al.* 1995, Clark *et al.* 1995, Madeira *et al.* 2000, Moraes *et al.* 2007]. Field measurement is based on data collected on the system and its environment, concerning failures, fault tolerance and maintenance processes: time to failure occurrences, nature of failures, impact on system services, recovery time, etc. This data allow the evaluation of measures such as mean time between failures (known as MTBF), failure rate, system availability, etc. Field measurement can also be used to feed data into the design of new systems, avoiding the weaknesses found in the previous systems and enhancing the dependability of the new ones. However, since fault occurrence constitutes rare events, the execution of fault injection based experiments is usually used as a practical way to accelerate the characterization of the system faulty behavior. It consists on the deliberate introduction of artificial faults in a system or component, through the use of a workload and a faultload, in order to assess its

behavior in the presence of faults, and to obtain the relevant dependability measures [Arlat *et al.* 2002, Vieira *et al.* 2003, [Durães *et al.* 2004b, Kanoun *et al.* 2006] and characterize the system.

A reference model for implementing dependability benchmarks is represented in Figure 2-4.

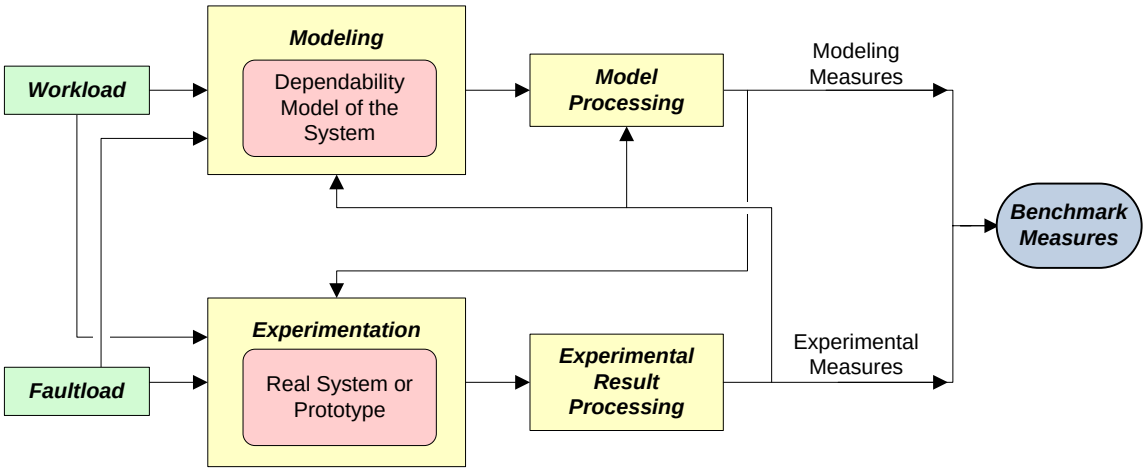


Figure 2-4 – Reference model for implementing dependability benchmarks [Kanoun *et al.* 2008].

Despite each one of the mentioned approaches (modeling or experimentation) has its advantages and limitations, results based on the observation in real field data are naturally more significant that those based on modeling or prototypes. Nevertheless, as mentioned, such kind of analysis is usually impracticable, as it requires the collection of data related to very specific and rare data events, and hence requiring a long period of system observation in order to get statistically significant results. As a result, the great majority of dependability benchmark proposals presented so far is experimental and based on fault injection techniques, allowing the execution of benchmark controlled experiments.

2.3.2 Dependability benchmark properties

According to [DBENCH 2004], in order to be useful, cost-effective and accepted by the computer industry and user community, an experimentation based dependability benchmark should satisfy a set of properties:

- **Representativeness** – important in all benchmarking dimensions, representativeness is of special relevance in measures, workload and faultload. Measures should be meaningful to the benchmark context in order to attain the expected usefulness of the benchmark. The workload should represent a typical and realistic set of activities found in real systems in the benchmark, being, therefore, dependent of it. The faultload should represent a set of real faults that may affect the target system in real use. The definition of the faultload should also consider the context of the application area and the operating environment.
- **Repeatability and Reproducibility** – concerning the guarantee that statistically equivalent results are obtained when the benchmark is run more than once in the same environment, i.e., the same System Under Benchmark (SUB), with the same workload and faultload and with the same prototype. Reproducibility assures that statistically equivalent results are obtained by different teams when the benchmark is implemented from the same specifications and is used in the same SUB.
- **Portability** - concerning the ease of transfer among various target systems, within a particular application area. This property allows the benchmark to compare computer systems and components. The portability is very dependent on the specification of some key benchmark components like faultload and workload. For example, the lack of portability of the faultload can limit the portability of the benchmark.

- **Non-intrusiveness** – regarding the changes that the benchmark inflicts on the SUB, which should be as small as possible. In order to avoid intrusiveness on the Benchmark Target (BT), faults must be injected only in components of the SUB outside the target of the benchmark. Thus the non-intrusiveness is guaranteed with regard to the BT.
- **Scalability** – concerning the capability of the benchmark to evaluate systems of different sizes. The scaling rules of the benchmark specification typically affect its workload and faultload. It is worth noting that very large faultloads may also require, as large workloads do, a huge time to execute the benchmark process. This circumstance constitutes a major limitation to execute dependability benchmarks in very large systems.
- **Benchmarking time and cost** – regarding the time and cost needed to obtain the result from the benchmark. This property embodies the usability that a benchmark should have. The benchmark time comprises not only the execution time of the benchmark, but also the time needed for the setup and preparations and for data analysis. A dependability benchmark should take the minimum time possible, preferably only a few hours per system (in very large systems may be acceptable to have a benchmark time of a few days). With regard to the cost, the user perceived value of the benchmark should be higher than the cost associated to its execution, as a key objective of dependability benchmarks is to provide a cost-effective way to characterize the dependability of components and computer systems.

All these properties should be considered not only in the specification phase, namely, in the definition of the measures and

experimental dimensions, but also in the implementation and validation phases of the benchmark development process.

2.3.3 Dependability benchmark proposals

Dependability benchmarking has caught researchers' attention in the last years and many dependability benchmarks have been proposed for different application domains.

With the aim of promoting the research, practice adoption of dependability benchmarks, the IFIP (International Federation for Information Processing,) and, particularly, the 10.4 Working Group on Dependable Computing and fault Tolerance, created, in 1999, the Special Interest Group on Dependability Benchmarking (SIGDeB). The resulting work, merging the contributions from both academia and industry, has identified a set of standardized classes to characterize the dependability of computer systems [Wilson *et al.* 2002]. The work carried out aimed to allow the comparison of computer systems concerning four dimensions: availability, data integrity, disaster recovery and security. Complementary work was developed in the context of the DBench project[2] - a European

[2] Dependability Benchmarking Project, IST-2000–25425 DBENCH [DBENCH].

project on dependability benchmarking, partially supported by the European Commission.

The work done in SIGDeB and in project DBENCH marked the beginning of several proposals of dependability benchmarks for various kinds of systems. Due to the huge diversity of applications and systems in the computer industry, several dependability benchmarks have been developed for different application areas and systems (e.g., general purpose operating systems, real-time kernels, engine control applications, on-line transaction processing systems). However, they all share the properties presented, at least at an abstract level, and constitute an instantiation of it to a specific domain or a particular computer system.

A general methodology for benchmarking the availability of computer systems was introduced in [Brown *et al.* 2000]. This work uses fault injection to cause situations where software RAID (Redundant Array of Inexpensive Disks) systems availability may be compromised. It adopted the workload and performance measures from existing performance benchmarks.

An attempt to incorporate human behavior in dependability benchmarks and system designs as a way to incorporate effects of a human operator in dependability measures is presented in [Brown *et al.* 2001]. In [Brown *et al.* 2002] is presented a methodology for developing dependability benchmarks that capture the impact of human operators on systems. The proposal adopts the workload and the performance measures of existing performance benchmarks. The systems dependability is characterized by the performance degradation induced by the injected faults and by the perturbations generated by human operators. Research work towards the development of a dependability benchmark for human assisted recovery processes and tools in server systems is presented in [Brown *et al.* 2004a]. The proposed methodology, developed at the University of California-Berkeley, aims to evaluate human-assisted failure recovery tools and processes and

can be used both to quantify the dependability of recovery systems and to compare different recovery approaches.

A practical characterization and comparison of COTS operating systems behavior in the presence of faulty device drivers is presented in [Durães *et al.* 2002a, Durães *et al.* 2003a]. This work is based on the emulation of high level real software faults through the modification of the ready-to-run binary code of the target software module, and proposes the use of a multidimensional perspective to evaluate different views of the benchmark results. The used fault emulation technique, named G-SWFIT, requires the existence of a library containing the complete set of code mutations, previously defined for the target platform, formerly scanned. A similar study proposing a practical approach to characterize the robustness of operating systems with respect to faulty drivers is presented in [Albinet *et al.* 2004]. In this work a Software Implemented Fault Injection (SWIFI) technique is used to corrupt the parameters of the interface between the device drivers and the kernel of the OS. In order to characterize the faulty behaviors, both internal (kernel error codes) and external measurements (e.g., raised exceptions, kernel hangs, and workload behavior) were considered.

A comparison of fifteen commercial OS POSIX (Portable Operating System Interface) implementations concerning their robustness was first presented in the context of the Ballista Project, from Carnegie Mellon University [Koopman *et al.* 1999b]. A dependability benchmark comparison of three operating systems (Windows NT4, 2000 and XP) focused on robustness and with respect to erroneous inputs provided by the application software to the Operating System via the Application Programming Interface (API) is proposed in [Kalakech *et al.* 2004]. The workload used in this dependability benchmark was the TPC-C performance benchmark for transactional systems [TPCC], an already well-established and agreed benchmark. A similar dependability benchmark and its application to six versions of Windows operating

system and four versions of Linux operating system is presented in [Kanoun *et al.* 2005]. The workload used in this study was the PostMark, a file system performance benchmark for operating systems [Katcher 1997]. Concerning the faultload, this work mainly considers corrupted parameters in the Operating System (OS) system calls. In [Kanoun *et al.* 2006], a dependability benchmark for general-purpose operating systems is proposed, considering analogous faultload, and presented its application in several versions of windows and Linux operating systems. The workload used in this study is the JVM (Java Virtual Machine) and the benchmark measures considered are the OS robustness and the OS system reaction and restart times in the presence of faults.

At IBM, the Autonomic Computing Initiative [IBMACI] aims to develop a suite of benchmarks to quantify the autonomic capacity of a system, which is defined as the capability of the system to react autonomously to problems and changes in the environment. This self-managing capability should incorporate four fundamental features: self-configuration, self-healing, self-optimization, and self-protection [Ganek *et al.* 2003]. A first discussion on the requirements of those benchmarks and a proposal of a set of metrics for the evaluation of a systems autonomic level is presented in [Lightstone *et al.* 2003]. In [Brown *et al.* 2004b] are presented the main challenges and pitfalls about benchmarking the autonomic capabilities of a system. This work proposes that autonomic benchmarks must quantify four dimensions of a system autonomic response: (i) the level of response; (ii) the quality of the response; (iii) the impact of the response on the system user; and (iv) the cost of any extra resources needed to support the autonomic response. A configuration complexity benchmark, process-based, that generates metrics that reflect the level of human involvement in the systems configuration process is presented in [Brown *et al.* 2004c]. In [Brown *et al.* 2005] is presented a benchmark for assessing the self-healing dimension of the autonomic capability. In this work, the system self-healing capabilities were

quantified with two metrics: (i) a measure of how effectively the system under test heals itself in response to the injected disturbances; and (ii) a measure of how autonomic that healing response is.

A preliminary proposal of a dependability benchmark for real time kernels for onboard space systems is presented in [Moreira *et al.* 2003]. This work focuses mainly on the assessment of the predictability of response time of service calls in a Real-Time Kernel (RTK) used in space domain systems. The benchmark, called DBench-RTK, uses an Onboard Scheduler (OBS) process as workload and its faultload consists of a set of faults that is injected into kernel functions calls at the parameter level by corrupting parameter values.

A dependability benchmark for OLTP (On-Line Transaction Processing) application environments is proposed in [Vieira *et al.* 2003]. This benchmark uses the workload of the TPC-C benchmark [TPCC] and specifies the measures and all the steps required to evaluate both the performance and dependability features of OLTP systems, with emphasis on availability. This study uses as faultload, a set of operator faults that emulates real faults experienced by OLTP systems in the field. Another dependability benchmark for transactional systems is proposed in [Buchacker *et al.* 2003]. Although this study also adopted the workload from the TPC-C performance benchmark, it considers a faultload based on hardware faults.

Research work at Sun Microsystems proposes a high-level framework specifically dedicated to availability benchmarking of computer systems [Zhu *et al.* 2003a]. The proposed approach decomposes availability in three key components: fault/maintenance rate, robustness, and recovery. Within the scope of that framework, two dependability benchmarks were developed: one that measures specific aspects of a system robustness on handling maintenance events, such as the replacement of a failed hardware component or the installation of a software patch [Zhu *et al.* 2003b]; and a

second benchmark for measuring system recovery on a non-clustered standalone system [Mauro *et al.* 2004].

A dependability benchmark for engine control applications to allow the characterization of the impact of faults in on the control software embedded in engine Electronic Control Units (ECUs) is presented in [Ruiz *et al.* 2004]. This benchmark, based on the injection of transient hardware faults in the ECU, provides a set of measures that allows a comparison of two different diesel engine control systems concerning its safety. The workload used is based on the Europe standards for the emission certification of light duty vehicle.

A dependability benchmark based on the injection of software faults was first proposed in [Durães *et al.* 2004a]. This benchmark uses the G-SWFIT technique (Generic Software Fault Injection Technique) in order to directly inject mutations at machine-code level that emulate high-level software faults [Durães *et al.* 2002b]. The inserted modifications reproduce the code that would have been generated by the compiler if the intended software faults were in the high level source code. A complete dependability benchmark for web-servers that also uses the G-SWFIT technique is proposed in [Durães *et al.* 2004b]. Adopting the workload and the performance measures of SPECWeb99 performance benchmark [SPEC], the benchmark uses a faultload that emulates both a realistic software defects and the effects of hardware and operator faults.

A study at Intel Corporation has focused on benchmarking semiconductor technology [Constantinescu 2005a]. The work discusses the impact of semiconductor technology scaling on neutron induced Soft Error Rate (SER) and presents an experimental methodology and results of accelerated measurements carried out on Intel Itanium® microprocessors. The work can be used as a dependability benchmark, as the used approach does not require any proprietary data about the microprocessor under evaluation. Relying on environmental test tools, Intel Corporation has also developed a set of benchmarks that allow the benchmarking of undetected

computational errors, also known as Silent Data Corruption (SDC) [Constantinescu 2005b]. This study performs a temperature and voltage operating test (the so-called Four Corners Test) on several prototype systems.

Three analytical dependability benchmarks that examine the Reliability, Availability, and Serviceability (RAS) characteristics of computer systems were developed at Sun Microsystems [Elling *et al.* 2008]: the Fault Robustness Benchmark (FRB-A) allows the evaluation of the robustness techniques used to enhance systems resiliency, including redundancy and automatic fault correction; the Maintenance Robustness Benchmark (MRB-A) allows the evaluation of how the maintenance activities affect the ability of the system to provide a continuous service; and the Service Complexity Benchmark (SCB-A) allows the evaluation of the complexity of servicing mechanical components of computer systems.

A dependability benchmark intended to evaluate the robustness of partitioning mechanisms of real-time operating systems is proposed in [Barbosa *et al.* 2010]. The benchmark includes both hardware-based and software-based faultloads and measures the spatial and temporal isolation among tasks.

A software framework for assuring system dependability based on benchmark scenarios and quantitative measures is presented in [Fujita *et al.* 2012]. The DS-Bench toolset performs benchmark test on the target system and obtains dependability metrics using various benchmarks programs and anomaly generators.

Two different approaches for extending TPC benchmarks with dependability measures are presented and discussed in [Almeida *et al.* 2010]: extending each TPC specification in a customized way; and, a more unified approach, defining a generic and independent specification that could be applied to any TPC benchmark. The advantages and disadvantages of each approach are also presented.

A proposal for the integration of dependability benchmarks into the recent ISO/IEC 25045 standard [ISOIEC 2010][3] is presented in [Friginal *et al.* 2011]. The approach provides the standard with the ability to assess the eventual impact of faults (referred as disturbances in the standard) on the quality of software components. The effectiveness and usefulness of the approach is demonstrated using three distinct different versions of Optimized Link State Routing (OLSR) as software components.

However, despite the great efforts in the last decade in developing a vast variety of dependability evaluation methods and techniques, dependability benchmarks do not benefit yet from the level of maturity, recognition and consensus of the well-established area of performance benchmarks, which is supported by major companies in the computer industry and where TPC and SPEC play a key role.

---

[3] The ISO/IEC 25045 is an extension of the ISO/IEC Systems and software Quality Requirements and Evaluation (SQuaRE) standard [ISOIEC 2005] in order to incorporate the viewpoint of recoverability into the procedures for evaluating the quality of software components.

2.4 Fault injection

As mentioned in the previous section, one of the experimental methods used for dependability evaluation consists of analyzing the behavior of a system from the real field operation and collecting the information about its dependability, known as measurement-based analysis. Despite the advantage of allowing the identification of the failures and faults that more frequently occur in a system, this method requires the collecting of data over a long period of time, due to the infrequent occurrence of errors and failures observed in systems with high dependability levels. Factors such as the mentioned long time between failures, the destructive nature of a crash or the long error latency, make it difficult to identify the causes of failures in the system operational environment. Moreover, it is particularly difficult to recreate a scenario of failures in large and complex systems.

The fault injection technique, also using an experimental approach, allows to overcome these drawbacks, by carrying out controlled experiments where the observation of the behavior of the system in the presence of faults is explicitly induced by the deliberate introduction (injection) of faults in the system [Arlat *et al.* 1990a]. Its recommendation by leading safety standards like NASA standard 8719.13B for software safety [Nasa 2004] and the ISO/DIS 26262 standard for automotive safety [ISODIS 2009], and its wide use over the last decades by many providers (e.g., ESA, IBM, Intel, Siemens, Sun, Volvo, etc.) and by the practitioners of dependable computer systems demonstrates the relevance of the method. Recently, reinforcing that pertinence, the fault injection technique was also included in the ISO/IEC Systems and software Quality Requirements and Evaluation (SQuaRE) standard [ISOIEC 2005] as a disturbance injection methodology for the assessment of the recoverability of software systems, through the evaluation module ISO/IEC 25045 [ISOIEC 2010].

According to [Hsueh *et al.* 1997], a typical fault injection environment consists of the following components, as shown in Figure 2-5:

- **Target system** – system in which the faults are injected, as it executes the tasks submitted by the workload generator.
- **Fault Injector** – component responsible for the injection of faults in the target system. It could be implemented by hardware (HWIFI) or software (SWIFI) and it can support different fault types, fault locations and fault injection triggers.
- **Fault Library** – Contains information about the type, location and number of faults, as well as of hardware semantic or software structure used by the fault injector. It should be considered a separate component in order to attain greater levels of flexibility and portability.
- **Workload Generator** – Component responsible for the workload generation that is executed by the target system.
- **Workload Library** – Contains information about the workload executed by the target system. May contain applications, benchmarks or synthetic workloads. Like the fault library, and for analogous reasons, it should be considered separated from the workload generator.
- **Controller** – program that controls the fault injection experiments. It can be executed either on the target system or on a separate computer.
- **Monitor –** Tracks the execution of the commands and initiates the data collection whenever necessary.
- **Data Collector –** Performs the online collection of the experiments data.

- **Data Analyzer –** Performs, eventually offline, the processing and analysis of the collected data.

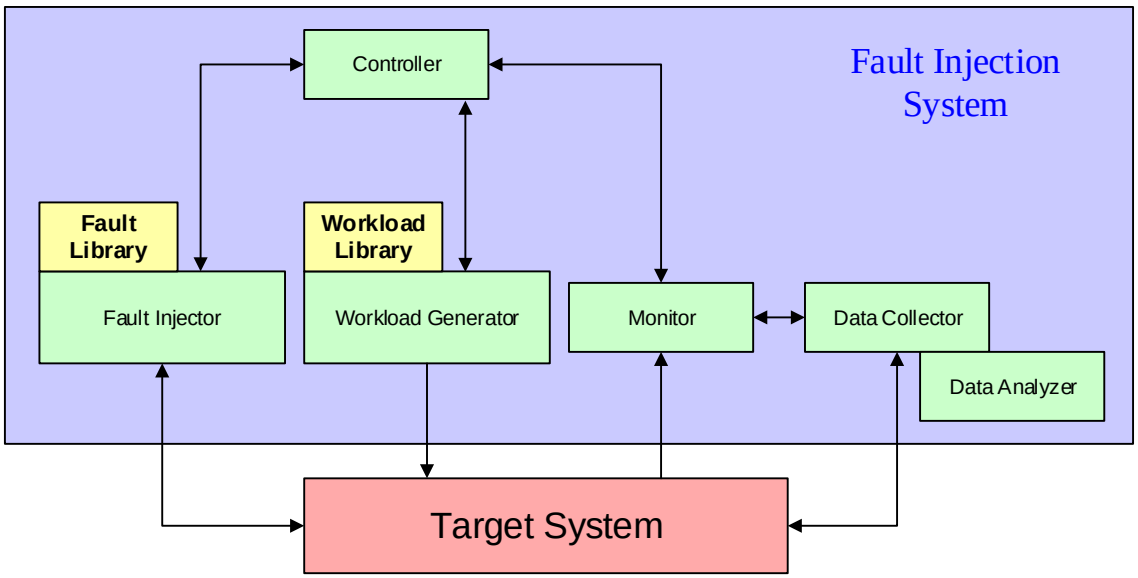


Figure 2-5 – Typical components of a fault injection environment [Hsueh *et al.* 1997].

2.4.1 Goals of fault injection

In [Arlat *et al.* 1990b] the two complementary main goals of fault injection are identified and characterized: validation and design-aid. The first is related to the fact that fault injection can be viewed as a means to testing the methods and mechanisms used to obtain the confidence in the system, with respect to the inputs they have been designed to cope with - the faults. In this context, two key aspects should be considered: (i) the validation of the verification procedures, used to reveal faults during all the phases of the development process, and (ii) the validation of the fault tolerance mechanisms, aimed to achieving the dependability of the system in the operational phase. Therefore, the fault injection participates in two of the techniques used to attain the

dependability of a system, as refereed in section 2.2.4: fault removal, through the reduction, by verification, of the presence of faults in the design/implementation of the fault tolerance mechanisms; and fault forecasting, through the rating, by evaluation, of the efficiency of the operational behavior of such mechanisms [Arlat *et al.* 1990b, Avresky *et al.* 1996, Christmansson *et al.* 1996a, Voas *et al.* 1997b]. Concerning the design-aid, the fault injection can be applied at the various stages of the development process. Their results are mainly used to measure the quality of the selected solutions and to change them, if necessary.

It must be noticed that, due to the fact that faults are introduced in the target system, which causes the system to run in an altered state, the fault injection is generally unable to determine the accuracy of the results. That is, the fault injection is inadequate to ensure that an application, for example, produces the correct results, according to its specification. Instead, fault injection is very useful to prove that an application produces incorrect results under abnormal operating conditions [Voas *et al.* 1998]. Fault injection is thus appropriate for evaluating the behavior of the systems in the presence of faults and validating their fault tolerance mechanisms [Powell *et al.* 1995, Christmansson *et al.* 1996a, Rela *et al.* 1996, Voas *et al.* 1997b, Cukier *et al.* 1999].

2.4.2 Fault injection in software development cycle

Depending on the phase of the software development cycle in which the system is, different fault injection techniques can be applied, as summarized on Table 2-1: (i) Simulation-based fault injection and (ii) Prototype-based fault injection [Hsueh *et al.* 1997].

The simulation-based fault injection technique is used to evaluate the dependability of a system that is represented by a series of high-level abstractions, allowing early detection of design faults, before the system is

started to be built. The early stage of development, characterized by the absence of any implementation details, imposes a simulation based on simplified assumptions, like the occurrence of errors and failures according a predetermined distribution, such as the exponential distribution. With this technique, the faults are injected by directly modifying the computational state of the simulation [Carreira *et al.* 1999]. Among the most known simulation-based fault injectors, one can mention the FOCUS [Choi *et al.* 1992], the MEFISTO [Jenn *et al.* 1995] and the DEPEND [Goswami *et al.* 1997] tools. Although this method is suitable for the evaluation of the effectiveness of fault tolerant mechanisms and a system dependability in the early phases of its development (conception and design), known as its main advantage, it requires accurate input parameters that are difficult to supply [Hsueh *et al.* 1997]. It should be noticed that parameters from previous experiments could not be adequate due to design and technological changes. This technique is also highly appropriate for the evaluation of dependability of critical systems where the injection of faults in the actual prototype or operational system would be dangerous, as happens in nuclear power systems and avionics. Despite these advantages, accurate results demand very detailed models, whose development can be very expensive. Moreover, manufacturers might not reveal the information needed and the simulation can take a long time to complete.

Phase in Software Development Cycle	Technique
Conceptual and Design	Simulation-based fault injection
Prototype and Operational System	Prototype-based fault injection
Operational System	Measurement-based analysis

Table 2-1 - Experimental techniques for dependability evaluation and their suitability for the different phases of software development cycles.

On the other hand, Prototype-based fault injection allows the evaluation of the system without any assumptions about the system design, and thus, allows more accurate and realistic results, compared to simulation-based analysis. This technique consists on the injection of faults on the target system and on the observation of the corresponding effects. The prototype-based fault injection is useful to:

- Identify system weaknesses, regarding components causing dependability bottlenecks.
- Analyze the system behavior in the presence of faults: (i) determine the coverage of error detection and recovery mechanisms, and (ii) evaluate the effectiveness of the fault tolerance mechanisms and the corresponding performance loss.

In this context, most of the approaches fall into two main categories [Hsueh *et al.* 1997]:

- **Hardware Implemented Fault Injection (HWIFI) –** The faults are injected on hardware level, through logical or electrical faults. This category can further be subdivided into HWIFI with contact, when there is physical contact with the circuit pins of the target system (e.g, methods that use pin level active probes and socket insertion), and HWIFI without contact, in the cases where the injector has no direct contact with the target system (e.g., faults are injected through heavy ion radiation and electromagnetic interferences).
- **Software Implemented Fault Injection (SWIFI) –** The faults are injected at software level (through the corruption of code or data), reproducing errors that would have been produced by faults occurring in hardware of software. SWIFI techniques can also be further categorized into two new classes, depending on the time at which the faults are injected: (i) compile-time injection, corresponding to the case when the faults are injected into the

source code of the target program, and (ii) run-time injection, when the faults are injected during system run-time.

Contrasting with SWIFI, HWIFI techniques require the use of additional and specific hardware to introduce physical faults on the target system, which increase the cost of its use. Moreover, the increasing complexity of hardware makes it harder to inject physical faults as well as to define the corresponding simulation models that effectively represent the systems. Thus, due to its greater flexibility, portability, lower cost and ease of development, the SWIFI tools have become a clear and popular choice in the last decades. However, despite these advantages, the SWIFI tools have some intrinsic drawbacks that should be mentioned:

- Inaccessibility of some locations, when compared to HWIFI tools (e.g. some processor and system resources cannot be reached) [Carreira *et al.* 1998b];
- Difficulty in injecting permanent faults, except for very particular circumstances;
- Disturbance of the execution and, consequently, on the performance of the system under test. This problem, known as intrusiveness, is a consequence of the instrumentation necessary to inject faults and monitor the corresponding effects in the target system. Special care should be taken in order to minimize its effects.
- Poor time resolution due to the possible inability to follow some error propagation, particularly, for errors with very short latency like CPU and bus faults.

Generically, as major drawbacks to the use of the prototype-based fault injection, one can mention the restriction of the study to the set of faults that can actually be emulated and the impossibility to obtain measures like availability and the mean time between failures.

Although all of the experimental techniques have their limitations, they should be used in their appropriate phases and, given their complementarity, their combination can result in a more complete study of the dependability of systems.

As stated in section 2.3, fault injectors are a crucial part of dependability benchmarks. The next section briefly presents the most relevant fault injection tools developed in the last decades. For the purpose of this thesis, only those belonging to the SWIFI family are mentioned.

2.4.3 SWIFI tools

Many fault injection tools have been developed in the last decades. One of the early SWIFI fault injectors is FIAT (Fault Injection-Based Automated Testing Environment) [Segall *et al.* 1988]. This tool adds fault injection and monitoring capabilities to application code and operating system, by changing the code and data that is copied into memory at load time. Faults are triggered when target system execution reaches the locations where special instructions have been inserted in the code. Although this tool could inject memory faults at runtime, it was not able to inject most transient faults. A similar pre-runtime approach of changing the file image generated by the compiler was taken by DOCTOR, a tool developed for the HARTS real-time distributed system [Han *et al.* 1993].

The FINE tool (Fault Injection and moNitoring Environment) [Kao *et al.* 1993] uses a software monitor to trace the control flow and inject faults. Despite its large overhead and the need of the source code of the target application to inject faults, it is significantly more powerful than its predecessors, particularly, in the type of faults that could be injected. DEFINE [Kao *et al.* 1994], an extension of FINE, was developed to include distributed capabilities, introducing a modified hardware clock interrupt

handler to inject CPU and bus faults with time triggers and inject some kind of software faults.

A tool called FERRARI (Fault and ERRor Automatic Real-time Injector), developed for injecting faults using the UNIX *ptrace* function is presented in [Kanawati *et al.* 1995]. The fault injection process initiates and executes the target process in a special trace mode, enabling the injection of transient and permanent faults. It is able to inject a very wide set of fault types, but was restricted to injection in user space.

FTAPE (Fault Tolerant and Performance Evaluator) [Tsai *et al.* 1996] is part of a fault tolerant benchmark, which measures system failures and the system performance degradation during faulty conditions. It also includes a synthetic program for generating CPU, memory and I/O (Input/Output) activity. FTAPE is able to inject fault in CPU registers, memory and disk subsystem. It is capable to select the time and location of faults either based on the workload activity or randomly.

The Xception tool, which uses the debugging and monitoring capabilities of the modern processors, is presented in [Carreira *et al.* 1998b]. It provides a set of spatial, temporal and data manipulation fault triggers like FERRARI or FTAPE, but with a minimal intrusion on the target system, apart from being able to also target system space. Xception was originally implemented on a PowerPC based machine, and has been ported to other processors since then, having originated the unique commercial fault injector available today. Another fault injector, called MAFALDA, presented in [Rodríguez *et al.* 1999], uses principles very similar to Xception, adding mechanisms to intersect and inject system calls in micro-kernels.

The GOOFI (Generic Object-Oriented Fault Injection) tool, presented in [Aidemark *et al.* 2001], is designed to inject faults in various target systems, using different fault injection techniques. The generic architecture of GOOFI assists the user to adapt the tool for new target systems and new

fault injection techniques. The version presented in [Aidemark *et al.* 2001] supports pre-runtime SWIFI, to inject faults into the program and data areas of the target system before it starts to execute, and Scan-Chain Implemented Fault Injection (SCIFI). The SCIFI injects faults via the built-in test logic, such as boundary scan-chains and internal scan-chains, existent in many modern VLSI (Very Large Scale Integration) circuits. An extended and improved version of GOOFI is presented in [Skarin *et al.* 2010]. This new version of the fault injector, named as GOOFI-2, extends the predecessor version with one test port-based technique, which provides the ability to inject errors into some microprocessors, and two SWIFI techniques, which include the ability to use the debugging and monitoring functions available in advanced processors, and to inject faults into registers and memory without any specific hardware.

An improved *ptrace*-based SWIFI tool is presented in [Xu *et al.* 2002]. HiPerFI (High-Performance Fault Injector) reduces very significantly the intrusiveness and overhead caused by the context switch between the injector process and the target application. It also integrates a method, similar to the approach used by Xception, which enables the fault injection mechanism to intersect the kernel exception handlers and thus extends significantly the tools triggering and injection capabilities.

A SWIFI tool also capable of executing hardware-based and simulation-based fault injections is presented in [Stott *et al.* 2000]. NFTAPE (Networked Fault Tolerance and Performance Evaluator) is able to inject multiple fault models (bit-flips in registers and memory, communication errors and I/O faults) with multiple fault triggers, and is especially adequate for distributed systems.

A pioneering fault injector tool, specifically developed for dependability benchmarking, is presented in [Costa *et al.* 2003]. The DBench-FI uses a flexible runtime kernel upgrading algorithm to provide a unique set of characteristics: (i) great simplicity of installation and use, since it can be downloadable from the web and executed *on-the-fly*, without

any special installation procedure; (ii) capable to inject faults/errors in the whole target system, and not just in the user space as some *ptrace*-based tools do, nor requiring a special launching procedure like the one requires by many debugging mechanisms; (iii) does not require the availability of source code of any system component or process; (iv) capable to inject faults even in tasks that are already running when it is installed, irrespectively of their complexity; (v) very low intrusiveness, since it is essentially undetectable; (vi) can be dynamically loaded into the system.

It should be noticed that none of the previous tools satisfied the requirements of web distributable dependability benchmarking, either because the overhead caused would be too high; or only user space could be targeted; or the source code of the target applications was required; or a special debug mode imposing a particularly launch mode was required. Moreover, all of the previous tools have been proposed for the emulation of hardware faults and they are not adequate for the emulation of more complex faults such as software faults [Madeira *et al.* 2000, Jarboui *et al.* 2002].

Despite the version presented in [Costa *et al.* 2003] is only able to inject memory faults, the DBench-FI fault injector is, as shown in [Costa *et al.* 2009] and in the present work, actually compatible with G-SWFIT [Durães *et al.* 2006], the state-of-the-art in software faults model being one of the most versatile fault injector available. A detailed description of DBench-FI is presented in chapter 4, as it constitutes a central tool of the present research work.

2.4.4 Software fault injection

Despite the innumerous works on physical hardware fault injection and emulation, the problem of injecting software faults have barely been addressed. In fact, the potential of the mentioned tools for the emulation of

more complex faults such as software faults is very limited [Madeira *et al.* 2000, Jarboui *et al.* 2002]. This gap can be explained by the limited knowledge about software faults in the real operational environment of systems and, consequently, by the difficulty of defining meaningful and representative sets of software faults. Nevertheless, several studies [Gray 1990, Sullivan *et al.* 1992, Lee *et al.* 1995, Chou 1997, Kalyanakrishnam *et al.* 1999, Li *et al.* 2006] showed that software faults are actually predominant, when compared to other types of system faults, and, considering the huge and growing complexity of today's software, its weight on the overall system dependability will tend to increase. In fact, nowadays it is generally accepted that most of the existing software components have residual defects or bugs, which escape the traditional testing phases of the software development process. Consequently, complex software systems, in which our society increasingly relies, are being executed under potential faulty conditions that have been neither detected nor foreseen [Gray 1985, Chillarege *et al.* 1992, Musa 1996, Weyuker 1998, Knight 2002].

Despite the permanent nature of the software faults [Avizienis *et al.* 2004], practice shows that their behavior is transient. That is, when a failure is observed, it is very difficult to repeat all the precise conditions that trigger it, like particular timing relationships between several system components or other rare and somewhat irreproducible circumstances. Software faults typically manifest only during operations in real field, and usually under heavy or unusual workloads and timing contexts. In fact, studies on field data analysis show that most of software faults are due to overloads, race conditions or timing and exception errors [Sullivan *et al.* 1991, Chillarege *et al.* 1995].

The huge complexity of today's software and the increasing pressure to reduce time to market, together with the recognized and well-known technical difficulties associated to the software development and testing processes [Lyu 1996, Musa 1996], have contributed to the actual scenario. The emulation of software faults and the assessment of the impact of

residual bugs on the validation of software fault tolerance mechanisms is thus of crucial importance as a measure of confidence that can be relied on a given system.

The majority of the studies on software faults have addressed the software development process since that is their decisive origin. Software faults are always a consequence of an incorrect development process, revealing flaws introduced in any of its phases (requirement, specification, design, coding, testing, etc.).

Contributions to the improvement of software development methodologies, namely on software testing, software reliability modeling and risk assessment were presented in [Lyu 1996, Musa 1996].

Mutation testing, sometimes considered the first form of software fault injection, is used for evaluating the adequacy of test data, while minimizing testing times [Budd 1981, DeMillo 1988, King *et al.* 1991]. Originally proposed in [Hamlet 1977], mutation testing consists of a software testing technique based on the automatic[4] creation of different

---

[4] Within the scope of mutation testing, the introduction of changes can also be done through manual insertions, usually by experienced engineers, known as *hand-seeded faults*. However, while hand-introduced faults have argued to be more realistic [Hutchins *et al.* 1994], more recent empirical studies show that automatically generated faulty versions

versions of a program (called *mutants*), each one with a single and simple fault (based on a *mutation operator*), and on the definition of test cases capable to detect the largest number of the injected faults. The mutation testing technique determines the adequacy of the set of test cases by measuring the ratio of faulty versions that have been detected (in which case that mutant is considered "killed"), based on the comparison of the its output with the one produced by the original program, and hence it can be used to estimate and improve the reliability of software [Geist *et al.* 1992, Lyu *et al.* 2003, Dimov *et al.* 2010]. However, in spite of having been widely studied and used over three decades, some problems[5] such as the high computational cost of executing the huge number of mutants against a test set, has preventing mutation testing from being a practical testing technique [Jia *et al. 2011*].

Mutation testing can be considered a case of static or compile-time fault injection, as the source code of the original program is changed before its image is recompiled, loaded and executed, as opposed to the classical

---

(*mutants*) provide a less costly, more practical and accurate method to estimate the fault detection ability of test cases [Andrews *et al.* 2005, Do *et al.* 2006].

[5] Other difficulties related to the oracle cost [Budd *et al. 1982*, Weyuker 1982], i.e. the process of comparing the output of mutated programs with the original one, in each test case, have also been reported.

and dynamic fault injection, characterized by the change in the state of the program/system, during runtime. Regardless the similarities, it is important to highlight the difference of goals between the mutation testing and the fault injection techniques. While the first uses program mutations to identify an adequate test suite during the software development phase, fault injection aims to validate the fault tolerance mechanisms of a system at runtime, and evaluate the behavior of the system in the presence of faults. It is also worth noting that, despite its wide use in software testing, the mutation testing technique is not applicable in the context of COTS, since in this case the source code is typically not available.

Some other studies collect the system operational data from field in order to improve the software development process. In [Gray 1990, Lee *et al.* 1995] are presented the results of the analysis of the software dependability of Tandem systems, based on a census of costumer system outages. The impact of software defects on the availability of a large IBM system is presented in [Sullivan *et al.* 1991]. Also based on field data, [Iyer 1995] presents a study of the effect of the workload on the reliability of an IBM operating system.

In [Voas *et al.* 1997b] the injection of artificial faults, both software and hardware, is proposed for the assessment of software components behavioral quality. Although the fault injection was initially developed in the context of hardware faults, namely with the emulation of transient and permanent faults using the simple bit-flip and stuck-at models, the need for software fault injection has arisen with the emergence of software faults as a major cause of system outages.

With the recognition that the emulation of the most frequent types of programmer mistakes is a good approach for the emulation of software faults, like primarily stated in [Ng *et al.* 1996, Ng *et al.* 1999], some studies on the emulation of software faults by software fault injection and their representativeness have been made. The first studies about the problem of the accurate emulation of software faults by fault injection were presented

in [Christmansson *et al.* 1996a, Christmansson *et al.* 1996b]. Both works propose a general procedure to generate injectable errors and accelerate the failure process, based on the analysis of field data about discovered software faults that have been classified according to the Orthogonal Defect Classification (ODC) - a classification framework for software faults. In the first proposal, [Christmansson *et al.* 1996a] addresses the fault forecast issue, while in [Christmansson *et al.* 1996b] the procedure to generate injectable errors is proposed for fault removal.

An experimental study on the accurate emulation of software faults by fault injection is presented in [Madeira *et al.* 2000]. In a first experiment, a set of real software faults has been compared with faults injected by the Xception SWIFI tool in order to evaluate the accuracy of the injected faults. Results showed the limitations of the usual SWIFI tools in the emulation of different classes of software faults, either because the right error patterns cannot be injected or the tool is too intrusive. This study also discusses the use of field data about real faults and suggests the use of software metrics as an alternative way to guide the injection process when field data is not available. A second experiment evaluates a set of rules for the injection of errors intended to emulate classes of faults.

In [Ng *et al.* 2001] software faults (as well as low-level hardware faults) are injected into an operating system with the aim to improve and validate the robustness of a write-back file cache designed to be as reliable as a write-through file cache. Although the used fault model imitates some specific programming errors in the OS, it is not necessarily applicable to other software systems.

An innovative technique for the injection of software faults is primarily proposed in [Durães *et al.* 2002b] and further developed and extended in [Durães *et al.* 2006]. The G-SWFIT (Generic Software Fault Injection) technique consists of finding key programming structures or patterns at the machine code level in order to emulate high level software faults through the modification of the ready-to-run binary code of the

target software component or module. It uses a set of operators for software fault emulation through low-level code mutations based on an extensive collection of real software faults, as represented in Figure 2-6.

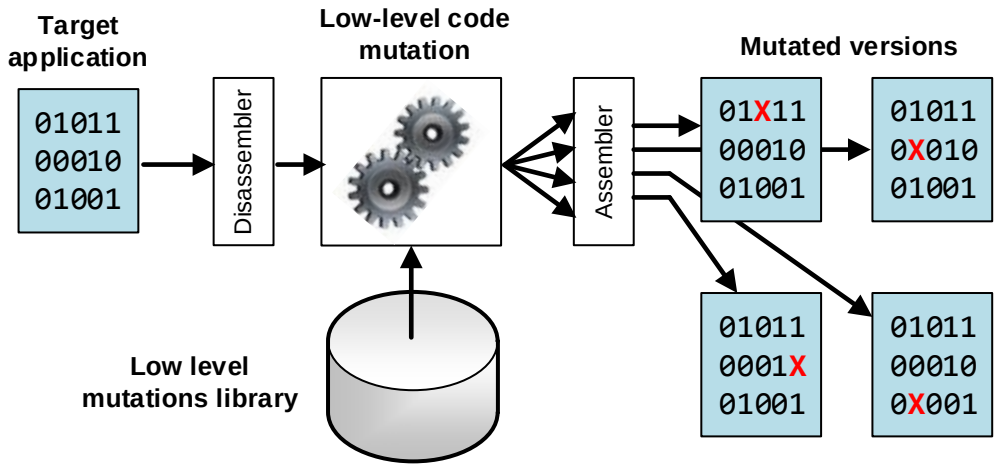


Figure 2-6 – Automated low-level code mutations [Durães *et al.* 2002b].

In fact, the idea that mutations and actual software faults produce identical error patterns and program behavior is supported by the results presented in [Daran *et al.* 1996]. One central advantage of the G-SWFIT method is that software faults can be emulated even when the source code of the target application is not available, as usually happens with COTS. This characteristic is essential for the evaluation of COTS or for the validation of fault tolerance mechanisms in COTS based systems. It should be emphasized that this technique presents an important advantage over the previously mentioned proposal of [Kalakech *et al.* 2004], based on the corruption in the API calls, as the later tries to emulate the effects of real software faults (i.e., errors [Avizienis *et al.* 2004]) instead of emulating the existence of the fault itself. Despite this work was based on the C language, the study also concludes that the considered fault types are independent on specific features of the C language and only minor differences should exist in the fault emulation operators for other languages, such as C++ and Pascal.

Furthermore, some studies show that it is unlikely that software faults could be easily emulated only by API level fault injection [Jarboui *et al.* 2002, Jarboui *et al.* 2003], or even provide empirical evidences that interface faults and software component faults cause substantial different impact in the system [Moraes *et al.* 2006b].

Besides the emulation accuracy, the injection of software faults encompasses two additional challenges:

- The representativeness of the faultload;
- The way of distributing the faults among different components in the target system.

The first issue is related to the fact that the software faults should emulate a set of real software faults that may occur in the system, i.e., they should represent realistic faults that escape the software testing phases of the software development process and still persist in the system. Several recent research works, such as [Durães *et al.* 2006, Moraes *et al.* 2006a , Natella *et al.* 2013], address this subject and present several notable proposals for the definition of representative faultloads based on software faults, as explained later in section 3.4 - Representativeness of Software Faults.

The second challenge concerns the practical difficulty of carrying out a software fault injection campaign using such representative, but huge, faultloads, induced by the vast number of possible fault types and target locations. This problem is even more evident and dramatic in large and complex systems, where the execution time of those campaigns can take several months or even years due to the faultload dimension. This issue is a central topic of the study presented on this thesis. It is fully presented in chapter 4 and discussed in depth in chapters 6 and following.

2.5 Summary

This chapter described the terminology related to the dependability, their attributes, impairments and the mechanisms used to increase the level of confidence that can be relied on a given system.

The state of the art of the area of dependable computing was also presented, through a survey on the relevant work in the areas of dependability benchmarking, fault injection and software faults.

Chapter 3

Dependability Benchmarking of Software Systems

This chapter shows the importance of dependability benchmarking focusing on software systems as well as the challenges that arise in this area. It starts to present a conceptual framework for dependability benchmark, as well as its key dimensions, and highlights the difficulties concerning the experimentation issues of the dependability evaluation of software systems. Finally, the problem of the representativeness of software faults is presented, and the relevant studies that have been carried out with the aim to solve this problematic are discussed in detail.

3.1 Introduction

Despite the substantial improvements in the design and implementation processes of software systems over the last years, it is obvious that the complete elimination of software defects during software development process is very difficult to attain in practice. As a consequence, our society is increasingly dependent on complex software systems that are executed under potential and unforeseen faulty conditions. Due to this difficulty in producing software without defects or

bugs, software developers adopted fault tolerant mechanisms to prevent the consequences of potential failures, which can range from minor inconveniences to real catastrophes [Weinstock *et al.* 1997]. Modern software systems must be fault tolerant (at least to a certain extent), that is, they should be able to provide the expected service even in the presence of faults. In fact, fault tolerance is even recommended by leading safety standards like NASA standard 8719.13B for software safety [Nasa 2004] and the ISO/DIS 26262 standard for automotive safety [ISODIS 2009].

The importance of fault tolerance mechanisms has been reinforced by the current trend of using COTS and COTS-based systems to build larger and more complex systems [Durães *et al.* 2002b, Madeira *et al.* 2003], in application areas that require high dependability. In this context, residual software faults represent a growing risk of unpredictability consequences. According to [Lyu 1995], software fault tolerance techniques are divided into two groups: (i) single version and, (ii) multi-version software techniques. Single version techniques focus on the addition of design mechanisms into a single piece of software, aiming the detection, containment and handling of errors caused by the activation of design faults. Examples are concurrent error detection, checkpointing and recovery, and exception handling [Gray 1985, Cristian 1982]. Multi-version techniques consist on the structured use of multiple versions (or variants) of a piece of software in order to ensure that design faults in one version do not cause system failures. Examples of such techniques include N-version programming (NVP), recovery blocks (RcB), and N self-checking programming (NSCP) [Avizienis 1985, Lyu 1995].

Despite several studies have shown the pertinence and the efficiency of fault tolerance mechanisms on the dependability of systems [Arlat *et al.* 1993], its validation and evaluation are complex and challenging tasks.

Dependability benchmarks allow the answer to that challenge: they should provide generic ways of characterizing the behavior of components

and computer systems in the presence of faults, allowing the quantification of dependability measures.

3.2 General framework

The goal of dependability benchmarks is to provide a cost-effective and reproducible way to evaluate the behavior of components and computer systems in the presence of faults, allowing the quantification of dependability attributes or the characterization of system into well-defined dependability classes. Furthermore, dependability benchmarks should provide a uniform, repeatable and comparable way of performing that evaluation and compare alternative solutions. As these properties represent fundamental goals of a dependability benchmark, they should be taken in consideration right from the earliest phases of the benchmark definition.

A general framework for defining dependability benchmark for computer systems was presented in the context of the DBench Project [DBENCH 2004]. The work carried out presents a conceptual framework and an experimental environment for dependability benchmarking of COTS and COTS-based systems and identifies the following three main classes of impacting dimensions:

- **Categorization –** This dimension describes the considered target system, as well as the dependability benchmark context. It impacts the selection of meaningful benchmark measures, as well as all aspects related to experimentation on the target system.
- **Measure –** This dimension specifies the dependability benchmark measures to be assessed, considering the choices made for the categorization dimension.

- **Experimentation** – This dimension includes all the aspects related to the execution of the experiments on the target system in order to get all the measures selected in the measure dimension.

Figure 3-1 outlines the classification dimensions, as well as their relationships.

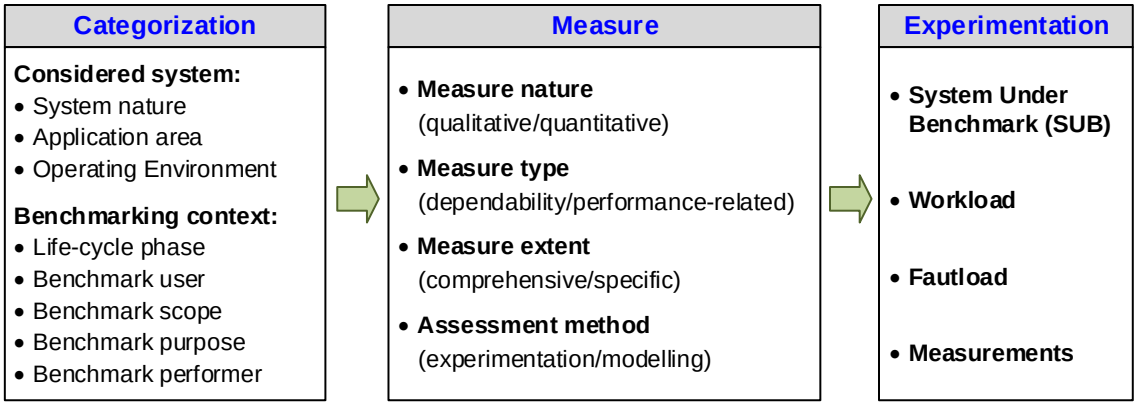


Figure 3-1 – Dependability benchmarking dimensions [DBENCH 2004].

The following subsections detail the mentioned dimensions.

3.2.1 Categorization dimension

This dimension aims to unambiguously identify and specify the Benchmark Target (BT), with respect to its nature, application area and operating environment. It is worth noting that the application area is a key dimension, as it impacts the system execution profile, the operating environment and the benchmark measures. Different application areas require different dependability benchmarks. It should also be noticed that the operating environment may affect both the workload and the faultload, as it encompasses not only functional activity, but also faults, induced by external sources or human-related interaction ones [Voas *et al.* 1997a].

This dimension also describes the benchmark context, which depends from the perspective of its execution and use of results and determines the requirements and the objectives of the benchmark. The benchmarking context is considered a composite dimension, since it includes: (i) the life cycle phase of the BT, in which the dependability benchmark is executed (the benchmark measures greatly depends on the specific phase in which they are obtained); (ii) the benchmark user, concerning the person or entity which is using the benchmark results; (iii) the benchmark scope, related to the possibility of the benchmark results to be used either internally, for system validation and tuning, or externally, for public distribution; (iv) the benchmark purpose, concerning the characterization of the dependability of the target system either in a qualitative or quantitative manner; and (v) the benchmark performer, regarding the person or entity that actually executes the benchmark (manufacturer, integrator, third-party or end-user).

3.2.2 Measure dimension

This dimension encompasses the measures that are relevant for the dependability benchmark, allowing a quantitative or qualitative characterization of the BT. It includes: (i) performance related measures, concerning the evaluation of system performance under faulty conditions; (ii) comprehensive measures, which characterize the system at the service delivery level (expected service), such as transactions per minute, availability or safety; and (iii) specific measures, associated to particular system features, such as the coverage factor or the latency time of fault tolerance mechanisms.

Usual measures include the identification of system failure modes and the system performance evaluation, such as system time response and system throughput (as the injected faults may lead to performance degradation without leading to system failure). It is worth pointing out that, more than the absolute value of the workload execution time, what is

really important in dependability benchmarks is the identification of the impact of the faultload on that execution time. Moreover, dependability benchmarks usually also measure the time needed for the restoration of the expected service, after the occurrence of a faulty situation.

3.2.3 Experimentation dimension

The experimentation dimension includes all aspects related to the experiments executed on BT, according to the categorization and measure dimensions. They include: (i) the System Under Benchmark (SUB), a wider system which includes the Benchmark Target (BT); (ii) the workload, which should represent a typical operational profile for a specific application area; (iii) the faultload, which should also be representative of the real threats that may occur in the system; and (iv) the measurements to be performed, that allows the observation of the behavior of the BT under the applied execution profile, composed by the workload and the faultload.

This dimension should identify and specify the System Under Benchmark (SUB), which consists in a setup (hardware and software resources) that hosts and runs the BT, and performs the experiments defined by the benchmark. The SUB is also used to apply both the workload and the faultload, and to collect the measurements relevant to the dependability benchmark.

It is worth mentioning that the definition of a faultload is a practical process, based on observations, knowledge and reasoning. Information about failure data reported in the field [Kanoun *et al.* 1997], knowledge about the most frequent residual software defects found in deployed software systems [Durães *et al.* 2003b], characteristics of the operating environment, like the most frequent common administrator mistakes [Vieira *et al.* 2003], or even information from experimental and simulation studies, are examples of inputs used for the proper definition of faultloads.

3.2.4 Benchmark scenarios

All the steps, and their interactions, needed to achieve a dependability benchmark form a benchmark scenario. According to [Kanoun *et al.* 2002, Madeira *et al.* 2002], there are three different key steps for system dependability benchmarking: *analysis*, *experimentation*, and *modeling*. Figure 3-2 shows a high-level scheme that depicts these stages and their relations.

A benchmark starts by an *analysis* step, in which specific choices are made concerning the categorization and measure dimensions of the target system. Depending on the measure assessment method, the output of this step can consist in two different types: (i) the workload, faultload and measurements, for experimental measures, (output represented by link A), and (ii) a deeply analysis of the system behavior (output represented by link B) in order to prepare a system modeling, in case it is required.

According to the choices made in the *analysis* step, the selection of the elements concerning the experimentation dimension is then achieved in the *experimentation* step (link A), which allows the characterization and assessment of the target system dependability. This step includes the execution of the workload and faultload, and the collecting of the measurements under the applied execution profile. As a consequence of the strong relationship between the experimentation process and the target system, all the components already defined at a high level during the previous steps (workload, faultload and measurements), should be refined in order to incorporate all the target system specificities. The correct implementation of these components at system level should be carefully addressed according to the procedures and rules defined in the benchmark, which usually include configuration disclosures and rules related to the scalability and to the benchmark measurements.

A *modeling* step is also required when comprehensive measures of the target system are likewise deemed of interest (link B). It is used to build a

representation of the system, in order to model the system behavior considering failure occurrences, errors detection and propagation, system recovery, and other similar events or activities. It is worth recalling that these analytical models require the allocation of numerical values to the model parameters, which is usually done through experimental measurement, field data or past experience related to similar systems.

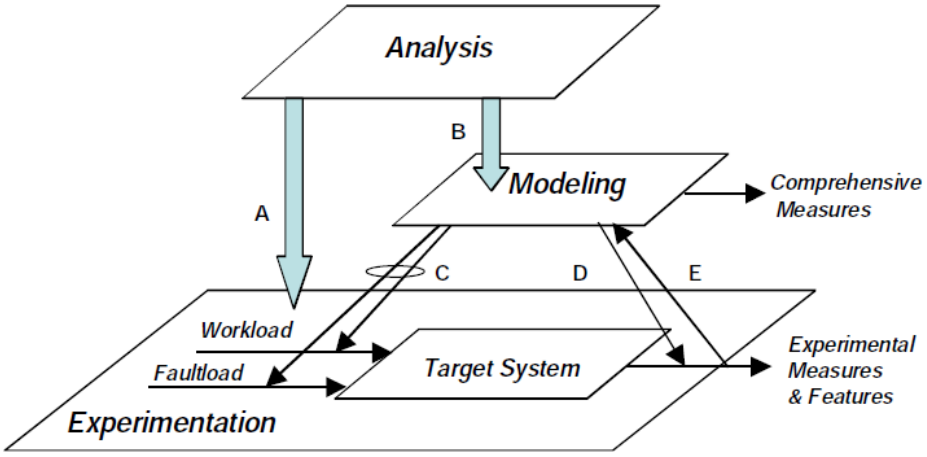


Figure 3-2 – Dependability benchmarking scenarios [Kanoun *et al.* 2002].

The modeling and the experimentation steps are usually used in a complementary way, as depicted in Figure 3-2. Modeling can be used to improve both the workload and the faultload, by assisting in the selection of their most significant classes (link C), and also to guide the selection of most relevant experimental measures and features that need to be assessed by the benchmark experimentation (link D). This is the case of dependability benchmarks in which the experimentation is supported by modeling (scenario 1: represented by all the three steps and the links A, B, C and D). On the other hand, in some benchmarks the experimentation may also help in the improvement and validation (or even in the correction) of the analytical model produced in the modeling step (link E). This occurs in benchmarks in which the modeling is supported by experimentation (scenario 2: represented by all the three steps and the links

A, B, and E), such as when some experimental measures are used by the analytical models.

There are also dependability benchmarks in which modeling and experimentation are supported by each other (a combination of the previous scenarios 1 and 2), and where outputs are simultaneously constituted by experimental measures and features, as well as of comprehensive measures based on modeling (scenario 3: represented by the full steps and links of Figure 3-2).

In addition to these three types of benchmark scenarios, there are also dependability benchmarks based only in experimentation (scenario 4: represented by the analysis and experimentation steps and by the link A). This is the case of the well-known performance benchmarks extended with dependability measures, as the ones used in this thesis.

3.3 Performing the experiments

The benchmark experiments aim to execute the workload and evaluate the behavior of the BT in the presence of faults, as a result of measurements. In practice, the SUB is often a wider system that includes the BT, such as when the BT is a software component like an operating system or a database management system (DBMS). It is also very important to note that the SUB should be carefully and explicitly documented, as the benchmark must be properly interpreted and reproducible.

Furthermore, as already mentioned, in the case of benchmarking of software systems using software fault injection, it is fundamental the existence of a clear separation between the BT and the software components that are selected as Fault Injection Target (FIT). The BT should not be modified by the faultload in order to guarantee the inviolability of the BT and the credibility of the dependability benchmark, especially from the point of view of the BT provider. Instead, the software faults should be

injected in one component (the FIT) in order to evaluate their impact in the other components (the BT) or in the overall system.

Figure 3-3 depicts the relation between the SUB, the BT and the FIT, in the case when the FIT is an operating system and the BT is an application program, such as, for example, a web-server.

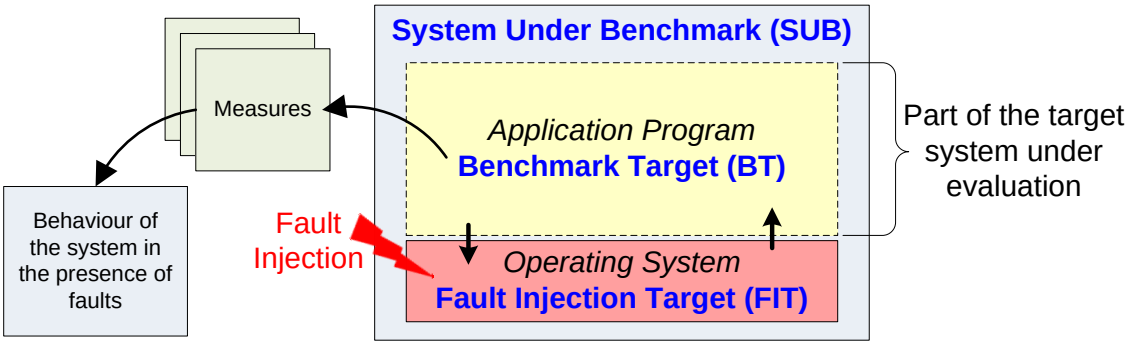


Figure 3-3 – Relation between System Under Benchmark (SUB), Benchmark target (BT) and Fault Injection Target (FIT).

To perform the dependability benchmark, concerning the benchmark experimentation dimension, another element is needed in order to manage and automate the experiments. This key component, known as the Benchmark Management System (BMS), is responsible for the control of all the aspects of the benchmark experiments, namely: the workload submission, the injection of faults, the coordination and synchronization of the several components involved in the experiments and collecting the information needed to process measurements. The BMS usually includes several resources and instrumentation modules in order to fulfill its functions. Moreover, the specific tasks assigned to the BMS should be clearly defined in the benchmark specification, since they are very dependent on the benchmark characteristics.

Beyond a description of the setup required to run the benchmark, in order to control the way a dependability benchmark is applied and used, and to ensure uniform conditions for measurements, dependability

benchmarks should also describe a set of procedures and rules. These procedures and rules are, naturally, dependent on the specificities of the benchmark itself and usually include system configuration disclosures, rules related to the scalability of the benchmark and rules related to the benchmark measurements. This latter kind of rules encompasses: (i) a precise specification of the benchmark measures; (ii) information about the domain in which those measures are valid and meaningful; and (iii) a detailed specification of all the procedures and steps required to obtain those measures (usually programs source code, language specification texts, etc.).

3.4 Representativeness of Software Faults

The acceptability of dependability benchmarks is mainly supported on two fundamental and complementary characteristics: reproducibility and generalization. The former requires the existence of well-defined procedures that allow repeating the benchmark in the same environment, possibly by a different team, and obtaining statistically equivalent results. The latter consists of the ability to generalize the experimental results through some kind of inductive and logical reasoning, making the results useful and meaningful in broader context than the one used in the experimental setup.

Reproducibility is sometimes referred as normalization and encompasses the ability to reproduce the observations and the measurements, either in a deterministic or in a statistical way, providing confidence in the experimental results.

Unfortunately, the reproducibility and the generalization are, in practice, very difficult to attain. The lack of portability of the tools used in the experiments, together with the difficulty to reproduce the experimental conditions, limits the reproduction of the results to a merely statistical

basis. On the other hand, the absence of the necessary representativeness of the experiments can also prevent the desired level of generalization.

Representativeness concerns the ability of a dependability benchmark, its measures and experimental conditions, to represent real world scenarios in a realistic way. It determines the validity and the usefulness of the benchmark results. Representativeness concerns not only the statistical perspective of the results, but also the representativeness of almost all elements of the benchmark. For example, it is of crucial importance regarding the techniques used for fault injection, since it is fundamental to guarantee that the injected faults do represent the real faults experienced in the field. However, that is not an easy task. Several studies on fault representativeness, accuracy and equivalence of fault injection techniques [Daran *et al.* 1996, Folkesson *et al.* 1998, Madeira *et al.* 2000] showed that not all injection techniques can accurately emulate all types of faults.

The representativeness issue also assumes a special importance for the workload and faultload components of the benchmark. Concerning the workload, it is essential that execution profile simulates the activities found in real systems. Regarding the faultload, it must be ensured that the injected faults do represent real faults that may affect the systems in the field. However, unlike the definition of adequate workloads, which is an already resolved issue, with large use in performance benchmarks, the definition of representative faultloads is still an open issue. In fact, it is one of the most critical and difficult tasks in a dependability benchmark definition.

Random fault distributions based on the size of the physical devices have been commonly accepted and used for the injection of hardware transient faults. However more sophisticated distributions are necessary for the injection of software faults. In fact, regarding software faults, the representativeness of the faultload is a special and central property, as the injected faults should represent realistic faults experienced in the field

[Vieira *et al.* 2003, Durães *et al.* 2004a], i.e., software faults that escape the usual software testing phases of software development process and still persist in the system. Only a faultload that is representative of these residual software faults can assure an accurate evaluation of dependability attributes, seen as measures, and an efficient validation of the fault tolerant mechanisms. Unfortunately, the representativeness of software faultloads is very difficult to attain. Information about real software faults found in field is fundamental to understand software faults and help in the characterization of significant fault attributes, such as fault locations and types, as well as their respective frequency of occurrence. However, field data and research works concerning software faults are rare and only in recent years they have been the focus of attention of researchers [Gray 1990, Lee *et al.* 1995, Chillarege *et al.* 1995, Christmansson *et al.* 1996a, Madeira *et al.* 2000, Durães *et al.* 2006, Moraes *et al.* 2006a, Basso *et al.* 2009, Sanches *et al.* 2011, Natella *et al.* 2013].

The gathering and study of software faults have been widely used for the analysis and improvement of the software development and maintenance processes – the main goal of leading software quality standards and frameworks, such as the Capability Maturity Model Integrated (CMMI) [Chrissis *et al.* 2003].

A uniform approach for the classification of software anomalies is provided in the IEEE Standard Classification for Software Anomalies [IEEE 1994], which was further revised in 2010 [IEEE 2010]. This standard, sponsored by the Software & Systems Engineering Standards Committee of the IEEE Computer Society, states that software anomalies, seen as problems or defects, may be found during any stage of the software development life cycle (review, test, analysis, compilation, use of software products, use of documentation, etc.) In its initial version [IEEE 1994], the standard presents a comprehensive categorization of the potential defects into a set of defect types: logic problem, computation problem, interface/timing problem, data handling problem, data problem, documentation problem, document

quality problem, and enhancement. The finite nature and the specificity of the categories considered in this classification forced a redefinition of the standard. The latest version of the standard [IEEE 2010] replaced the list of defect types by a set of defect and failure attributes (Table 3-1 and Table 3-2, respectively), aimed to help the identification and tracking of software anomalies and to improve the software development process.

A significant contribution on collecting and analyzing observed software faults is presented in [Chillarege *et al.* 1992, Chillarege 1996]. This work presents the Orthogonal Defect Classification (ODC), a classification framework for the classification of software faults (i.e., defects) into mutually exclusive classes, in which signatures are extracted from defects that occur through development and field use, in order to improve the software product and the software development process. The usefulness of the ODC methodology in providing this feedback was confirmed by several pilot projects [Chillarege *et al.* 1992].

Though the intended primary goal of ODC is to provide a feedback on to the software development process at IBM, it ends up to be a useful defect classification regarding the problem of software fault emulation by fault injection. ODC is based on the previous observation that there is a case-effect relationship between the semantics of the software defects and the activities of the software development process [Chillarege *et al.* 1991]. According to ODC, a software fault is classified based on the modification that is necessary to undertake in the code in order to correct the defect. It is worth noting that this classification considers that mistakes may occur in every stage of the software development process (specification, design, coding, testing, documentation, etc.). Table 3-3 shows the ODC defect types directly related to code, and, therefore, relevant to the present work. Besides this fault classification has been built and used for the improvement of the software designing process at IBM, it also constitutes a central basis to understand and classify software faults from the injection point of view.

Attribute	Definition
Defect ID	Unique identifier for the defect.
Description	Description of what is missing, wrong, or unnecessary.
Status	Current state within defect report life cycle.
Asset	The software asset (product, component, module, etc.) containing the defect.
Artifact	The specific software work product containing the defect.
Version detected	Identification of the software version in which the defect was detected.
Version corrected	Identification of the software version in which the defect was corrected.
Priority	Ranking for processing assigned by the organization responsible for the evaluation, resolution, and closure of the defect relative to other reported defects.
Severity	The highest failure impact that the defect could (or did) cause, as determined by (from the perspective of) the organization responsible for software engineering.
Probability	Probability of recurring failure caused by this defect.
Effect	The class of requirement that is impacted by a failure caused by a defect.
Type	A categorization based on the class of code within which the defect is found or the work product within which the defect is found.
Mode	A categorization based on whether the defect is due to incorrect implementation or representation, the addition of something that is not needed, or an omission.
Insertion activity	The activity during which the defect was injected/inserted (i.e., during which the artifact containing the defect originated).
Detection activity	The activity during which the defect was detected (i.e., inspection or testing).
Failure reference(s)	Identifier of the failure(s) caused by the defect.
Change reference	Identifier of the corrective change request initiated to correct the defect.
Disposition	Final disposition of defect report upon closure.

Table 3-1 –Defect attributes [IEEE 2010].

Attribute	Definition
Failure ID	Unique identifier for the failure.
Status	Current state within failure report life cycle. See Table B.1.
Title	Brief description of the failure for summary reporting purposes.
Description	Full description of the anomalous behavior and the conditions under which it occurred, including the sequence of events and/or user actions that preceded the failure.
Environment	Identification of the operating environment in which the failure was observed.
Configuration	Configuration details including relevant product and version identifiers.
Severity	As determined by (from the perspective of) the organization responsible for software engineering. See Table B.1.
Analysis	Final results of causal analysis on conclusion of failure investigation.
Disposition	Final disposition of the failure report. See Table B.1.
Observed by	Person who observed the failure (and from whom additional detail can be obtained).
Opened by	Person who opened (submitted) the failure report.
Assigned to	Person or organization assigned to investigate the cause of the failure.
Closed by	Person who closed the failure report.
Date observed	Date/time the failure was observed.
Date opened	Date/time the failure report is opened (submitted).
Date closed	Date/time the failure report is closed and the final disposition is assigned.
Test reference	Identification of the specific test being conducted (if any) when the failure occurred.
Incident reference	Identification of the associated incident if the failure report was precipitated by a service desk or help desk call/contact.
Defect reference	Identification of the defect asserted to be the cause of the failure.
Failure reference	Identification of a related failure report.

Table 3-2 – Failure attributes [IEEE 2010].

The characteristics of the ODC classification, namely the fact that the considered classes are unambiguously close to the code and to the programmer, showed to be fundamental to a new perspective on the problem of the accurate emulation of software faults by fault injection. This problematic, fundamental in dependability benchmarks of software systems, was first addressed in [Christmansson *et al.* 1996a]. The study proposes a framework for the generation of errors that emulate real software faults, based on field data of the system under analysis, about discovered software faults that have been classified using ODC. Despite the innovative character of this work, its interest is, in practice, strongly restricted by the existence of field data on real software faults found in the target system, which makes the technique very difficult, or even impossible, to apply in practice.

Defect type	Description
Assignment	Value(s) assigned incorrectly or not assigned at all
Checking	Missing or incorrect validation of data or incorrect loop or conditional statements
Interface	Errors in the interaction among components, modules, device drivers, call statements, or parameters lists
Timing/Serialization	Missing or incorrect serialization of shared resources
Algorithm	Missing or Incorrect implementation that can be fixed by (re)implementing an algorithm or data structure without the need for requesting a design change
Function	Affects a sizeable amount of code and refers to the capability that is either implemented incorrectly or not implemented at all

Table 3-3 – ODC defect types.

A subsequent study [Madeira *et al.* 2000] also showed that typical SWIFI tools were not adequate for the emulation of software faults through the use of error patterns like the ones proposed in [Christmansson *et al.* 1996a], as only some types of those error patterns could be injected. One of the reasons relies on the fact that, in its genesis, the ODC classification does not take in account the fault emulation point of view, regardless the fact that the considered classes are unambiguously close to the code and to the programmer, once they are based on the correction of the software defects.

With the aim bridging this gap, an ODC classification extension, built under the fault emulation perspective, is presented in [Durães *et al.* 2003b, Durães *et al.* 2006]. This proposal resulted from an exhaustive field study of real software bugs found in well-known open source software written in the C language (including user applications and system code) and is based on the observation that a software defect consists of one or more missing, wrong or superfluous programming language constructs (such as program statements, functions, expressions, etc.). Accordingly, this study classifies each one of the ODC defect types into three new additional types, according to the corresponding erroneous program construct: Missing construct, Wrong construct or Extraneous construct. Table 3-4 shows the extended ODC classification, with concrete examples of each class of defect types, as well as the corresponding percentage of faults found in the field. It should be noticed that, as the analyzed field data does not include any information about the timing or serialization properties, the Timing/Serialization defect type was not considered.

It is worth pointing out that both of the distributions, the one presented in [Durães *et al.* 2003b, Durães *et al.* 2006] and that presented in [Christmansson *et al.* 1996a], follow the same trend in the fault distribution across the ODC fault types (see Table 3-5).

Defect type	Nature	Examples of code mistake	% of Faults
Assignment	**Missing**	A variable was not assigned a value, a variable was not initialized, etc.	9.3%
	Wrong	A wrong value (or expression result, etc.) was assigned to a variable	10.5%
	Extraneous	A variable should not have been subject of an assignment (value, expression result etc.)	1.6%
Checking	**Missing**	An "if" construct is missing, part of a logical condition is missing, etc.	16.9%
	Wrong	Wrong "if" condition, wrong iteration condition, etc.	7.9%
	Extraneous	An "if" condition is superfluous and should not be present	0.1%
Interface	**Missing**	A parameter in a function call was missing; incomplete expression was used as parameter	1.6%
	Wrong	Wrong information was passed to a function call (value, expression result etc.)	5,7%
	Extraneous	Surplus data is passed to a function (e.g. one parameter too many in function call)	0.0%
Algorithm	**Missing**	Some part of the algorithm is missing (e.g. function call, an iteration construct, etc.)	33.2%
	Wrong	Algorithm is wrongly coded or ill-formed	6.0%
	Extraneous	The algorithm has surplus steps or a unnecessary function is called	0.9%
Function	**Missing**	New program modules were required	3.1%
	Wrong	The code structure has to be redefined to correct functionality	3.0%
	Extraneous	Portions of code were completely superfluous	0.0%

Table 3-4 – Fault nature totals across ODC types [Durães *et al.* 2006].

In fact, it can be observed from Table 3-5 that, for both distributions: Algorithm defects are the dominant fault type; Assignment and Checking defects have similar frequency; and the Interface and Function defects are clearly the less frequent type of faults found in field, according to both works. Moreover, both works show similar values for all ODC types.

ODC Defect type	ODC Defect type distribution	
	[Durães *et al.* 2006]	[Christmansson *et al.* 1996a]
Assignment	21.98%	21.4%
Checking	17.48%	24.9%
Interface	8.17%	1.6%
Algorithm	43.41%	40.1%
Function	8.74%	6.1%

Table 3-5 – Comparison of Fault distribution across ODC defect types.

The independency of both research works and the fact that they analyzed quite different program types, suggest that this fault distribution is reasonably independent from the nature of the program and, thus, it seems to confirm the representativeness of the respective software defects distribution for programs in general.

The work presented in [Durães *et al.* 2006] used the new classification scheme to classify 668 faults from the field, through the analysis of 12 widely deployed software systems. Results show that most of the software faults found belong to a small set of fault types, and that the remaining fault types encompass a small number of faults. Table 3-6 presents the most common set of fault types found. It is worth noting that this set of fault types represent a total of approximately 68% of all faults collected in field. The study shows that these types of software faults can be considered representative of the most common types of software faults and,

consequently, they should be considered in software faults emulation experiments. The paper [Durães *et al.* 2006] argues that, although other fault types may occur in the field with the analysis of more field data on real software faults, they are probably very rare, since they were not found among the analyzed faults. Moreover, they would not change the analysis of the most frequent types.

The research work carried out in [Durães *et al.* 2006] presents important results towards the characterization of software faults. The proposed methodology allows a greater adaptability to software fault injection, as it contains clear indications of how to manipulate the target program code in order to inject a fault. In fact, it also proposes a library of fault emulation operators for software fault injection, as explained in section 2.4.4. These operators guide the mutation of the ready-to-run binary code of software modules in order to mimic real software faults, reproducing the code that would be generated by the compiler if the intended software faults were in the high-level source code. The technique, named G-SWFIT, consists in the scanning of the target code application for specific low-level instruction patterns (sequence of machine code instructions) and in applying the mutation to emulate the intended software fault. It is worth pointing out that, unlike [Christmansson *et al.* 1996a], this work presents a technique for the emulation of real software faults, even when field data is not available for the target system, as it usually happens for third-party software components.

Moreover, despite the full work was based on the C language, other languages like C++ and Pascal were also analyzed in this study. Results show that the considered fault types are not dependent on specific features of the C language and only minor differences should exist in the fault emulation operators. It should also be noticed that, as the G-SWFIT operators reproduce faults that escape the traditional testing phases of software development process, they only encompasses 12 software fault types of the total of 71 mutation operators proposed in [Delamaro *et al.*

1996] for the assessment of the exhaustiveness of test cases (regarding the C language).

Fault Types		# Faults	ODC Type				
			Ass.	Chk.	Int.	Alg.	Fun.
Missing	*if* construct plus statements	71				✓	
	AND sub-expr in expression used as branch condition	47		✓			
	function call	46				✓	
	if construct around statements	34		✓			
	OR sub-expr in expression used as branch condition	32		✓			
	small and localized part of the algorithm	23				✓	
	variable assignment using an expression	21	✓				
	functionality	21					✓
	variable assignment using a value	20	✓				
	if construct plus statements plus else before statements	18				✓	
	variable initialization	15	✓				
Wrong	logical expression used as branch condition	22		✓			
	algorithm - large modifications	20					✓
	value assigned to variable	16	✓				
	arithmetic expression in parameter of function call	14			✓		
	data types or conversion used	12	✓				
	variable used in parameter of function call	11			✓		
Extraneous	variable assignment using another variable	9	✓				
Total Faults for these types in each ODC type		**452**	93	135	25	192	41
Coverage relative to each ODC type (%)		**68**	65	81	51	72	100

Table 3-6 – Most common faults found in field for several software systems [Durães *et al.* 2006].

In order to extend this fault model to different high level languages, with different programing paradigms, some subsequent studies were presented. The works presented in [Basso *et al.* 2009, Sanches *et al.* 2011] use the Java language to show that, when and object-oriented languages are considered, the set of the most common software fault types presented in Table 3-6 can be extended with new object-oriented fault types, according to the Java language specific characteristics and the object-oriented paradigm.

An approach for improving software fault representativeness and, at the same time, reducing the size of the faultload produced by the G-SWFIT technique is presented in [Natella *et al.* 2013]. This study analyzed the representativeness of a large set of injected faults, representing the most frequent software fault types (as summarized on Table 3-7) found in field (according to [Durães *et al.* 2006]), with respect to its ability to escape actual test suites adopted by software developers for detecting faults before software release: the study argues that faults that are easily identified by test suites should not be considered as representative.

Defect type	Examples of code mistake
MFC	Missing Function Call
MVIV	Missing Variable Initialization using a Value
MVAV	Missing Variable Assignment using a Value
MVAE	Missing Variable Assignment using a an Expression
MIA	Missing IF construct Around statements
MIFS	Missing IF construct plus Statements
MIEB	Missing IF construct plus statements plus Else Before statements
MLC	Missing AND/OR clause in branch condition
MLPA	Missing small and Localized part of the algorithm
WVAV	Wrong Value Assigned to Variable
WPFV	Wrong variable used in Parameter of Function call
WAEP	Wrong Arithmetic Expression in Parameter of function call

Table 3-7 – Most frequent software fault types analyzed in [Natella *et al.* 2013].

This work uses a fault injection tool called SAFE [SAFE] to inject the most common software fault types found in field in three real world software systems widely used in business and safety-critical contexts, for which real test suites are available: the MySQL [MySQL] and PostgreSQL [PostgreSQL] DBMS engines, and the kernel of the RTEMS Real-Time Operating System [RTEMS, Rufino *et al.* 2007]. It is worth noting that the software faults are injected in the source code, instead of the binary code, through the production of a set of different faulty source code files, each containing a specific software fault, as summarized in Figure 3-4. The fault injection tool starts to statically analyze the target program and builds an abstract representation of the source code, called an Abstract Syntax Tree, responsible for guiding the identification of locations where a specific software fault type can be introduced, according to the software fault operators defined in [Durães et al. 2006]. Thereafter, the tool creates a set of patch files, each one containing a different faulty, but syntactically correct, version of the code, which is then compiled.

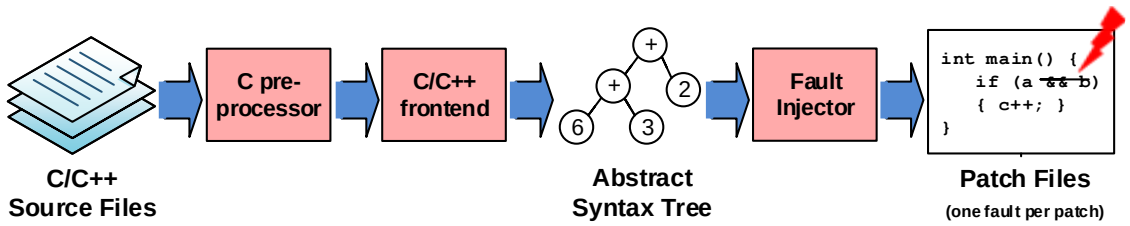


Figure 3-4 – Process for generating faulty versions of the target system [Natella *et al.* 2013].

The experimental setup used in this study is presented in Figure 3-5. In each experiment the system under test is replaced with a faulty version, in which the Test Manager executes a test case and collects the test result.

In order to analyze which faults can be considered representative, i.e. software faults that escape to test suites, each one of the generated faulty versions of the code was run against 50 test cases, randomly chosen for each software system.

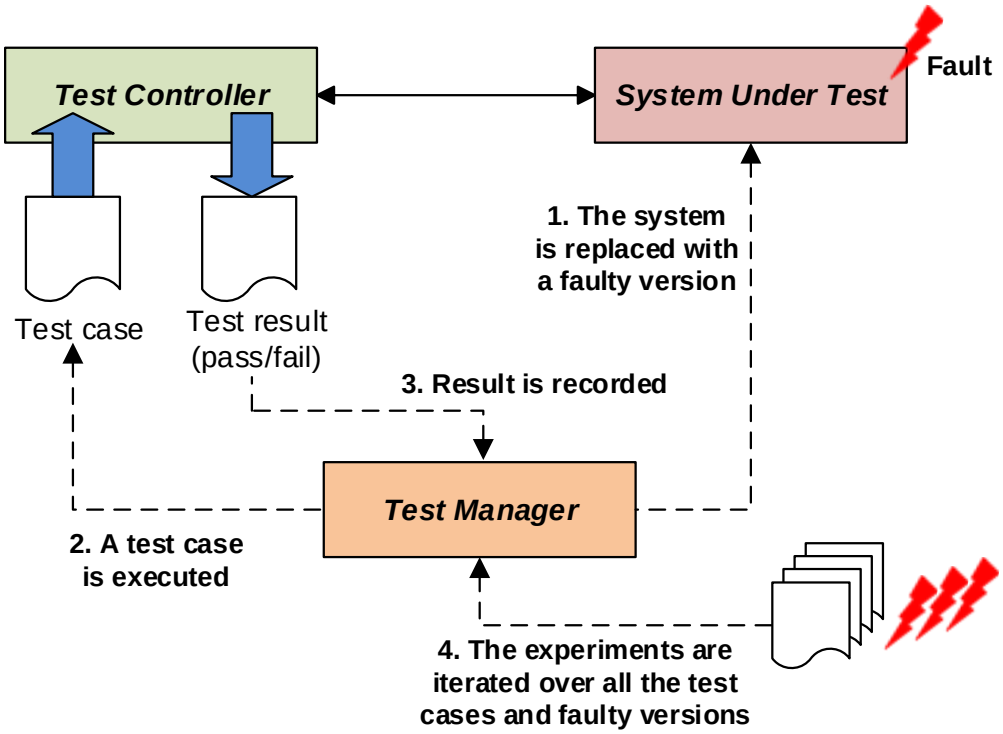


Figure 3-5 – Experimental setup used in [Natella *et al.* 2013].

The conducted experiments show that a significant part of the injected faults is detected by most of the test cases and, consequently, the study argues that they should not be considered as representative: 14.57% and 23.13% for the MySQL and PostgreSQL DBMSs, respectively, and 72.23% for the RTEMS.

This work also states that there is a relationship between fault representativeness and fault locations, and shows that fault representativeness can be improved with the use of classification algorithms and software metrics for the selection of a subset of components suitable for the injection of representative software faults. Both a supervised (decision trees) and an unsupervised algorithm (Lloyd k-means clustering) were evaluated for the improvement of the faultload representativeness and a set of software metrics commonly used by

researchers and practitioners was used for analyzing software complexity: Lines of Code, McCabe's Cyclomatic complexity and FanIn/FanOut[6].

This study concludes that the faultload can be improved, by including a greater number of representative faults, using either the supervised or the unsupervised algorithms (with an increase of 4.10% to 26.08% and of 2.16% to 16.24%, respectively). At the same time, the proposed approach can reduce the faultload size of 30.30% to 69.43% for the supervised algorithm, and of 22.16% to 59.13% for the unsupervised one.

It should be noticed that the supervised classifier requires a training set in order to classify unknown elements, which, in practice, reveals to be a strong limitation, since it involves an extensive and time consuming experimental analysis. In order to overcome this need (the main limitation of the supervised algorithm), this study also presents an unsupervised classifier, relying on the observation that suitable components have lowest FanIn and FanOut values, as those components are less exposed to testing.

---

[6] FanIn/FanOut are software complexity metrics based on system structure and information flow, derived from the concept presented in [Henry *et al.* 1981]. FanIn represents the count of unique components (functions or files) that call (or are called by, in the case of FanOut) a given component, either directly, or indirectly (via other components).

3.5 Summary

This chapter presented a conceptual framework for dependability benchmarking. It also discussed the challenges and difficulties faced with the dependability benchmarking of software systems, namely, concerning the experimentation issues and the representativeness of software faults. Relevant works in the area are also described in detail.

Chapter 4

Software Fault Injector

Previous chapters introduced the basic concepts of dependability and presented the fault injection as a method for its evaluation. Special emphasis was given to software fault injection and to dependability benchmarks, given its remarkable importance to industry and end users.

This chapter is dedicated to the presentation of an innovative fault injector, called DBench-FI, specially designed for dependability benchmarks and whose unique characteristics make it the most flexible fault injector available. DBench-FI constitutes a central tool in the present study.

4.1 Introduction

As mentioned in Chapter 3, reproducibility and the generalization are two main properties of dependability benchmarks, which support its indispensable acceptability, among other demanding requirements. However, in practice, those two properties are very difficult to attain, mainly due to the inexistence of especially adequate tools which support the experiments. One of those crucial tools is the fault injector. Dependability benchmarks must include fault injectors with very specific

features: (i) they should be very easy to install and use, without the need for any complex setup or installation procedure; (ii) have high level of portability; (iii) have very low intrusiveness; (iv) be capable of injecting faults in both user and system spaces; (v) and in code and data segments of any process, irrespective of their complexity; (vi) be independent of the availability of any source code of any system component or user process; (vii) be dynamically linked into a target system; and (viii) be compatible with the latest and most advanced software fault models.

Despite all the developments, none of the existing fault injection tools (presented in section 2.4.3) satisfied these requirements, mostly because of one or more of the following reasons:

- The overhead caused by the fault injector is too high;
- Only user space could be targeted;
- The fault injector requires the availability of any system component or user process (usually, the source code of the target application);
- A special debug mode imposing a particularly launch mode is required;
- Complex installation procedures are required.

In order to fulfill the mentioned requirements, a new version of the DBench-FI fault injector, primarily presented in [Costa *et al.* 2003], was developed. In addition to all the other characteristics that makes this SWIFI tool special adequate for dependability benchmarking, DBench-FI is now fully compatible with the Generic Software Fault Injection Technique (G-SWFIT), the state-of-the-art in software faults model [Durães *et al.* 2006]. Despite this new capability, this new version of DBench-FI still maintains its initial characteristics. Namely, it still does not require any special installation procedure, contrasting with the majority of the existing SWIFI tools, like for instance Xception [Carreira *et al.* 1998b] (which requires some

changes to kernel that have to be done offline). With DBench-FI everything is done on-the-fly.

It is worth noting that the main idea that supports the initial development of this fault injector was the creation of a conceptual model and an experimental environment for dependability benchmarks (the main goal of Project DBench [DBENCH], project in which it was developed), and the observation of the inexistence of a fault injector compatible with its integration. Therefore, the first version of the DBench-FI fault injector, presented in [Costa *et al.* 2003], uses a very simple error model - it just changes the value of memory locations (data segment) of user applications. This simple error model was deemed sufficient to demonstrate its ability to inject faults, and appropriate for the first versions of the benchmarks, particularly if the target areas for injection are carefully chosen, as was done in the experiments reported in the mentioned study.

The new version of DBench-FI, supporting more complex fault models like G-SWFIT, was already tested and used in the research work presented in [Costa *et al.* 2009]. The current version targets the Linux OS on 32 bit Intel processors, and uses a flexible runtime kernel upgrading algorithm to allow access to the target process memory space, that can be in either user or system space, enabling in this way the injection of faults. Presently DBench-FI is, to the best of our knowledge, one of the most versatile fault injectors available.

The next section presents the architecture of the current version of DBench-FI, showing the modules that constitute it and the way they interact with each other and with the user. The implementation details are also presented, as well as some characteristics of operating systems in which relies the fault injector tool.

The methodology used, which forms the basis of the fault injector, constitutes the main innovation comparing to the existing SWIFI tools, being responsible for the unique characteristics presented by the fault

injector. The DBench-FI enables a breakthrough in the areas of fault injection and dependability benchmarking, opening new perspectives hardly achievable with existing methods.

4.2 Fault Injector Architecture

The current version of DBench-FI consists of two modules, as shown in Figure 4-1: *a fault injector core module* and a *fault injector controller module*.

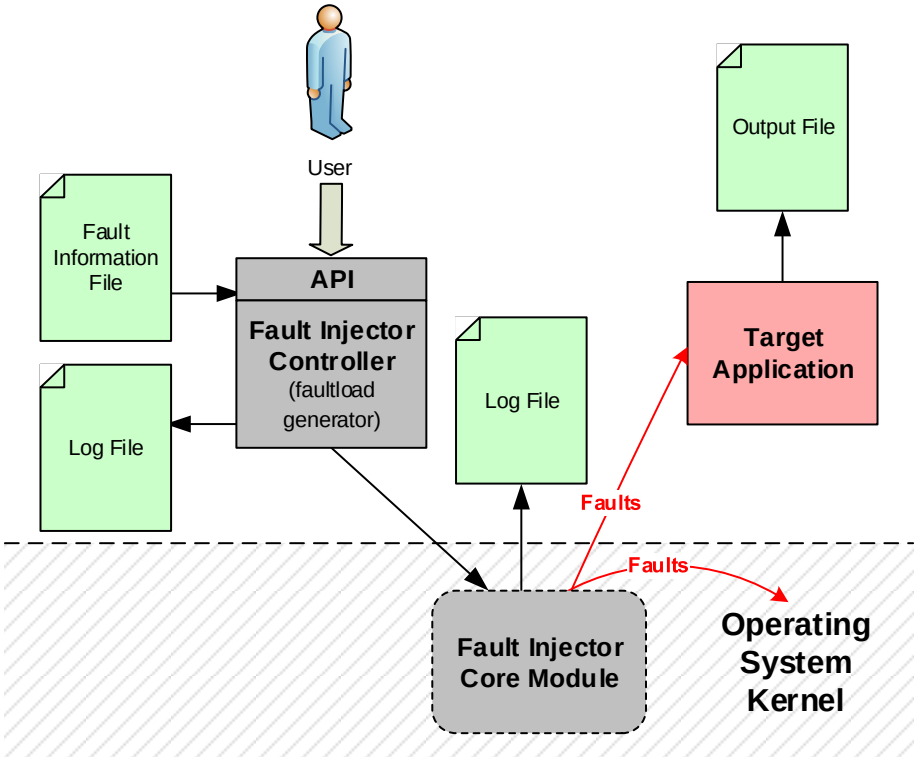


Figure 4-1 – The DBench-FI fault injector architecture.

The *core module*, dynamically linked with the kernel, is responsible for implementing the runtime kernel upgrading algorithm in order to add the fault injection functionality to the system, independently of any debug mode. The new kernel, incorporating this module, provides the user the capability of injecting faults into any process (in either data or code

segments) running on the target system, including those that are part of the operating system itself. The user interface is given by the *fault injector controller module*. It is worth noting that the integration of the *fault injector core module* with the OS kernel enables the injection of faults in the system space, in addition to the user space.

All the information necessary for the fault injection process, such as the identification of the target process (through the process *pid*), the desired fault model, the type of faults to be injected (for example, stuck-at-0, stuck-at-1, bit-fip, etc.), the target address range, among others, are sent to the *core module* through the *fault injector controller module.*

When integrated in a dependability benchmarking, the *fault injector controller module* is responsible for providing the API to the Benchmark Management System (BMS), becoming the faultload generator of the system. The target system with these two modules (the *fault injector core module* is integrated in the kernel) provides the user the ability to inject faults in whatever process that is already in execution, including those that are part of the OS itself.

It should be noticed that there is no restriction on the fact that both modules have to reside on the same machine. They may be placed in different machines, if necessary for a particular experiment.

4.3 Fault Injection Design and Implementation

The fault injector has been implemented on an Intel Pentium IV system running the Linux RedHat 7.3 (kernel version 2.4.18-3). It has also been tested with Linux RedHat 9 (kernel version 2.4.20-8) and Ubuntu 10.04 (kernel version 2.6.32-31). The dynamic algorithm responsible for the

linking of the fault injector with the OS kernel was implemented using Linux Loadable kernel Modules (LKMs)[7].

The DBench-FI fault injector is based on common characteristics and concepts of modern preemptive multitasking operating systems, which explains its high level of portability, not found in other SWIFI tools. For reasons that are explained below, two mechanisms of modern operating systems are of particular importance in the methodology used by DBench-FI: the memory management mechanism, where any process running on the system is viewed as having its own memory address space, and the process management mechanism, responsible for the implementation of the abstraction which consists on the existence of multiple processes seemingly running simultaneously, even on systems with a single processor. A thorough description of the components and mechanisms of the Linux kernel are described in [Mauerer 2008, Kerrisk 2010, Love 2010].

---

[7] Loadable Kernel Modules allow a running operating system kernel to be dynamically extended, increasing its flexibility concerning the addition of new hardware support or functionality. They are usually used by device drivers and filesystems. Currently, most modern Unix-like operating systems, such as *Solaris*, *Linux* and *FreeBSD* use or support LKMs.

It is worth pointing out that, in Linux, like in all monolithic architectures, the operating system functionality is concentrated within the kernel. Regarding the architecture of the OS kernel, it should be noticed that Linux is considered essentially monolithic[8], as it is packed in a single, large, binary image, which includes all its subsystems such as process management, memory management, file systems, etc., and runs in a single address space[9]. However, at the same time, the Linux kernel is also modular, as it supports the dynamic insertion and removal of code from itself at runtime, and thus compensating some of the known disadvantages of the monolithic kernels[10]. As a consequence, Linux kernel is not

---

[8] Despite the Linux kernel incorporates both monolithic and microkernel ideas, it was originally developed according the monolithic paradigm in order to avoid the need to develop a message passing mechanism and a module loading architecture, and accelerating the achievement of a ready-to-run and fully operational OS [Maxwell 2002].

[9] The great majority of commercial Unix variants are monolithic. Most notable exceptions are the *Carnegie-Mellon's Mach 3.0*, as well as other Unix-like systems based on this microkernel, such as the *MAC OS X* and the *GNU Hurd* operating systems, which follow a microkernel approach [Bovet *et al*. 2005].

[10] The supporters of monolithic kernels argue a greater efficiency and performance in module communication, made through the direct call of functions (in kernel mode, in same

considered a pure monolithic kernel, as it incorporates both monolithic and microkernel ideas.

The kernel function responsible for deciding the next executable task that will be dispatched to the CPU, known as *schedule*, assumes a special role in the design of DBench-FI. The *schedule* function is called in the following circumstances: (i) a task yields the processor; (ii) a task blocks in an I/O operation; (iii) a task uses up its time slice (*quantum*); or (iv) a task is

---

address space), when comparing to the overhead caused by the necessary message-passing mechanisms that must exist between the various processes of a microkernel. On the other hand, microkernel supporters claim that they force system programmers to use "clean" and modularized programming approaches, which leads to an improved ease of development of new system modules. Other benefits of the microkernel architecture are the dynamic extensibility of the kernel and the ability to swap kernel components at runtime, and, consequently, a more efficient use of the system memory, since the modules are only loaded when they are actually required. These characteristics support the increased flexibility, portability and maintainability of microkernels design when compared to the monolithic variants.

preempted by another task (with higher priority[11]). Figure 4-2 gives a common view of the Linux kernel architecture, focusing on the interaction between applications, scheduler and hardware.

Concerning the design and implementation of DBench-FI, another important characteristic is the Linux memory management system, which is made-up to be architecture independent. As any modern multitasking operating system, the Linux kernel provides memory protection mechanisms (vital to the system stability), which prevent any attempt, on behalf of a user process, of illegitimate access to a memory area that belongs to another user process or to the kernel itself. Moreover, any user process running on the target system is regarded as having its own virtual

---

[11] Although the Linux kernel is preemptive (user mode processes may always be interrupted), there are some kernel critical regions which cannot be preempted by the scheduler until its execution ends. For this reason the Linux kernel is said to provide soft real-time behavior (its kernel tries to schedule applications within timing deadlines, although it may not always get it). Usually, fully preemptive kernels are associated with hard real-time operating systems, since they ensure the compliance with very stringent timing requirements for scheduling.

memory address space[12], which includes its code, data and stack areas. A representation of a user process address space in Linux is shown in Figure 4-3. It is worth noting that the kernel is mapped in the address space of every process, in the top area of its memory address space (from TASK_SIZE[13] to 2^{32} or 2^{64}, in IA-32 systems or IA-64 systems, respectively).

---

[12] Virtual memory is referred as the practice of lying to processes about the real (physical) addresses at which they reside. To each user process is given the illusion that its address space always starts at 0 and extends from there. It is worth noting that some purists differentiate the concept of virtual memory from the notion of "disk-as-memory". In fact, although the virtual memory is usually associated with swapping and paging techniques, it can be, in *sensu stricto*, differentiated from them (the latest techniques refer the OS ability of blending primary and secondary storage, providing to processes the use all of its memory as if it were always available): an OS can give each process a logical address space without making any association between primary and secondary storage [Maxwell 2002].

[13] In Linux, every user process has its own virtual address space ranging from 0 to TASK_SIZE (an architecture specific constant defined as a kernel symbol, which represents the maximum size that a user process can access in bytes, i.e., since the space address always starts at 0, it assumes the maximum address that a user process can access+1). On IA-32 systems, for instance, the TASK_SIZE assumes the value of 3 GiB (i.e., 3×2^{30} bytes).

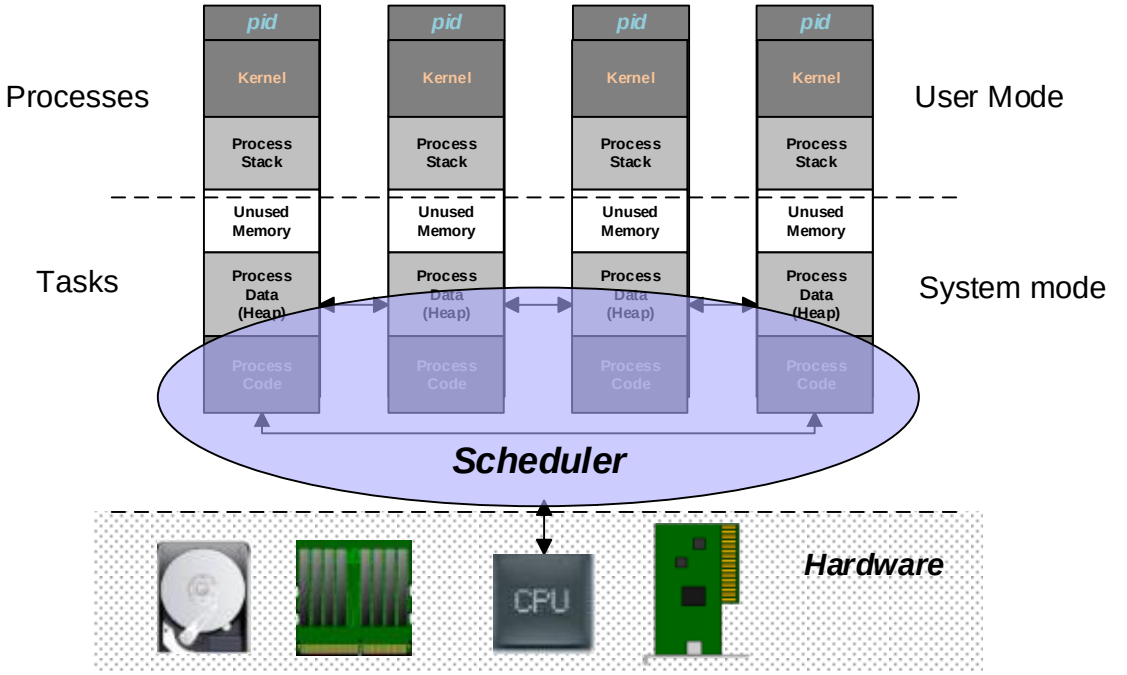


Figure 4-2 – The Linux operating system architecture.

Concerning the mapped regions, for a correct understanding of the interconnection of the fault injector and the memory management functions of the OS kernel, it is important to point out the most significant differences that they have with each other. The code segment, referred as *Process Code* in Figure 4-3, is write-protected and shared by all processes that execute the code it contains. This represents a significant difference when compared to the remaining areas (data and stack), which are private to each process and where writing is allowed. Another fundamental distinction between the code area and the data and stack areas relates to the fact that the first cannot be dynamically reserved. In fact, a Linux user process can dynamically allocate three types of memory: *stack*, *heap* and *mmaped*

memory[14]. A thorough description of the components and mechanisms of the Linux kernel are described in [Mauerer 2008, Kerrisk 2010, Love 2010].

As already mentioned, the DBench-FI was initially developed for the purpose of injecting faults in the memory address space of a given process. In its first version, presented in [Costa *et al.* 2003], it is possible to inject stuck-at-0, stuck-at-1, and bit-flip type of faults in the data segment of any user process (as well as on its stack area). Thereafter, it was added the ability to inject faults in the code segment of any process, as well as the possibility of the injected faults that assume a user defined value through a fault information file, as depicted in Figure 4-1 – The DBench-FI fault injector architecture. In the context of the software fault emulation, the

[14] The range of valid virtual addresses of a process can change throughout its lifetime, as the kernel allocates and deallocates memory according to its needs. A process can allocate memory by increasing the size of the heap - raising the *program break* (the current limit of the heap), through the use of the *brk()* and *sbrk()* system calls (upon which the well-known malloc functions are based). A process can also create and free memory mappings into its virtual address space, using the *mmap()* and *munmap()* system calls, respectively. The process stack dynamically grows and shrinks as functions are called and returned. Special process registers are used for this purpose, as explained later on this chapter.

possibility of using this new type of faults, together with the possibility of targeting the code segment of any process, enables the use of more representative fault models. In fact, these improvements provided the compatibility of DBench-FI with the state-of-the-art in software faults model – the mentioned G-SWFIT, presented in [Durães *et al.* 2006].

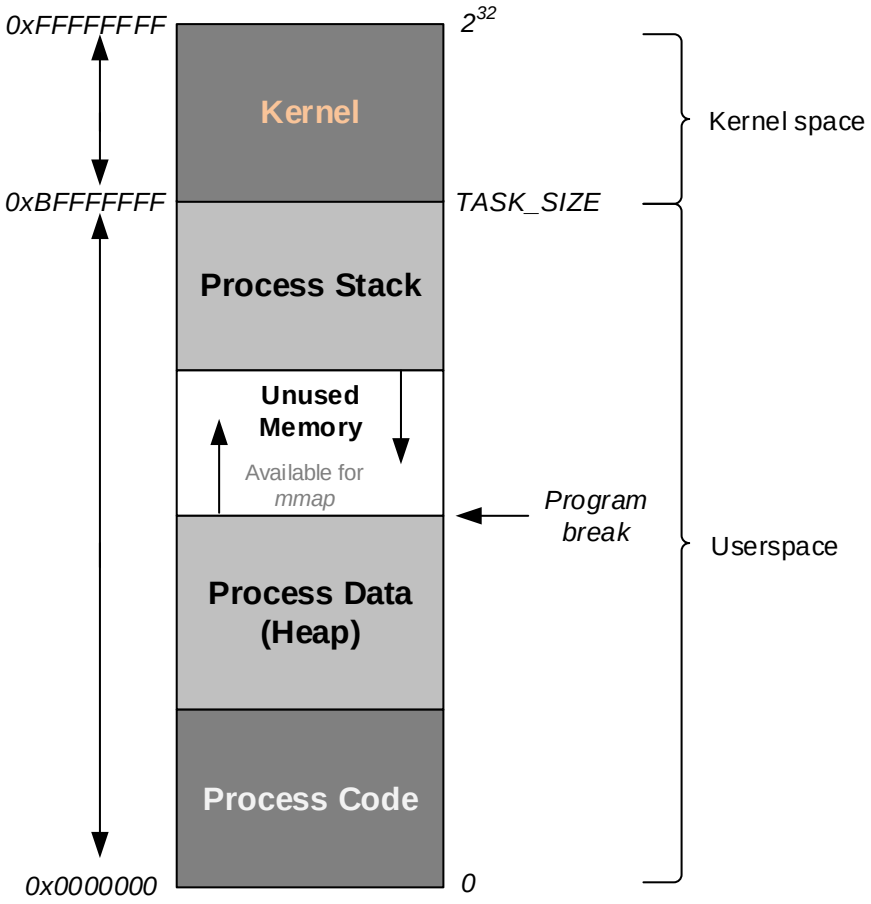


Figure 4-3 – The process virtual address space in IA-32 systems.

It is worth noting that, as expected, these latest enhancements did not involve any change in the methodology or in the model of the fault injector. It should be also emphasized that, in consequence of the possible share of the code segment across multiple processes, the faults injected in that area may affect the behavior of all processes which share that region.

Concerning the design and implementation of DBench-FI, as one of the goals of DBench-FI consists on injecting faults in the address space of any process, including the operating system kernel itself, two different solutions were initially considered, both based in a new process running in kernel mode:

- The interception of the OS scheduler and the detection of the target process in order to access its virtual address space. It is worth noting that the virtual address space of a process is only available when that same process is chosen by the schedule function to use the CPU;
- Access the memory area of the target process through the lookup of the corresponding *page table* entries used by the memory management system of the OS. It is worth pointing out that the OS kernel maintains a *page table* for each process, in order to map the virtual addresses of a process to the corresponding physical addresses.

Reasons of clarity, elegance and portability, justified the choice for the interception of the OS scheduler (the first solution considered). In order to detect the time when the target process was chosen to use the CPU, and its virtual memory address space is available for the injection of faults, the DBench-FI dynamically intercepts and changes the OS schedule function. The required fault can then be injected.

In a first step, the address of the kernel *schedule* function is found, and then redirected to a new function called *new_schedule*, responsible for both the target process detection and the fault injection. The memory address where the schedule function resides is determined through a search in the

Linux file */proc/ksyms*[15], which contains a list of every symbol that is exported by the OS kernel (known as *kernel symbol table*)[16]. This methodology presents a higher degree of portability across different versions and distributions of the Linux OS, when compared, for example, with the memory pattern search algorithm used in the first version of the fault injector [Costa *et al.* 2003]. However, this approach requires that the used kernel supports LKMs, which are, however, also required for the dynamic installation of the *Fault Injector Core Module*. Moreover, considering the benefits of the dynamic extensibility of the kernel, typical of the microkernel architectures, most of the current Linux kernels and distributions are compiled with this option enabled, which is indeed considered as default. It is important to mention that the used methodology requires supervisor privileges, since both the accesses to the LKMs features and to the */dev/ksyms* file demands it for security reasons.

---

[15] The Linux file */proc/ksyms* is created on-the-fly when the kernel boots up. For Linux kernels version 2.6, and above, the */proc/ksyms* file was replaced by */proc/kallsyms*.

[16] The file */boot/system.map* could also be used for this purpose, since it contains all symbols used by the kernel. However, this file is usually used for debugging purposes and, sometimes, it is not available (as it is not required for the OS booting process).

The procedure used by DBench-FI is illustrated in Figure 4-4 and consists of the following steps:

1) Determine the runtime address of the schedule kernel function on the OS kernel symbols table;
2) Copy the first nine bytes of the kernel schedule function (represented by instructions A, B and C in Figure 4-4) to a new function called *saved_instructions*;
3) Generate a *jump* instruction with the runtime address of the *new_schedule* function (where the target process detection and the fault injection will take place) and overwrite the first bytes of schedule code with the generated *jump* instruction;
4) Create a *jump* instruction in order to execute the saved nine bytes of the kernel schedule function (saved in step (2) to *saved_instructions*) after the execution of *new_schedule*;
5) Create a *jump* instruction in order to execute the rest of the original schedule function code (from the 10th byte forward of the original schedule function).

It should be noticed that, considering the methodology used by the fault injector, as well as the implementation of the *new_schedule* function in a high level language (C language), it is fundamental to restore the *stack* after the identification of the target process and before the *jump* (step 4) to the original schedule instructions (saved in *saved_instructions*). Such need is justified for the following two reasons:

1. The compiler, according to the calling conventions, automatically creates a prologue and an epilogue, which allows the use of the *stack* for passing data between the caller code and the called subprogram;

2. The function *new_schedule* is finished with a *jump* to *saved_instructions* (step 4) instead of using the conventional epilogue[17].

It should also be noticed that when the fault injector kernel module is loaded, the policy and the main algorithm of the original operating system scheduler remains the same. Additionally, when it is unloaded or removed, the redirections that were made are undone and the scheduler becomes exactly the original.

Concerning the intrusiveness, it is important to enhance that when the fault injector is loaded but no faults are injected, the performance penalty corresponds to ten machine assembly instructions that were added in order to intercept and redirect the scheduler. This fact guarantees a very low and totally negligible intrusiveness, considering the current processors.

---

[17] The x86 family processors have two general-purpose registers in order to manipulate data on the stack: the ESP and the EBP. While the first register points to the top of the stack, the second is used to reference data on the stack. At the end of a subprogram, the original values of the registers are restored (they are previously saved at the start of the subprogram). Detailed information about the stack and the calling conventions are presented in [Carter 2006]

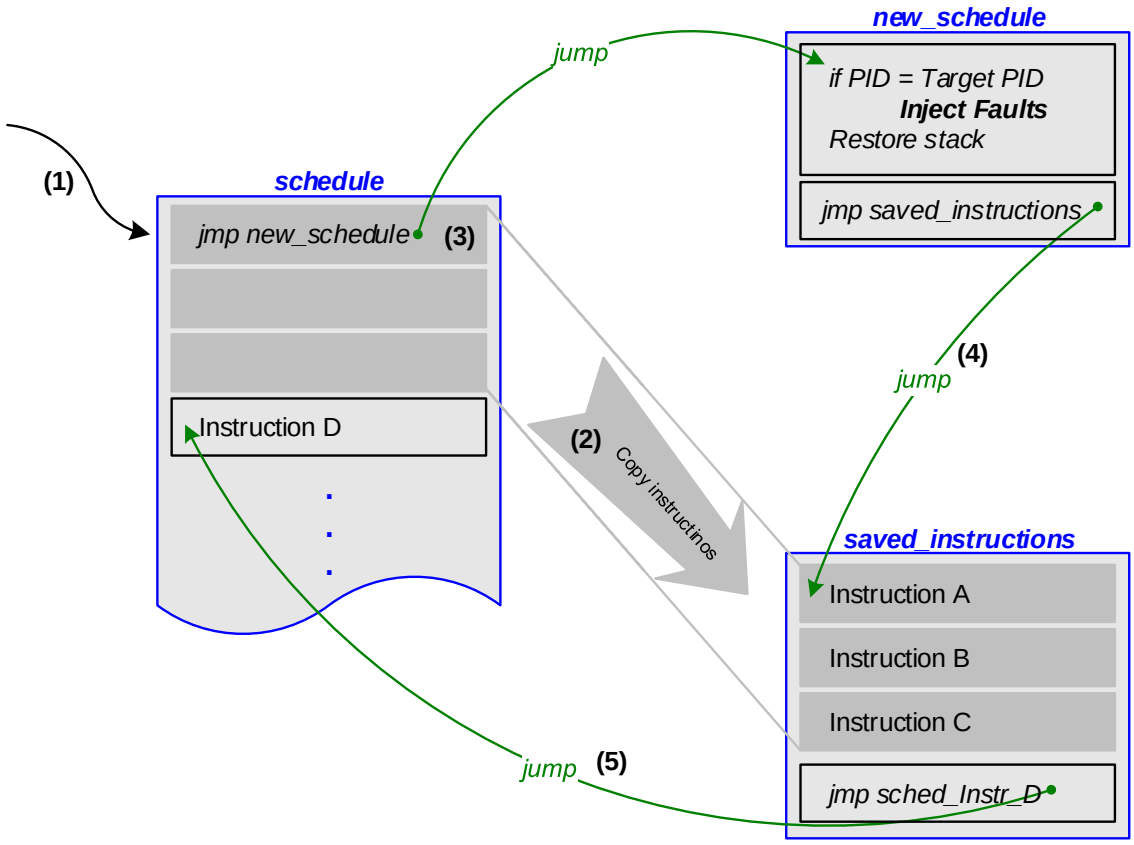


Figure 4-4 – The DBench-FI fault injector methodology.

4.4 Using the DBench-FI

The DBench-FI fault injector consists of two main files: *dbfi_drv.o* and *dbfi_controller*, corresponding to the *Fault Injector Core Module* and to the *Fault Injection Controller*, respectively. The use of the Dbench-FI fault injector involves two steps:

1. Loading the *Fault Injector Core Module* using the facilities provided by the LKMs;

2. Executing the *Fault Injection Controller*, providing the required parameters for the fault injection campaign through a command line with the following syntax:

```
dbfi_controller [{target_pid start_addr end_addr
nbytes init_t reg_t maxfi type | -f filename
| -gswfit filename fi_num}]
```

The command dbfi_controller can be executed by itself, without any argument. Thereby, all the fault injection parameters will be provided in an interactive way.

Though, the fault injection parameters can also be specified in the command line, through arguments, using the syntax:

```
dbfi_controller target_pid start_addr end_addr
                nbytes init_t reg_t maxfi type
```

The command arguments are explained below:

- *target_pid*: Identifier (*pid*) of the target process. Zero indicates that the fault will be injected in the kernel address space – one of the mentioned requirements;
- *start_addr*: Initial address (virtual) of a contiguous memory block that will be a potential target of fault injection. It should belong to the set of virtual addresses actually used by the target process (indicated in *target_pid*). Otherwise, in the case of any of these memory addresses actually be the target of fault injection (see the explanation of *nbytes* bellow), an appropriate error message will be sent to the user, referring that the address is not in use by the specified process;

- *end_addr:* End address (virtual) of a contiguous memory block that will be a potentially target of fault injection. As in the previous case, it should belong to the set of virtual addresses actually used by the target process (indicated in *target_pid*). Otherwise, an appropriate message will be sent to the user, as explained for *start_addr*;
- *nbytes:* Number of bytes, within the specified memory block (from *start_addr* to *end_addr*), that will be actually targeted by the fault injection campaign. Zero indicates that the entire block will be actually targeted, i.e., from *start_addr* to *end_addr*. If the value specified is less that the number of bytes of the memory block defined by *start_addr* and *end_addr*, a random location of contiguous *nbytes* within that memory block will be used as the actual target;
- *init_t:* Time, in seconds, that will elapse before the first fault injection take place;
- *reg_t:* The frequency of fault injection, in seconds. Zero indicates the use of temporal random fault injection triggers. A value of *n*, other than zero, indicates an interval of *n* seconds between fault injections;
- *maxfi:* Maximum number of fault injections;
- *type:* Type of faults that will be injected in the virtual address space of the target process. Values of 0, 1 or *bf*, indicate stuck-at-0, stuck-at-1, and bit-flip, respectively.

Concerning the fault triggers used by the DBench-FI, considering the options provided by the *Fault Injection Controller*, and more specifically, thought its *init_t* and *reg_t* arguments, the activation of the fault injection is based on temporal trigger conditions. This fact ensures the independence of the injected faults with respect to any specific activity of the target

application. Related to temporal fault triggers, two options are available in this implementation: a fault is injected once after a given time is elapsed since the application starts (using the *init_t* parameter), or a fault is repeatedly injected with a certain frequency (using the *reg_t* parameter with nonzero value). Temporal triggers that are randomly chosen (enabled with *reg_t* equal to zero) are particularly adequate to benchmarking, as they enable statistically significant results to be obtained. This way, regarding trigger conditions, two options are available:

1. The fault is injected only once (with *maxfi* equal to one), after a certain time, in seconds, set by the user thorugh the argument *init_t*;
2. The fault is injected repeatedly, after a given initial time (in *init_t*), in a certain frequency, random or user specified (with *reg_t* equal to zero or given it a nonzero value, respectively).

The parameters used for the definition of the fault injection campaign can also be specified in a text file (as showed in Figure 4-1) with the following format (all the parameters must be in the order shown, separated by space characters):

target_pid start_addr end_addr nbytes init_t reg_t maxfi type

In this case, the syntax should be

dbfi_controller -f *filename*

where *filename* is the name of the mentioned text file.

As explained, the current version of DBench-FI is also compatible with the state-of-the-art G-SWFIT software faults model [Durães *et al.* 2006]. In order to use this feature, the following syntax should be used:

```
dbfi_controller -gswfit filename fi_num
```

where *filename* is the name of a G-SWFIT format file containing the full set of software faults that can be injected in a system and *fi-num* is the number of the software fault that will actually be injected in the target system.

The identified file consists of a text file with one software fault per line. The specification of each software fault consists on an asterisk terminated string, with the corresponding fault injection parameters separated by commas, according to the following format:

*Type, Level, Arg3, Inj_Method, Addr, Nr_Bytes, Orig_Bytes, New_Bytes, # Comment1 # # Comment2 # … # Comment n #,**

The listed parameters have the following meaning:

- *Type*: Identifies the type of the software fault according the G-SWFIT model [Durães *et al.* 2006]. It can assume values like MIFS, MFC, MIA, etc.;
- *Level*: Concerning the level of depth of the G-SWFIT pattern search in each target software component or module. Level zero indicates that the pattern will only be performed directly on the target code. Level one indicates that the pattern search will be performed on the target code and on functions that are called by them. And so on. It is worth pointing out that level zero was used for the purposes of the current work. Greater values would lead to the repetition of some fault injection experiments and, consequently, to a non-homogenous distribution of faults, which will be inadequate in the context of this study;
- *Arg3*: Used for compatibility with the G-SWFIT faultload output file format. It represents the number of contiguous blocks of bytes

that would be changed by fault injection. According to the G-SWFIT methodology it should be equal to one;

- *Inj_Method*: Used for compatibility with the G-SWFIT faultload file format. Regarding the G-SWFIT model it should be set to SUBS (indicating that a block of bytes will be *substituted* by another, according to the low-level code mutations defined by the set of operators of the G-SWFIT methodology);
- *Addr*: The start address (hexadecimal) of the block of bytes that will be the target of the low-level code mutation for the emulation of the software fault according to the G-SWFIT methodology;
- *Nr_Bytes*: The number of bytes of the block that will be the target of the low-level code mutation for the emulation of the software fault, according to the G-SWFIT methodology;
- *Orig_Bytes*: The original bytes (hexadecimal) of the target that will mutated using the set of low-level operators for software fault emulation, according to the G-SWFIT methodology;
- *New_Bytes*: The new bytes (hexadecimal) that will be injected in the block defined by *Addr* and *Nr_Bytes*, according to the low-level operators of the G-SWFIT methodology.

The last section of the line, between the last pair of commas, is intended for posting comments, which are useful to increase human readability. In that section, each comment should be inserted between a pair of hash characters. The following line shows an example of a real G-SWFIT software fault specification:

*MIFS,0,1,SUBS,c0106ed0,2,7402,EB02,# je c0106ed4 # # MIFS c0106e60 <machine_real_restart_R3da1b07a>) #,**

In this case, a MIFS (*missing if statement*) is emulated at address *0xc0106ed0*, substituting the bytes *7402* by *EB02*. The comments indicate that the original machine code instruction is *je c0106ed4* located at function *machine_real_restart*, whose start address lies on *0xc0106e60*.

However, for an additional simplification of the process of installing and using the fault injector, it was created a script called DBenchFI. It is responsible for the loading and removal of the *Fault Injector Core Module* and the execution of the *Fault Injection Controller*, plus offering the possibility of identification of the target process by name and user to which it belongs. The syntax used is as follows:

```
DBenchFI [{-n|-p}] target_proc [-u user_id]
start_addr end_addr nbytes init_t reg_t maxfi
type

DBenchFI -f filename

DBenchFI -gswfit filename fi_num

DBenchFI -e
```

The options and arguments of the script DBenchFI have the following meaning:

-n The identification of the target process is done through the process name. That is, *target_proc* indicates the name of the target process.

-p The identification of the target process is done through the process pid, given in *target_proc* (default option).

-u Indicates that the target process belongs to the user mentioned in *user_id*. It is useful for resolving the ambiguity caused by

the existence of multiple users running the same process target. In the case that there is no univocal correspondence between *target_proc* and a particular target process, even after the specification of the user through *user_id*, the oldest process will be chosen (using the process creation date to resolve the ambiguity).

-e Used for removing the *Fault Injector Core Module*.

All the remaining arguments have direct correspondence with their homonymous for the *Fault Injector Core Module*, *dbfi_controller*, and have the same mentioned meaning.

4.5 Advantages

The methodology used in the design of DBench-FI confers it a number of important advantages (compared to the other existing fault injectors) regarding its inclusion in a dependability benchmark. Since the methodology it relies on is based on the interception the OS kernel scheduler and its redirection to a function that is within the kernel itself, DBench-FI is appropriate for the injection of faults into any system memory address, including the kernel memory segment. This capability makes this fault injector suitable to analyze the kernel robustness under faults, and represents a huge advantage comparatively to the *ptrace*-based SWIFI tools. Another important benefit relatively to the fault injectors based on the *ptrace* mechanism is that DBench-FI can inject faults into any running target application without having to load it explicitly or using any special procedure to execute it. It is worth noting that the fault injectors based on this function, like in any other debug tool, only allows the injection of faults in the user segment of target processes that they can explicitly launch. That is, DBench-FI allows the fault injection in processes that are already running when the fault injector is installed, regardless of the complexity of

the application they are part of. This is an essential requirement to analyze the dependability of complex systems like DBMSs and web-servers.

An important issue with SWIFI tools is their portability to other systems and processors. The proposed methodology can be, with some minor changes, adapted to almost any operating system and processor. A further advantage is the simplicity and ease of use of DBench-FI, since it does not require any special procedure. In particular, there is no need to recompile the kernel or the target application, nor the knowledge of the source code of any of them. Concerning intrusiveness, the presented methodology provides the fault injector a negligible disturbance factor on the target system.

The compatibility of DBench-FI with the G-SWFIT technique [Durães *et al.* 2006] is an important characteristic of this fault injector. This fact represents a major advantage when compared to other existing SWIFI tools, as, like stated in [Madeira *et al.* 2000, Jarboui *et al.* 2002], this kind of tools are not an obvious choice for the emulation of software faults.

4.6 Limitations

The main limitation of DBench-FI, besides the general SWIFI limitations described in section 2.4.2, is the limited set of fault models supported. This is not a limitation of the technique itself, but just of the current implementation, as the compatibility with the G-SWFIT model was considered more important for dependability benchmarking. However, if necessary, DBench-FI can easily be extended with the majority of the existing fault models of Xception [Carreira *et al.* 1998b], such as spatial fault triggers and the capability to inject faults in processor resources.

Another obvious limitation of the presented technique is the fact that supervisor level privileges are required to install and use the fault injection

tool, as operating system security rules understandably prevent user level processes from modifying the kernel.

It is worth to pointing out that, beyond the intrinsic restrictions that applies to the SWIFI tools, no other limitation is related to the used methodology, but rather with the current implementation of the fault injector.

4.7 Summary

Despite all the developments in the area of software fault injection, none of the existing SWIFI tools has characteristics compatible with the creation of a dependability benchmark.

This chapter presented a pioneering SWIFI tool, named DBench-FI, whose innovative features allow its use in dependably benchmarks. Its architecture and implementation details are also described, as happens to some features of operating systems and processors in which its development is based. Their unique characteristics make it one of the most versatile fault injectors available and a central tool for the study presented in this thesis.

The current version of DBench-FI is adequate for the injection of hardware faults (intermittent and transient faults) into the systems memory, as well as for software faults, according to the G-SWFIT model.

Chapter 5

Software Faultload for Large and Complex Systems

This chapter describes the problem of injecting realistic software faults in large and complex systems and puts into perspective the still open problem of the faultload size. It surveys the existing approaches that address this issue, discussing their strengths and limitations. Finally, it presents and provides and early assessment of an innovative experimental framework to define and evaluate different strategies for the definition of compact and representative faultloads. In this context, different hypothesis for the reduction of the number of software fault injection experiments are defined and an evaluation method of the error induced by the corresponding reduction is also presented. The proposed methodology is especially useful in large and complex systems, where dependability benchmarks usually take several months or even years due to its large faultload size.

5.1 Introduction

One of the main goals of dependability benchmarks is to offer practical and efficient methods to characterize the behavior of components and systems and quantify dependability measures,

considering the computing effort, the number of experiments and the time to run the benchmark.

Concerning software systems, most recent techniques such as G-SWFIT, firstly presented in [Durães *et al.* 2003b], and later in [Durães *et al.* 2006], use a set of operators for software fault emulation through low-level code mutations derived from an extensive collection of real software faults found in field. Although this innovative proposal emulates and represents real programming errors and application bugs, the sets of faults they generate tends to have a huge size, as it obviously happens with the resulting software faultload. This imposes a strong limitation to the execution of dependability benchmarks in software systems, especially in large and complex systems, where, in order to assure the necessary representativeness, the execution time of those benchmarks can take months or years due the mentioned faultload size. In fact, the great majority of studies on representativeness of software faults, mentioned in the previous chapter, just addressed the problem of finding realistic software fault models and ignored the important problem of the faultload size.

5.2 Fault distribution models

Despite some recent studies on software fault injection addressed the problem of finding realistic fault models, the problem of how to distribute the faults among different components in target systems have barely been discussed. Some recent studies on software fault injection use exhaustive fault coverage for small software components, injecting all the possible software faults, yet the most representative types, [Durães *et al.* 2004a, Durães *et al.* 2004b, Costa *et al.* 2009, Natella *et al.* 2013]. However, more sophisticated fault distribution models are needed when dealing with large components and systems, such as operating systems.

Large components and systems induce huge size faultloads, due to the vast number of possible software fault types and target locations, which could make impractical the fault injection campaign. For example, the software fault injection campaign carried out in the present work, presented in chapter 6 (Experimental Evaluation of Faultload Reduction Strategies), encompasses tens of thousands of software faults and resulted in more than two years of fault injection experiments. That problem, one of the currently most important issues in fault injection, and particularly in the area of software faults, has been largely neglected in the literature. Some exceptions are presented below.

A similar problem arose earlier in mutation testing, where the large number of experiments, induced by the large number of mutants that need to be compiled and executed against test cases, especially in large and complex systems, soon became a barrier to the practical use of this technique in identifying adequate test data. It is worth noting that, although there is evidence on the use of the mutation testing technique in increasingly larger programs, those empirical studies applied only a few mutations operators [Jia *et al.* 2011]. In fact, in order to turn the mutation testing into a practical testing technique, and reduce the high computational cost of executing the huge number of mutants against a test set, several studies only use a subset of the potential mutants for a given program, representing a subset of all the possible faults, expecting that these will be sufficient to simulate all faults.

Several approaches on the selection of a sufficient set of mutation operators were presented. Traditional approaches target only a few simple faults, constructed from several simple syntactical changes, which are close to the correct version of the program. This theory is based on two empirical principles first introduced by [DeMillo *et al.* 1978]: the Competent Programmer Hypothesis, and the Coupling Effect. The first principle states that programmers are competent and, consequently, they develop programs that tend to be close to the correct versions, as a result of their

multiple iterations through the software development process. On the other hand, the Coupling Effect states that test cases able to detect mutated programs differing from a correct one only by a simple error (fault), are so sensitive that they also implicitly detect more complex errors. In other words, it assumes a principle observed in real world programs, which states that complex errors are coupled to simple errors.

Another simple technique for the reduction of the number of mutants is the mutant sampling. It consists in the selection of randomly chosen mutants from the entire set. Many empirical studies addressed this approach, analyzing the appropriate random selection rate, and minimal sample size, that should be used in order to maintain its usefulness [DeMillo *et al.* 1988, Sahinoglu *et al.* 1990, King *et al.* 1991].

The reduction in the number of mutants through the reduction of the applied number of mutation operators was firstly proposed as Constrained Mutation, by [Mathur 1991]. The methodology consists in the reduction of the mutation operators set by omitting those that generate most of the mutants, since many of which may turn out to be redundant [Offutt *et al.* 1993, Offutt *et al.* 1996][18]. Another type of selection strategy, based on test effectiveness, is presented in [Wong *et al.* 1995].

---

[18] In [Offutt *et al.* 1993, Offutt *et al.* 1996] the method was called *Selective Mutation*.

In [Mresa *et al.* 1999] is used a heuristic based on scores and costs assigned to each mutation operators to choose a subset of operators for use in efficient selective mutation testing. This study takes into account both the costs of the test set generation and of the detection of equivalent mutants. The experiments carried out show that it is possible to reduce the number of equivalent mutants while maintaining the effectiveness.

A guideline for the determination of a sufficient set of mutation operators for C programs is presented in [Barbosa *et al.* 2001]. The results show that set of operators can be reduced by about 65%, while maintaining a mean mutation score of 99.6%.

The studies presented in [Namin *et al.* 2006, Namin *et al.* 2007, Namin *et al.* 2008] use a statistical analysis procedure together with an associated linear model that predicts mutation adequacy with high accuracy, to address the problem of finding an adequate small set of mutation operators. The results presented in [Namin *et al.* 2008] indicated the identification of a subset of mutation operators that generates less than 8% of the mutants generated by the full set, consisting in the highest rate of reduction when compared to the other approaches.

A different approach to improve the testing effectiveness is proposed in [Sridharan *et al.* 2010]. This work presents a Bayesian approach that prioritizes mutation operators whose mutants are likely to remain "hard-to-kill" by the existing test suites.

Regarding software fault injection, the use of a dependence analysis approach to reduce the number of experiments necessary to test the robustness of COTS is presented in [Moraes *et al.* 2005a]. This work extends the one presented in [Moraes *et al.* 2004], where the idea of architecture relevance for testing a COTS-based system was firstly presented. The proposed strategy is based on *chaining* [Stafford *et al.* 1997] - a software architecture dependence analysis technique aimed to reduce the portions of a system architecture that must be analyzed for some purpose, such as

testing or debugging. The approach is applied for testing a COTS database component called Ozone [Ozone], an Object-Oriented Database Management System (OODMS), executing the OO7 Benchmark Wisconsin [Carey *et al.* 1993, Zyl *et al.* 2006], a well-known benchmark used to evaluate OODBMS performance. This work concludes that the dependency analysis was effective in helping the selection of the target classes.

The use of stratified sampling to reduce the amount of fault injections needed to test the robustness of the system without losing the confidence in the results is presented in [Moraes *et al.* 2005b]. Stratified sampling consists of a sample technique for partitioning a population into subpopulations called strata, by grouping elements with similar values for one or more characteristics [Podgurski *et al.* 1993]. This work uses the Weighted Methods per Class[19] (WMC) [Chidamdber *et al.* 1991], to determine the strata. For the mentioned purposes two different strata are considered in this study: one for components with a WMC value greater than a pre-specified threshold value, obtained in an experimental study with several real world classes [Rosenberg *et al.* 2000], and the other for lower WMC values than the same

[19] *Weighted Methods per Class* is an object oriented software complexity metric that consists on the sum of the complexities of all methods defined in a class. It represents the complexity of a class as a whole and can be used to indicate the level of time and effort required to develop and maintain a particular class.

threshold. Results show that the exclusive use of the WMC metric is insufficient to choose the strata and other stratification criteria should be used for robustness testing purposes.

A field data study on the use of software metrics to define representative fault distributions for software fault injection experiments is presented in [Moraes *et al.* 2006a]. The proposed methodology uses software complexity metrics and logistic regression [Hosmer *et al.* 1989] to estimate fault densities for each one of the modules of the target system and to distribute the injected faults. Seven software complexity metrics are used in this work: Lines of Code (LOC)[20] (comment lines were not considered for the current purposes), Cyclomatic Complexity[21] [McCabe 1976], number of function parameters, number of function return statements, Maximum Nesting Depth[22], Program Length[23] and Vocabulary Size[24] (the last two

---

[20] *Lines of Code* is one of the earliest and easiest (and also controversial) measures of software complexity. It consists on the count of the lines of the software's source code.

[21] *Cyclomatic Complexity* is a measure of module's independent control paths based on the mathematical graph theory. It is one of the most widely-accepted software complexity metrics.

[22] *Maximum Nesting Depth* measures the maximum indentation depth of module's source code (e.g., in C language measures how deep is the maximum { } nesting in the module)

metrics are part of a broader suite of metrics known as Halstead's Software Science Metrics or Halstead Metrics [Halstead 1977], and more precisely, represent two of the four equations needed to compute the Halstead's Programming Effort complexity measure). This study uses the G-SWFIT technique [Durães *et al.* 2006] in order to scan the target system code and identify all possible locations for the injection of each type of software faults, identified as being representative of real software bugs found in field. Table 5-1 shows the software fault types considered in the study presented in [Moraes *et al.* 2006a]. The accuracy of the fault distribution generated by proposed methodology was compared with real fault distributions observed in field, which includes over more than 350 bug reports available from open source software projects. The study concludes that the used approach is consistent with field observations, for small and medium size software modules. Regarding large and complex software modules, the fault density observed in field data showed to be lower than the estimated by the proposed methodology.

---

[23] *Program Length* is the count of total number of operators and operands in a module.

[24] *Vocabulary Size* is the count of total number of distinct operators and distinct operands in a module.

Defect type	Examples of code mistake
Missing	variable initialization (MVIV)
	variable assignment using a value (MVAV)
	variable assignment using an expression (MVAE)
	"if (*cond*)" surrounding statements (MIA)
	"AND expr" in expression used as a branch condition (MLAC)
	function call (MFC)
	"if (*cond*) { statement(s) }" (MIFS)
	"if (*cond*) statement(s) else" before statement(s) (MIEB)
	small and localized part on the algorithm (MLPC)
	functionality (MFCT)
Wrong	value assigned to variable (WVAV)
	logical expression used as a branch condition (WLEC)
	arithmetic expression in parameter of function call (WAEP)
	variable used in parameter of function call (WPFV)
	algorithm – large modification (WALL)
	data types or conversion used (WSUT)
Extraneous	variable assignment using another variable (EVAV)

Table 5-1 – Software fault types considered in [Moraes *et al.* 2006a].

Despite having a different purpose (to improve the representativeness of the faultload generated by the G-SWFIT approach), the research work presented in [Natella *et al.* 2013], as already mentioned, also proposes a methodology to generate a smaller and refined faultload by removing the faults that are not representative of residual faults.

5.3 Experimental framework

A representative faultload must be one that contains faults that represent the common programming bugs that escape the traditional software testing phases and still persist in existent software products. Although the faultload definition of that kind of faults had already been proposed, based on fault operators derived from the most frequent software fault types found in the field [Durães *et al.* 2006], the fault locations aspects have been completely neglected and the choice of adequate fault injection targets (i.e., actual software components where the faults are injected) is still an open and crucial issue.

Given a particular software fault type, existing techniques, like G-SWFIT, allow the injection of faults in every software module or routine with a specific code pattern, emulating a particular type of software fault. However, the common large number of possible target components for fault injection leads to a huge number of possible software faults to be injected. Additionally, considering the time of each experiment (typically, the system should be restarted before injecting a new fault), one can easily observe that, in practice, it is impossible to run and test all the fault injection possibilities. This problem is even more obvious in large and complex systems, where the execution time of those dependability benchmarks can take several months or even years due the mentioned faultload size.

One of the main goals of this work is to define a method to reduce the number of software fault injection target locations and thus the number of experiments needed to execute a dependability benchmark, without restricting or limiting the accuracy and the representativeness of its results. This methodology provides a way to perform accurate dependability benchmarks for large and complex systems, including COTS, which currently does not exist.

The proposed experimental framework is based on the definition of a strategy to guide the fault injection target selection and to reduce the number of software faults required for a dependability benchmark, or for an experimental evaluation using software fault injection. It consists of the following steps, based on two complete software fault injection experiments with two completely different systems, considering their complexity and the required computer resources of each one:

- Define different hypothesis for the reduction of the number of software faults to inject (for example, select a subset of faults at random and inject only those faults, inject faults only in the code of functions with greater lines of code, etc.).
- Evaluate each hypothesis in order to determine the best strategy to reduce the number of faults to inject with the minimum error possible, comparing the results obtained with the total fault set.
- Propose practical guidelines for the definition of faultloads with a number of faults that can be used in practice (instead of faultloads with thousands of faults that would take months to be injected).

A fundamental aspect of this approach is the clear separation between the fault injection target component and the system under observation, avoiding the problem of changing the system that is under evaluation. That is, the faults are injected in the FIT, with the goal of evaluating their impact on the rest of the system, the BT. Both the FIT and the BT are part of the SUB, a larger system that, from the benchmark point of view, consists in a set of processing units needed to run the workload.

Another key element of the proposed framework is the Benchmark Management System (BMS), which includes a component, called the Benchmark Controller (BC), responsible for the control of all the aspects of the benchmark experiments: workload submission, software fault injection, coordination and synchronization of the several components involved in the experiments and collecting the information needed to process the

benchmark measures. Another component of the BMS is the Benchmark Client through which the BC sends and controls the workload execution on the BT (Figure 5-1).

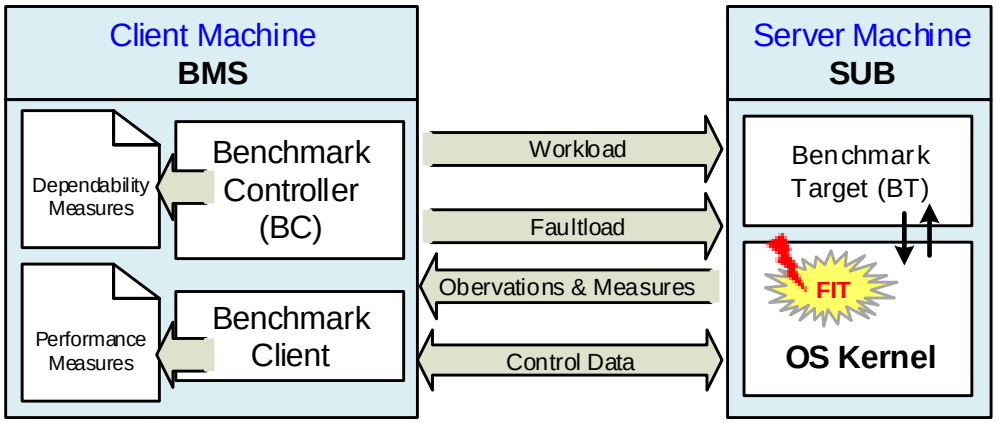


Figure 5-1 – Experimental Architecture.

5.3.1 Preliminary assessment study

In order to validate the proposed framework, an initial experimental study was carried out using the G-SWFIT fault model [Durães *et al.* 2006] and an early version of DBench-FI fault injector tool, especially designed and developed for dependability benchmarking (the current version of this tool was presented in chapter 4). This exploratory study on the guide of the fault injection target selection to reduce the number of faults required for the execution of dependability benchmarks is presented in [Costa *et al.* 2009]. It consists in the injection of software faults in the kernel code of the OS system calls used by two different benchmark target systems: (i) a web-server benchmark based on the SPECweb99 industry standard performance benchmark for web-servers [SPEC], extended with faultload and dependability measures (failure modes); and (ii) a client-server application to sort large-scale integer vectors, based on a Multithreaded Quicksort algorithm, extended with performance and quality metrics. This last system mainly serves as a control application and as a comparison

system, as it has some completely different requirements concerning to computer resources.

Four each BT system, there were analyzed four different hypotheses for the reduction of the number of faults to inject:

- Lines of code (LOC) of each targeted system call. It is worth noting that the analyzed LOCs were in machine code and thus some well-known restrictions, that are generally applied to this size oriented metric, like language and programmer dependence, no longer make sense in this context;
- The CPU time (CPUt) spent running in the kernel for each system call, relative to the SUB operating system, obtained during a normal execution, i.e. without any fault injection;
- The number of calls (NrCalls) made by the OS to each one of the system calls considered, during a normal execution of the system workload;
- A random selection of the software faults from the full set of faults, forming a subset of faults according to a uniform distribution.

The used approach consists of two phases as depicted in Figure 5-2:

- **A Pre-Injection Phase**, in which the benchmark is executed in order to identify the OS system calls used by the BT. The G-SWFIT faultload generator (software tool provided by the author of the study [Durães *et al.* 2006]) uses this list of OS system calls to identify all possible locations in the system calls code (it is worth noting that this code is part of the kernel) where it is possible to inject realistic software faults, according to the rules established by the G-SWFIT fault model.
- **A Fault Injection Phase**, in which, firstly, is injected the exhaustive set of software faults to obtain reference results, and

then are performed experiments to evaluate the error observed in the results for the different hypothesis of reduction of the number of faults.

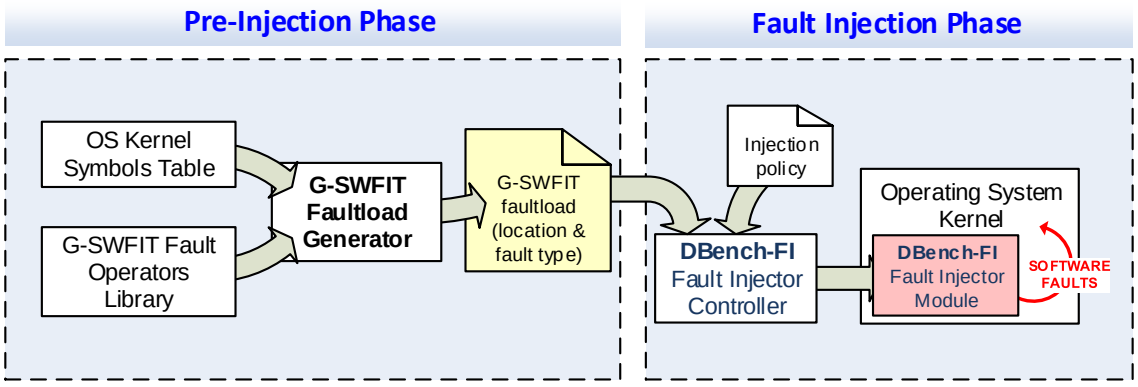


Figure 5-2 – Experimental methodology of the preliminary assessment study.

Regarding the Pre-Injection Phase, 50 OS system calls were used by the web-server system (see Table 5-2), whereas for the multithreaded quicksort algorithm, 27 OS system calls were found in use (see Table 5-3)[25]. For each BT system, Tables 5-2 and 5-3 also show the considered measures (LOC, CPUt and NrCalls) for the used system calls. As expected,

[25] All the OS system calls used by both BT systems were previously profiled and analyzed.

considering the computer resources used by each one of the BT systems, the web-server system revealed to be much more OS intensive than the multithreaded quicksort algorithm.

The results used in the experiments, also used to assess the error incurred by the reduction of the number of faults according to the different hypothesis, consist of the failure modes observed in each execution, from the external point-of-view of the Benchmark Controller. We consider the following well-known failure modes:

- **OK** – Representing the cases where the injected faults do not cause any kind of incorrect behavior in the SUB, neither in the benchmark measures, nor in the dependability ones. This failure mode is considered in most of the fault injection studies reported in the last decades;
- **CRASH** – Representing the cases where abrupt shut-down of the BT (process crash) is observed;
- **HANG** – representing the cases where the SUB is frozen, either the BT or the OS itself, and the experiment running time exceeded a predefined time limit;
- **ERRORS** – representing the cases where there is no hang or crash, but some incorrect results were observed by the BMS. More precisely, the Benchmark Client of the BMS detects errors in some of its requests.

In order to evaluate the error in the results, incurred by each different hypothesis on the reduction of the number of faults to inject, and determine the best strategy to reduce the faultload size without restricting the benchmark results, a deviation is calculated relatively to the values obtained for each failure mode with the injection of the complete set of software faults.

Web-Server Experiments							
System Call	**LOC** (#)	**CPUt** (secs)	**NrCalls** (#)	**System Call**	**LOC** (#)	**CPUt** (secs)	**NrCalls** (#)
read	94	10.689682	1,020,296	time	35	0.073008	10,332
lseek	50	4.513927	788,945	accept	80	0.393356	6,665
brk	92	4.836866	507,793	getsockname	43	0.085677	6,648
mremap	44	0.288144	204,735	shutdown	27	0.088628	6,648
close	29	2.173133	194,735	wait4	266	0.084326	5,530
open	54	1.711256	115,988	*geteuid32*	*5*	*0.076937*	*5,271*
old_mmap	99	1.299173	103,516	fork	11	0.291810	5,228
fstat64	21	0.813122	99,138	uname	41	0.068011	5,221
stat64	21	1.192313	91,806	execve	37	2.911191	5,169
fcntl64	62	0.410780	59,998	chdir	95	0.105711	5,169
poll	283	14.219557	53,543	lstat64	21	0.037768	5,168
munmap	31	0.612648	47,057	*getuid32*	*5*	*0.038894*	*5,167*
setsockopt	51	0.347240	46,860	*getgid32*	*5*	*0.034692*	*5,167*
rt_sigprocmask	142	0.325903	46,503	*getegid32*	*5*	*0.033765*	*5,167*
mmap2	56	0.347523	42,763	flock	58	0.035934	1,604
mprotect	193	0.447871	36,227	select	395	0.032389	361
getpid	*5*	*0.311780*	*32,996*	_llseek	77	0.009457	52
write	94	1.036779	27,883	setgroups32	39	0.013868	52
rt_sigaction	88	0.335088	25,999	setuid32	105	0.011012	52
gettimeofday	57	0.143740	22,855	setgid32	34	0.012440	52
writev	35	0.510542	20,212	socket	31	0.000943	45
sendfile	163	1.058075	16,988	connect	44	0.005816	45
pipe	37	0.183040	15,506	getsockopt	49	0.002144	45
dup2	62	0.187953	15,501	kill	24	0.000135	28
getrlimit	31	0.159430	10,386	unlink	94	0.000119	3
				Total (#50)	**3,520**	**52.603596**	**3,733,118**

Table 5-2 - System calls used by the web-server target system.

Multithreaded Quicksort Experiments			
System Call	**LOC** (#)	**CPUt** (secs)	**NrCalls** (#)
mmap2	56	5.165528	330,894
write	94	0.968005	21,351
getppid	*6*	*0.047278*	*4,476*
poll	283	0.099038	4,476
read	94	0.720567	4,468
rt_sigprocmask	142	0.325585	3,738
sigreturn	79	0.035028	3,703
wait4	266	0.253500	3,369
kill	24	2.490447	3,079
modify_ldt	43	0.223864	2,971
munmap	31	0.041457	2,021
mprotect	193	0.029026	2,021
clone	17	0.953065	2,020
rt_sigsuspend	85	0.178594	1,683
getpid	*5*	*0.009962*	*951*
old_mmap	99	0.000018	7
open	54	0.010015	5
brk	92	0.000014	5
rt_sigaction	88	0.000007	5
fstat64	21	0.000010	5
close	29	0.000012	4
time	35	0.000002	1
pipe	37	0.000022	1
uname	41	0.000003	1
_sysctl	54	0.000005	1
nanosleep	138	0.000001	1
getrlimit	31	0.000005	1
Total (#27)	**2,137**	**11.551058**	**391,258**

Table 5-3 - System calls used by the multithreaded quicksort target system.

The deviation df_i of each specific failure mode, relative to the failure mode rate value $\overline{f}_i$, obtained considering the complete set of targets, is calculated using

$$df_i = \sqrt{(x - \overline{f}_i)^2}$$

where f_i, for $i = 1, \ldots 4$, represents each one of the failure modes considered in this study (OK, CRASH, HANG and ERRORS), and x denotes the rate value obtained for that failure mode considering a subset of initial fault injection targets.

A global metric d_g is also used to measure the overall deviation from the failure mode values obtained with a subset of the software fault targets, relative to the initial failure mode values calculated with the overall set:

$$d_g = \sum_{i=1}^{n} \overline{f}_i \cdot df_i$$

where n is the total number of failure modes considered in the dependability benchmark.

The metric used in this study is based in the user point-of-view of the system, through the use of well-known failure modes (OK, HANG, CRASH and ERRORS), as can be observed in the mathematical expression of d_g. Other metrics could be used, such as the ones related to specific mechanisms available in the target system, such as the coverage and latency of error detection mechanisms. However, the failure mode analysis is more general (i.e., it does not depends on specific features of the target system) and is more complete, as it captures the user's perception of the system.

This initial experimental study used the most frequent software fault types according to the G-SWFIT model, as shown in Table 5-4. An exception was the MLPC fault type, corresponding to the "Missing small and localized part of the algorithm", which was not considered for this

preliminary study, as those kind of faults are not related to any specific statements in the code and its correction involves non trivial modifications [Durães *et al.* 2006]. The description of the considered software fault types, as well as the corresponding coverage, concerning the most frequent types of software faults found in field [Durães *et al.* 2006], and the respective ODC classes, are also indicated in Table 5-4.

Fault type	Description	Coverage	ODC Classes
MIFS	Missing "If (cond) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (cond)" surrounding statement(s)	4.32 %	Checking
MVAE	Missing variable assignment using an expression	3.00 %	Algorithm
WLEC	Wrong logical expression used as branch condition	3.00 %	Assignment
WVAV	Wrong value assigned to a value	2.44 %	Checking
MVI M	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Assignment
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**47.50 %**	

Table 5-4 - Representativeness of the fault types considered in [Costa *et al.* 2009], according to the G-SWFIT methodology [Durães *et al.* 2006].

It is worth noting that for this small set of fault types, which represents 47.50% of the complete set of software faults and four different ODC types, a total of 781 faults were defined for the web-server benchmark and 459 faults for the quicksort system. It is relevant to mention that in some system calls, 5 in the web-server system and 2 in quicksort system experiments, it was not injected any fault, as G-SWFIT model did not indicate any code mutation on that function targets. Moreover, that system calls are the smallest, in terms of LOC, of all of the considered set (*getpid*, *getppid*, *geteuid32*, *getuid32*, *getgid32* and *getegid32*).

Considering all the G-SWFIT software faults indicated, 1,240 experiments have been executed, corresponding to the same number of injected software faults defined according to the G-SWFIT model (one single software fault injection was considered in each experiment).

Results showed that in most of the experiments (82% for the web-server system and 80% for multithreaded quicksort system) the injected software faults did not cause any failure or visible impact on the system. This means that either the fault was not activated or the correspondent error remained latent until the end of the experiment. It may also happen that these errors have been corrected or canceled by the normal execution of the program (e.g., error overwritten by a fresh value).

This initial experimental study concludes that none of the strategies provide a dramatic reduction of the number of faults if the goal is to keep a very small error in the results (e.g., less than 1%). This seems to be related to the fact that the total number of faults used to establish the reference results is relatively small (781 and 459 for each system). Nonetheless, for the web-server system, starting the fault injection experiments with the functions with greater LOCs allow achieving faster convergence to the results obtained with the complete set of faults. With this strategy, after the injection of only 51.47% of the total software faults, the induced error is about 3.8%, when comparing with the results obtained with the full set of software faults. This way, the fault injection experiments can be reduced by

almost 50%, representing an enormous saving of time in carrying out the experiments. Considering the total of time needed to inject the 781 faults in the web-server experiments, the reduction of time of the total experimentation can be estimated in, approximately, 208h. Moreover, the LOC, in machine code, is an accessible and fairly easy measure to obtain. Depending on the operating system of the target system, it can even be the easiest one, when compared to CPUt and NrCalls (however, not as simple as the random selection).

It could also be observed that for complex and large workloads such as the web-server benchmark, the number of injected faults should be around 500 or higher in order to keep the error in the results lower than 3%. Despite more experiments with other complex benchmarks/workloads are necessary to confirm this insight, this is an important practical indication for designers of future dependability benchmarks. On the other hand, smaller and simpler workloads, such as the existing in the multithreaded quicksort system, seem to allow a clearly smaller number of faults, no matter the strategy used to select the subset of faults.

The work presented in [Costa *et al.* 2009] provides a first actual contribution to solve the problem of reducing the size of the faultload, which is essential to use practical dependability benchmarks in large and complex systems. But, more important than that, this study provided an early assessment of the proposed methodology that was subsequently developed and constitutes one of the key contributions of the study presented in this thesis.

5.3.2 Proposed metodology

The proposed methodology is an extension and refinement of the aforementioned framework assessment study, presented in [Costa *et al.* 2009], and incorporates the results of a three-year research effort focused on

showing that it is possible to obtain accurate fault injection using a faultload that contains only a small fraction of all the possible software faults that can be injected in the target system. It consists of using the results obtained with a comprehensive faultload that includes all possible fault locations (i.e., total coverage faultload) to evaluate the accuracy of the results obtained with the different strategies used to reduce the size of the faultload. The experiments include the use of different target systems resulting in one of the most extensive fault injection studies ever reported.

In order to inject representative software faults, like in the validation study, the G-SWFIT fault model [Durães *et al.* 2006] was used. G-SWFIT is based on a set of operators for software fault emulation through low-level code changes in the target executable code, mimicking the most common types of real software faults. It is worth noting that these operators resulted from a field study based on the analysis and classification of more than 600 software faults found in real software applications. Table 5-5 shows the software fault types selected for inclusion in the used faultload, corresponding to the 12 most frequent types of software faults found in [Durães *et al.* 2006]. It is worth pointing out that this small set of fault types represents 50.69% of the complete set of software faults and four different ODC types (adding the MLPC fault "Missing small and localized part of the algorithm" to the set of software fault types used in the experimental validation study [Costa *et al.* 2009]).

As shown in [Durães *et al.* 2006], the long tail that characterizes the complete fault type distribution (Table 5-5 shows only the most frequent types; the tail is quite long with many fault types that are rare) makes very difficult to include more fault types in the faultload. For example, the last type of fault considered in the list shown in Table 5-5 (WPFV - Wrong variable used in parameter of function call) corresponds to 1.5% of the faults found in the mentioned field study [Durães *et al.* 2006]. That is, it is already a relatively infrequent type of software fault. For this reason, the set of fault types proposed in [Durães *et al.* 2006] has been used in many

studies in recent years and is considered as a good approximation for a difficult problem that is the definition of fault models for software faults.

The problem of trying to inject fault types that correspond to faults that are relatively rare is that even very large pieces of software may have just a few code locations (or even none) where such fault types can be injected. In other words, according to mentioned field study, nearly 50% of the software faults found in field falls in 12 types of software faults shown in Table 5-5, while the remaining 50% of the faults represent a very large number of specific types that are rather infrequent[26].

---

[26] The software fault injection technique used in this work consists in the scanning of the target code application (ready-to-run binary code) for specific low-level instruction patterns (sequence of machine code instructions) and in applying a mutation to emulate an intended software fault. Each fault type is associated to a given code pattern and a given set of preconditions that make the faults (bugs) plausible. For example, the fault type MFC (Missing Function Call) means that the programmer has forgotten to call a given function (and such bug has escaped to all the testing procedures). The field study presented in [Durães *et al.* 2006] show the typical circumstances (related to code) where such kind of fault appears in the field making it possible to reproduce such fault type, provided that the target program has the code pattern and circumstances that allow the injection of the fault.

Fault type	Description	Coverage	ODC Classes
MIFS	Missing "If (cond) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (cond)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19%	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Algorithm
WLEC	Wrong logical expression used as branch condition	3.00 %	Assignment
WVAV	Wrong value assigned to a value	2.44 %	Checking
MVI M	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Assignment
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

Table 5-5 - Representativeness of the most common software fault types used in the present methodology, according to [Durães *et al.* 2006].

As software fault injection target locations we consider the operating system kernel of the SUB, as shown in Figure 5-3. More precisely, we consider as possible targets the complete set of the OS functions, referred in the kernel symbols table. The exact identification of the fault (code target

addresses and fault type) is thus obtained applying the G-SWFIT methodology to each of the functions of that set.

Given the G-SWFIT faultload, the software faults are injected using the DBench-FI fault injector (an innovative SWIFI tool specially developed for this purpose, presented in detail in chapter 4), as its unique set of features make it especially adequate for dependability benchmarks: (i) it provides a very low intrusiveness, since it is essentially undetectable and presents no noticeable performance degradation of the FIT; (ii) it is capable of runtime fault injection (on the fly) in both user and kernel spaces and in both data and code segments; (iii) it does not require any application source code to be available; (iv) it can be dynamically loaded into a system; and (v) it can inject faults even on applications that are already running in the system when it is installed.

The used approach consists of four main phases as indicated in Figure 5-3: (i) **Pre-Injection Phase**; (ii) **Kernel Analysis Phase**; (iii) **Fault Injection Phase**; and (iv) **Strategy Analysis Phase**.

In the **Pre-Injection Phase** are identified all the locations in the OS kernel where it is possible to inject realistic software faults, according to the rules established in [Durães *et al.* 2006]. It is worth noting that, in this approach, the G-SWFIT faultload generator (software tool provided by the author of the study [Durães *et al.* 2006]) uses the exported kernel symbols table of the OS in order to detect all the possible targets in the OS kernel functions. The result of the Pre-Injection Phase is the complete set of software faults that it is possible to inject in such set of targets (i.e., the set of the OS functions).

In the **Fault Injection Phase** the exhaustive set of software faults is injected to obtain the reference results necessary for the **Strategy Analysis Phase**, in which are performed the experiments to evaluate the error observed in the results for the different hypothesis on the reduction of the number of faults to inject.

The **Strategy Analysis Phase** consists in the comparison of the results obtained when considering a subset of the initial targets with the results obtained in the initial experiments (the reference results). This final step analyzes how one can choose a subset of the targeted OS kernel functions and fault locations without hampering the initial benchmark results obtained with the total set of faults.

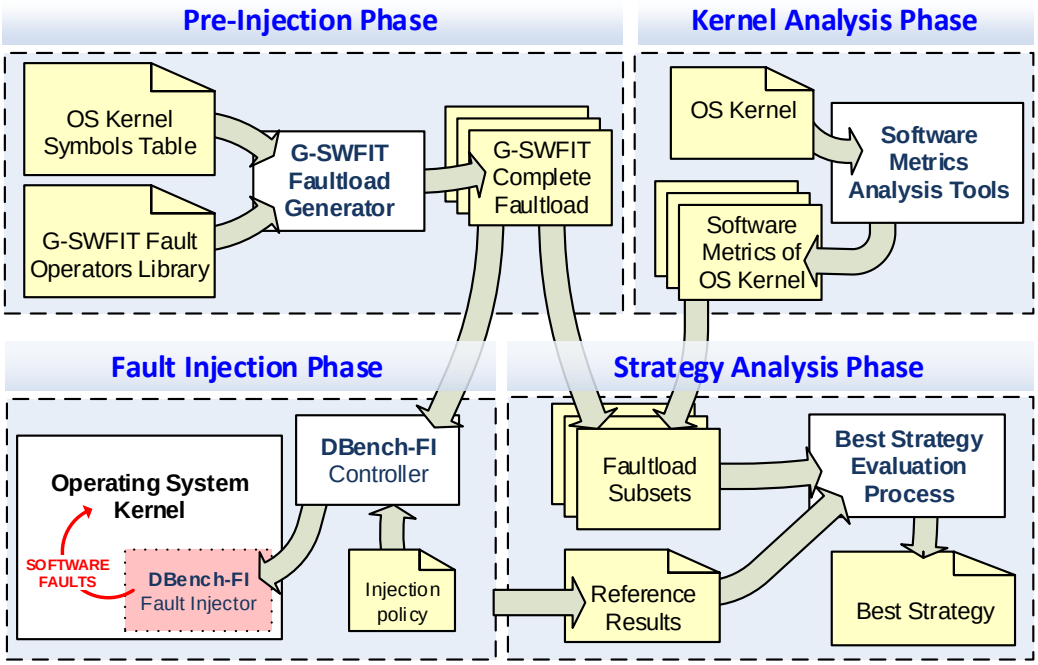


Figure 5-3 – Phases of the proposal experimental methodology.

In order to guide that faultload subset selection, some software metrics have previously been obtained from the OS kernel source code in the **Kernel Analysis Phase**. For that purpose, several characteristics and related metrics of each of the targeted kernel functions were considered:

- **Lines of code (LOC)**, which consists in the count of the lines of the software source code. It is worth noting that the analyzed LOCs were in machine code and thus some well-known restrictions generally applied to this size oriented metric, like

language and programmer dependence, no longer make sense in this context;

- **The Extended Cyclomatic Complexity (Vg**), based on the McCabe's software metric [McCabe 1976], describe the control flow complexity of each of the mentioned kernel functions. A higher Vg number corresponds to a function with greater number of execution paths and, consequently, a function harder to understand and implement;
- **Halstead's Delivered Bugs (B**), directly correlated with the complexity of code, estimates the number of errors (bugs) in the implementation. This measure is included in a broader set of measures developed by M. Halstead, to determine the quantitative measure of complexity based on operators and operands in a module [Halstead 1977];
- **Maintainability Index (Mi)**, a composite measure based on lines-of-code, McCabe's and Halstead's measures, which strives to express the relative maintainability of the code. It is worth noting that the used formula (the forms and rationale of which were developed by P. Oman [Oman *et al.* 1992]), widely accepted in the software industry, does not consider the amount of line comments, as some comments consist just of some standard blocks;
- **Functional Complexity (Fc)** is obtained by the sum of the number of input parameters, the number of return points and the Vg (Extended Cyclomatic Complexity) of each function.

Additionally, this study also considers a Random selection of software faults (RandSF), according to a uniform distribution, in order to obtain a subset of faults from the initial full set of faults initially considered.

The mentioned software metrics, with the exception of the LOC that was specifically calculated, were obtained with the RSM (Resource Standard Metrics) [RSM] and the CMT++ (Complexity Measures Tools for C/C++) Tools [CMT].

As already explained in the preliminary assessment study (section 5.3.1), the evaluation of the error incurred by the reduction of the number of faults according to each different hypothesis is based on well-known failure modes observed from the user point-of-view of the system (OK, CRASH, HANG, and ERRORS), and on the deviation relatively to the values obtained for each failure mode with the injection of the complete set of software faults (see the definition of the mathematical expression of d_g).

Chapter 6 (Experimental Evaluation of Faultload Reduction Strategies) presents and discusses the experimental results of this methodology with two real and different applications: a web-server dependability benchmark and a large-scale integer vector sort application extended with performance and quality measures. A proposal strategy for the reduction of the faultload can be found in section 6.4.

5.4 Summary

This chapter described the problem of the faultload dimension which arises from the adoption of realistic software fault models in dependability benchmarks of large and complex software systems. The execution of such benchmarks usually take several months or even years due to its large faultload size, which means that, in practice, it is not possible to execute them.

The chapter surveyed and discussed the strengths and limitations of the existing studies that address the issue of the distribution of faults among different components in target systems, and presented an experimental methodology that allows the definition of compact and

representative faultloads based on software faults. The presented methodology allows a significant decreasing on the execution time of dependability benchmarks, maintaining, simultaneously, its usefulness and representativeness. It is especially useful to open the possibility to extend dependability benchmarks to large and complex systems, where the experimentation time can significantly be reduced, making the benchmarks feasible and useful in such class of systems.

Chapter 6

Experimental Evaluation of Faultload Reduction Strategies

This chapter describes the experimental setup used to evaluate the different strategies for the reduction of the number of software fault injection experiments (presented in section 5.3 - Experimental framework) with two real and different applications: a web-server dependability benchmark and a large-scale integer vector sort application extended with performance and quality measures. It presents and analyzes the results of more than two years of comprehensive fault injection experiments, encompassing more than 41 thousand software faults, and proposes a strategy to choose adequate fault injection targets without restricting the benchmark scope and keeping accurate dependability benchmark results.

The proposed strategy will open the possibility to extend the use of dependability benchmarks to large and complex systems, which otherwise would be, in practice, impossible to run due to its large faultload size (such benchmarks usually take several months or even years to execute).

6.1 Introduction

More than 41 thousand continuous fault injection experiments have been carried out in more than two years[27], in order to evaluate different strategies to guide the fault injection target selection and to reduce the number of software fault injection experiments for a dependability benchmark, or for an experimental evaluation using software fault injection. The main goal is to find a strategy to reduce the fault injection target set and thus decrease the execution time of the dependability benchmark experiments, while maintaining the dependability benchmark usefulness and representativeness. This method will open the possibility to extend the dependability benchmarks to large and complex systems, making them feasible and practicably applied (such benchmarks usually take several months or even years due to its large faultload size).

---

[27] It is worth noting that the time needed to complete each fault injection experiment largely depends on the chosen BT system. For large and complex systems, such as the web-server benchmark used in this study, the injection of each software fault takes about 20 minutes (average value), as showed later in this chapter.

6.2 Experimental setup

The experimental setup used in this work is composed of two systems:

- A server machine (Intel Pentium IV 2.66GHz, 512MB), which corresponds to the SUB, including the BT and the DBench-FI fault injector;
- A client machine (Intel Pentium IV 2.0GHz, 512MB), which corresponds to the BMS, running the benchmark client.

Both machines are connected via a 100Mbps Ethernet connection and run the Linux RedHat operating system (kernel 2.4.18-3).

The dependability benchmark used is a web-server benchmark (WS) based on the SPECweb99 industry standard performance benchmark for web-servers [SPEC], extended with faultload and dependability measures (failure modes). In the specific setup used in the experiments, the Apache web-server was used.

In order to evaluate the different strategies to reduce the number of injected faults, a second workload was used, running in the same environment as the dependability benchmark mentioned above. This second workload (quite different from the WS benchmark, in terms of required computer resources) consists of a client-server application to sort large-scale integer vectors, based on a Multithreaded Quicksort algorithm (MtQs), extended with performance and quality metrics.

The experiments were chosen and designed to show that, even considering two totally different applications, it is possible to consider a subset of all the possible fault injection targets maintaining, at the same time, the usefulness of the benchmark results. This method will open the possibility to extend the dependability benchmarks to large and complex systems, making them feasible and practicably applied. As mentioned in previous chapters, that is exactly one of the main goals of dependability

benchmarks: to offer practicable and efficient methods, considering the computing effort, the number of experiments and the time to run the benchmark, in order to analyze a set of measures and charactering a system.

The web-server dependability benchmark is a very realistic benchmarking scenario already used as a case study in [Durães *et al.* 2004a]. In those experiments, the BT consists of the Apache web-server and the BC is the SPECWEB99.

The used SPECweb99 performance benchmark can be briefly described by its components:

- *Benchmark setup* – SPECweb99 uses a previously defined number of clients in order to submit requests to the web-server under evaluation. One of those clients, known as the prime client, coordinates all the actions of all the others. In these experiments, all the clients run in the same machine (the BMS) and are referred as the Benchmark Client in Figure 5-1 – Experimental Architecture. In fact, running in physical different machines or operating systems is not really a requirement of SPECweb99.
- *Workload* – the workload used by SPECweb99 and submitted to the server is representative of the most common web-server operations and is composed of typical POST and GET requests, including both static and dynamic operation types [SPEC]. The defined workload also emulates common actions such online registration requests and advertising services.
- *SPECweb performance measures* – the measures are obtained through the SPECweb prime client and for this specific work the following were considered relevant: (A) *SPEC*, the main SPECweb99 metric, measures the number of simultaneous connections that a server can support. Known as conforming connections, they are defined as those that have an average bit

rate of at least 320kbps and less than 1% of reported errors; (B) Throughput (Thr), considered as the number of operations (e.g., POSTs and GETs) per second; (C) Operation Count Errors (Err), considered as the number of errors found by the client in the requested operations.

- *SPECweb99 benchmark rules* – this performance benchmark require very specific rules for experiment conduction in order to the acceptance of the final reporting results by the SPEC organization. Concerning those rules, we recommend the reading of [SPEC] for more detailed information. In the conducted experiments, in order to reduce their total time, not all of those impositions were accomplished, as the respective final benchmark reporting results is completely out of the scope of this paper. Specifically, in this benchmark there were requested 40 simultaneous connections to the server, using three batches or iterations of 300 seconds each and a *Warmup Time*, *Rampup Time* and *Rampdown Time*[28] of 30

[28] *Warmup*, *Rampup* and *Rampdown* times are changeable SPECWeb99 benchmark parameters. The *Warmup time* is the time, in seconds, intended to warm up any caches before taking any measurement. The *Rampup time* is the warmup time, in seconds, before the 2nd and following iterations of the test. The Rampdown time represents the time, at the end of each iteration, required for the end of SPECWeb99 workload.

seconds each. These parameters represent SPECweb99 benchmark constants, as defined in [SPEC].

Each web-server experiment consists of running the workload on the Apache web-server and on the injection of one software fault few seconds after the experiments start (see Figure 6-1 - Web-server benchmark execution profile.). In this way, the software fault is injected after the web-server reaches the Steady State Condition[29] (the *warmup time* was set to 30 seconds). The DBench-FI fault injector, as mentioned, takes the faultload and injects each software fault directly in the code of the running target - a predefined function located in the OS kernel. It is worth noting that, concerning the HANG failure mode, the BMS defines for this group of experiments a maximum of 30 minutes each. This maximum time is sufficient considering that the normal execution time to complete each experiment of three iterations is about 20 minutes (as referred later in the section 6.3 - Results and discussion). After that time is elapsed, the SUB is considered hanged and is remotely restarted by the BMS, via software or hardware.

---

[29] The system achieves the Steady State Condition after a given *warmup time*. This state guarantees that the system is able to maintain its maximum transaction processing throughput.

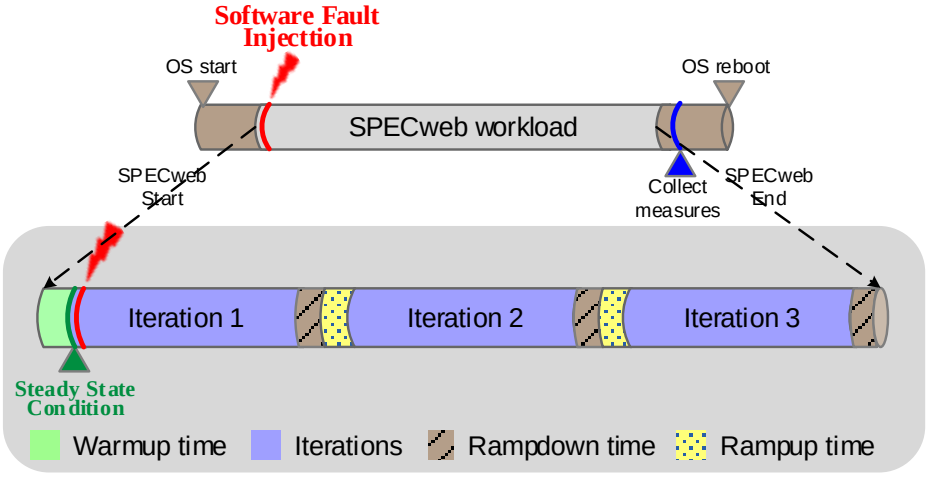


Figure 6-1 - Web-server benchmark execution profile.

After each experiment run, the BMS gather the measures related to performance degradation, mainly given through SPECweb99 performance benchmark, as well as some other metrics related to dependability, namely, the information about the resulting failure mode. In addition to the mentioned performance metrics, the total time to complete each one of the experiments (ExpT) it is also collected by the BMS.

Relatively to the Multithreaded Quicksort application it is important to note that it mainly serves as a control application and as a good comparison system, as it has some completely different requirements concerning to computer resources. It consists of an application responsible for the sort of a 10,000,000 integer randomly generated vector and a client that requests the resulting sorted file. In those experiments, the BT consists of the Multithreaded Quicksort application and the Benchmark Client is the application client that asks for that ordered vector file. Each of those experiments consists in generating the 10,000,000 integer random vector, executing the Multithreaded Quicksort on that vector and, finally, writing the resulting vector to a file that will be read and tested by the client that has made the request. In each experiment (see Figure 6-2 - Multithreaded quicksort benchmark execution profile.), one single software fault is

injected when the application starts. Thus, the software fault is injected at the start of the random vector generation phase, just before the execution of the sorting algorithm.

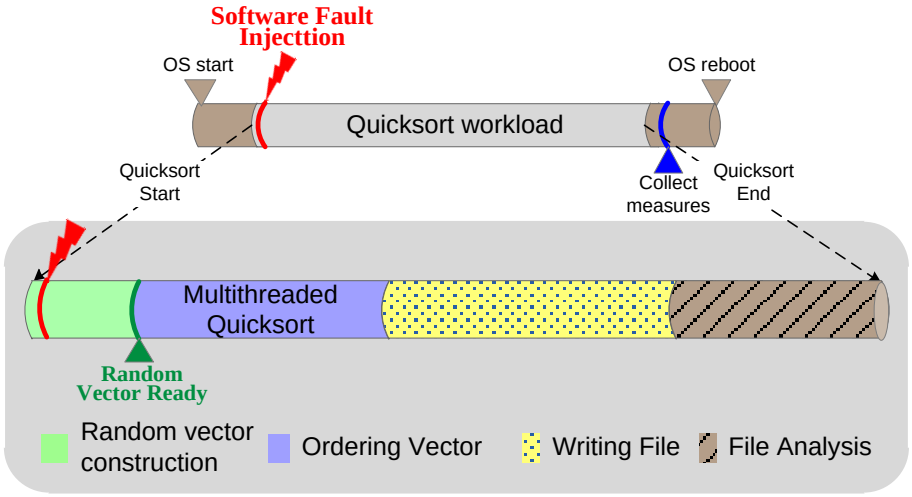


Figure 6-2 - Multithreaded quicksort benchmark execution profile.

Like the WS experiments, the DBench-FI fault injector takes the faultload and injects each software fault directly in the OS kernel code of the SUB, on top of which is running the sorting algorithm. For this group of fault injection tests, the BMS defines a maximum time of 15 minutes for each experiment to run. This maximum time is sufficient considering the normal execution time to complete each experiment (as mentioned later in section 6.3 - Results and discussion). Similarly to what we mentioned for the WS experiments, after that maximum time is elapsed the SUB is considered hanged and is remotely restarted by the BMS, via software or hardware.

After each run of this MtQs experiments, the BMS provides measures related to performance degradation, based on the execution time to complete all the process of generating, sorting and writing the vector (ExpT), as well as some other metrics related to dependability, namely the information about the resulting failure mode. For this purpose we consider

that an error exists (Err) if the vector is not correctly ordered. In that case, a metric based in the number of wrong placed integers in the vector is calculated as an indicator of the quality of the obtained result.

All the executed experiments, either related to the WS or to the MtQs, required no human intervention as their execution were completely automated through the use of a set of appropriate tools incorporated in the BMS.

6.3 Results and discussion

For each kind of experiments, concerning both the WS and the MtQs benchmark experiments, some previous performance tests, 100 for each of the following types, were made in order to obtain a measure of the intrusiveness of the DBench-FI fault injector in the benchmark systems:

- Without DBench-FI fault injector. That is, without the respective fault injector module inserted in the OS kernel;
- With the DBench-FI fault injector in profile mode. That is, using software fault injection but without really changing any target.

The Table 6-1 shows the average values for every performance measure considered in each type of experiment.

The comparison of these performance results and the degradation value obtained give us a measure of the DBench-FI fault injector overhead and intrusiveness in all of the experiments presented throughout this section. As can be observed, it is not detected any intrusiveness or performance degradation imposed by the used fault injector. Moreover, as no errors were observed in any of those experiments, we can conclude that the intrusion factor of the fault injector either in the WS or in the MtQs calculations is non-existent.

WS Experiments	Without DBench-FI	With DBench-FI	Intrusion Factor
SPEC (#)	40	40	0
Thr (Ops/sec)	126.5	126.5	0
ErrR (# Ops)	0.0	0.0	0.0
ExpT (hh:mm:ss)	00:19:00	00:19:00	-

MtQs Experiments	Without DBench-FI	With DBench-FI	Intrusion Factor
ExpT (hh:mm:ss)	00:00:26	00:00:26	-
Err (#)	0.0	0.0	0.0

Table 6-1 – Average performance results (no faults injected).

As some strategies to reduce the number of injected faults rely on the characteristics of the kernel functions, the entire OS kernel functions were analyzed and the related metrics obtained - **Kernel Analysis Phase** (Figure 5-3 - Phases of the proposal experimental methodology). Recall that the considered metrics are, as defined in section 5.3 - Experimental framework: Lines of code (LOC), Extended Cyclomatic Complexity (Vg), Halstead's Delivered Bugs (B), Maintainability Index (Mi), and Functional Complexity (Fc).

Considering all the G-SWFIT software faults indicated in Table 5-5 (Representativeness of the most common software fault types used in the present methodology, according to [Durães *et al.* 2006]), 41,750 fault injection experiments have been executed in 1,153 kernel functions - 20,875 for each type of workload (WS and MtQs). These faults corresponds to the total number of software faults that can be injected in the code of the OS kernel (considering the entire exported kernel symbols table), according to the rules proposed in [Durães *et al.* 2006] for the realistic emulation of software faults. It is worth mentioning that in some very small OS functions (with very few lines of assembly code) referred by

the exported kernel symbol table it was not injected any fault, as the G-SWFIT fault model did not indicate any suitable code locations on those target functions. It is also important to note that for 21 targeted functions, originally programmed in assembly language, only the ASM LOC measure was collected. This is due to restrictions of both of the used tools to extract the software metrics [RSM], [CMT].

Concerning the performance in the presence of injected faults, the final experimental results obtained are shown in Table 6-2.

	WS Experiments*				**MtQs Experiments**	
	SPEC (#)	**Thr** (Ops/sec)	**ErrR** (# Ops)	**ExpT** (hh:mm:ss)	**Err** (#)	**ExpT** (hh:mm:ss)
Min	0.00	2.2	0.0	00:00:43	0.0	00:00:05
Max	40.00	171.2	65,239.3	00:30:07	9,990,571.0	00:15:00
Avr	37.8	126.5	23.7	00:19:15	3,172.7	00:00:40
StdDev	9.1	3.6	865.6	00:03:10	174,255.7	00:01:46

** Experimental results considering the average value of all the 3 SPECweb iterations.*

Table 6-2 – Performance results in the presence of faults.

Concerning the WS experiments, as result of the mentioned execution profile, there were observed that some injected software faults caused several non-conforming connections (SPEC) and/or some lower values of throughput (Thr) and also several error operations (Err) detected by the SPECweb99 benchmark. It is worth noting that for this type of experiments, the presented values are the average of all iterations executed (3 SPECweb iterations in each one of the 20,875 experiments). The resulting charts are shown in Figures 6-3 to 6-6.

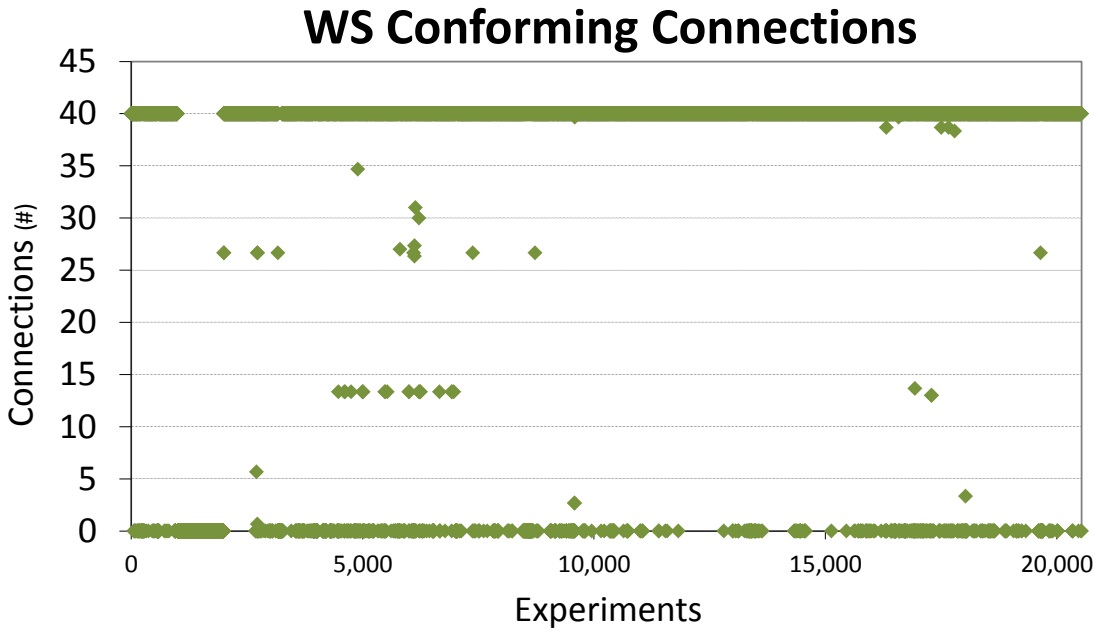


Figure 6-3 - WS Experimental results: Conforming connections.

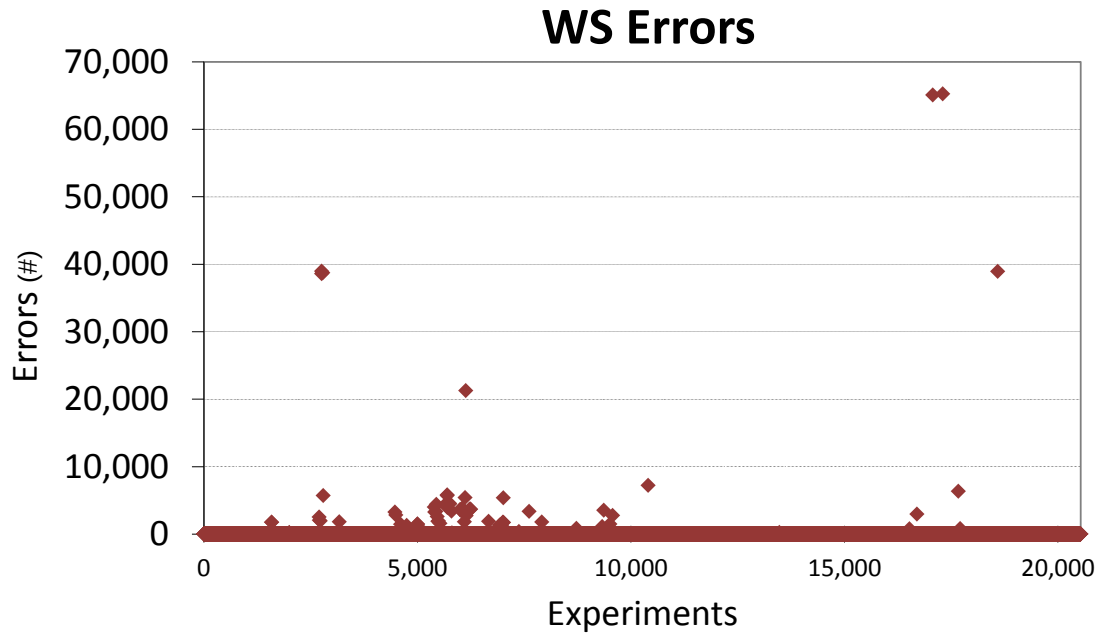


Figure 6-4 - WS Experimental results: Errors.

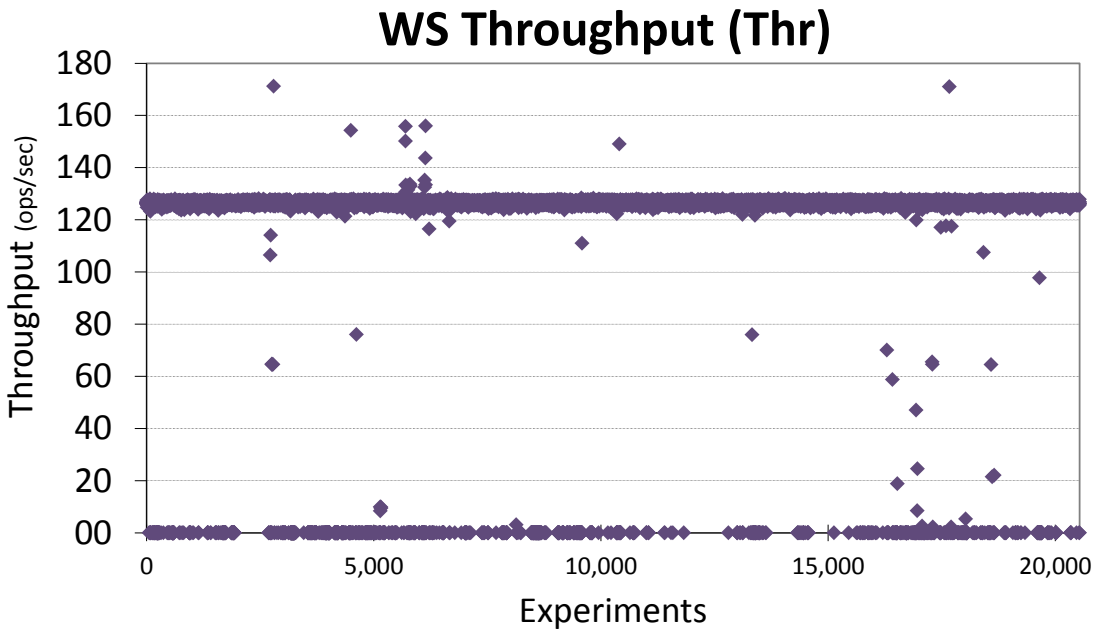


Figure 6-5 - WS Experimental results: Throughput.

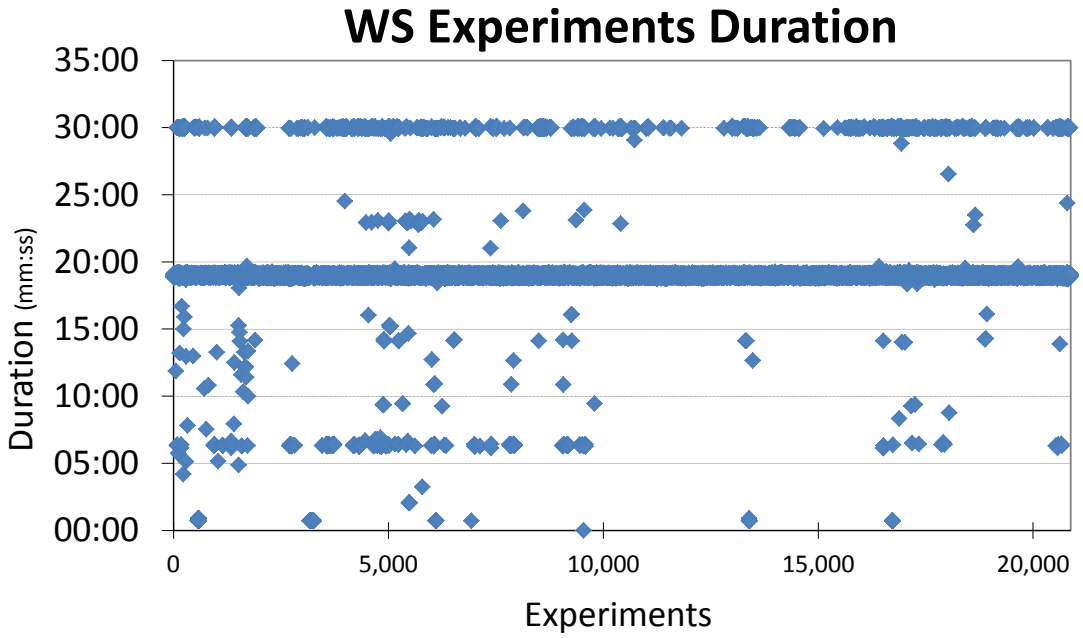


Figure 6-6 - WS Experimental results: Experiments duration.

With respect to the MtQs experimental results (see Figure 6-7 and Figure 6-8), some experiences have also led to errors in the results. Such situations occurred when either the result vector was completely unavailable by the client (considering that there is no HANG or CRASH of the BT) or the result corresponds to an existent but incorrect ordered vector. It is worth noting that this last case was only observed in 16 experiments. This means that, in most of the times, when the resulted sorted vector was written to disk, no errors had been detected by the client in the ordered integer vector. This is explained by the specific characteristics of the MtQs application, namely, by the file based result to the client's request. We also observed that, like in the WS, some experiments present different execution times due to the induced kernel code disturbance.

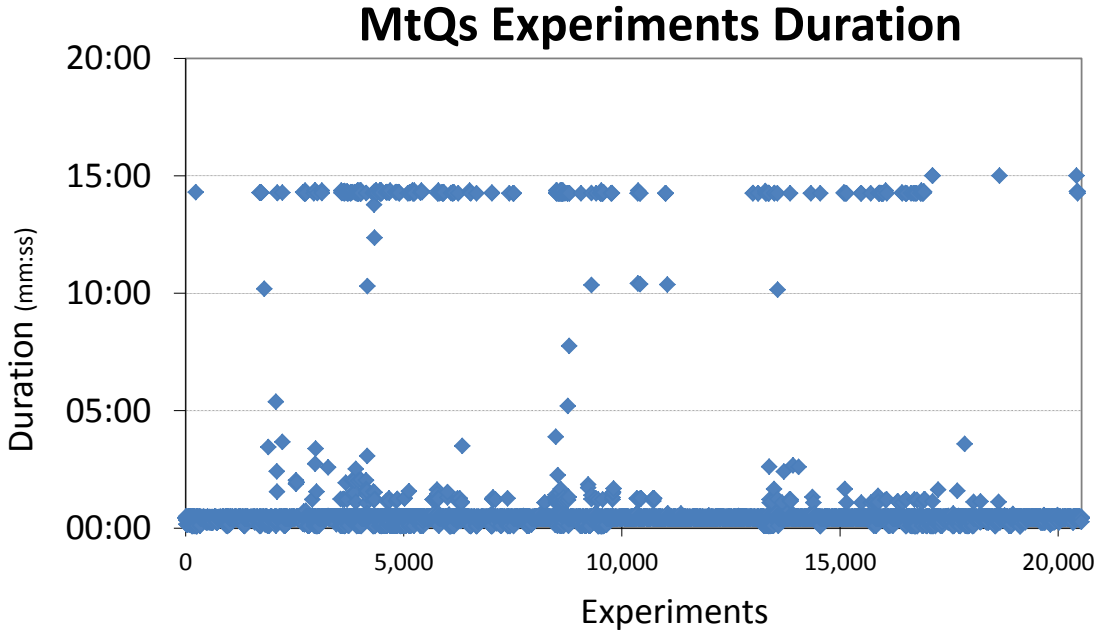


Figure 6-7 - MtQs Experimental results: Experiments duration.

Regarding the mentioned failure modes, it can be observed that, in most of the experiments (92.50% for the WS and 93.32% for the MtQs), the

software faults injected in the OS kernel did not cause any failure or visible impact on the application (see Figure 6-9 - Failure modes of WS experiments, and Figure 6-10 - Failure modes of MtQs experiments). It is worth noting that these results are consistent with the results of fault injection experiments reported in the literature. Moreover, this rarefaction (i.e., only a few faults cause a visible impact on the BT) is one of the reasons for the difficulty in reducing the faultload size.

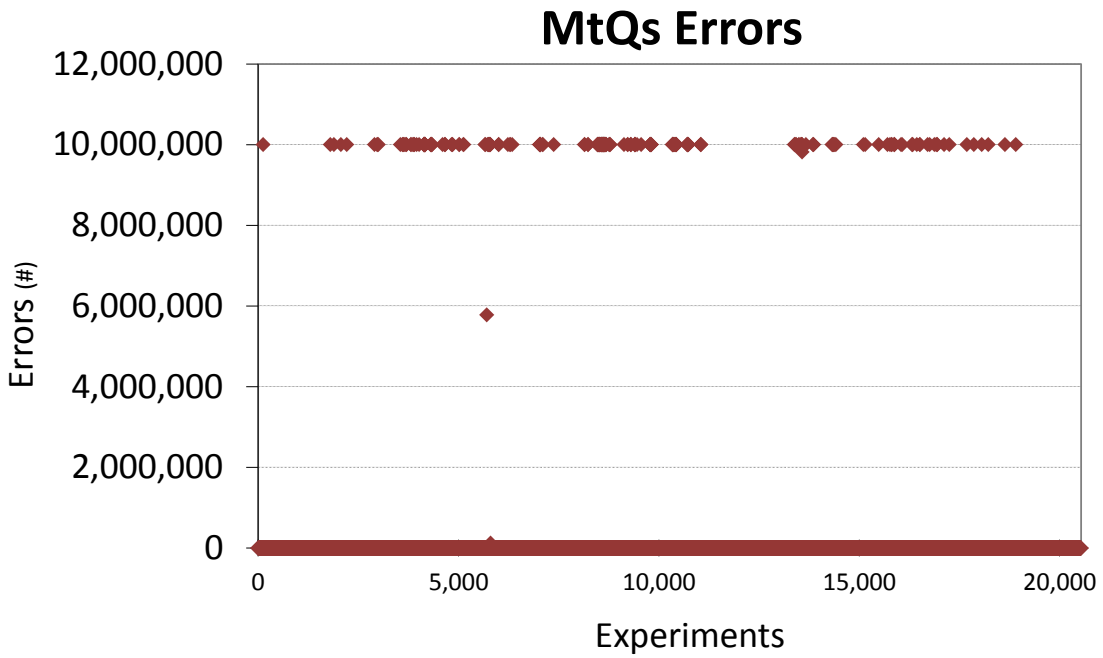


Figure 6-8 - WS Experimental results: Errors.

Many factors may contribute to this behavior. Since we need the failure modes obtained with the total set of faults (i.e., all the possible faults that G-SWFIT can inject in the Linux kernel) to be used as a reference result for the evaluation of the different strategies proposed to reduce the number of faults to be considered in the faultload, we consider the whole Linux kernel code (memory management, scheduler, file system, I/O management, etc.) as the fault injection target. This suggests that many faults have not been activated, which *per se* explains a large fraction of the

faults (injected in the OS) that caused no visible effects on the applications. Other factors such as errors that remained latent until the end of the experiment or have been corrected or canceled by the normal execution of the program (e.g., error overwritten by a fresh value) are plausible causes as well. Obviously, even when the injected fault damages the OS, it may happen that the components affected by the fault had no effect on the applications (WS and MtQs).

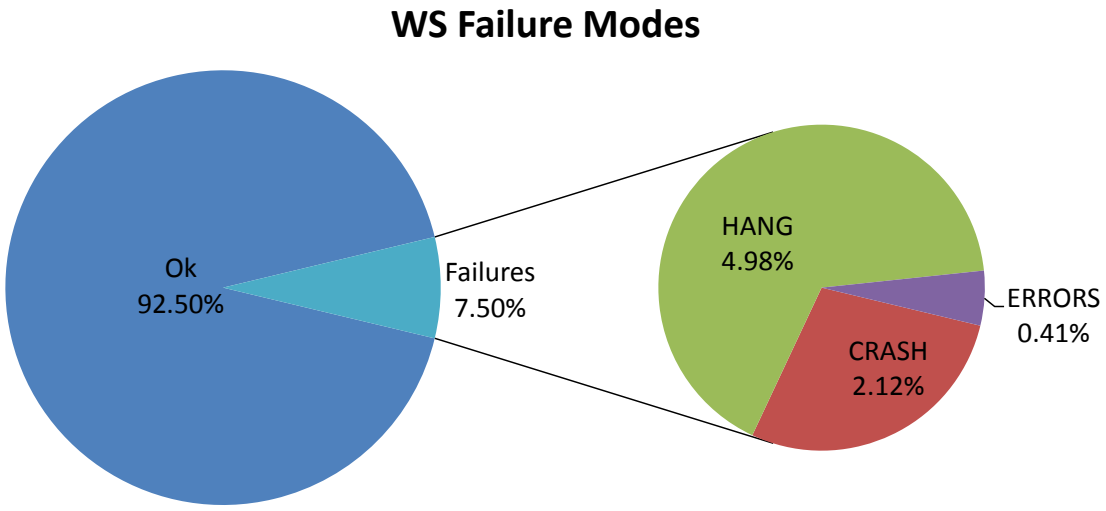


Figure 6-9 – Failure modes of WS experiments.

It is worth noting that we excluded the use of well-known techniques such as monitoring (to detect when the fault is activated) or code profiling (to previously identify the OS code areas that are used more intensively by the application) because the goals of our research require reference results obtained by a non-intrusive faultload that include all the possible faults.

The similarity between the values obtained for the OK failure mode for both systems, despite the great difference between their computational characteristics, suggests a similarity behavior of the systems in the presence of a faulty OS (considering the occurrence of problems), independently of the used applications (BT).

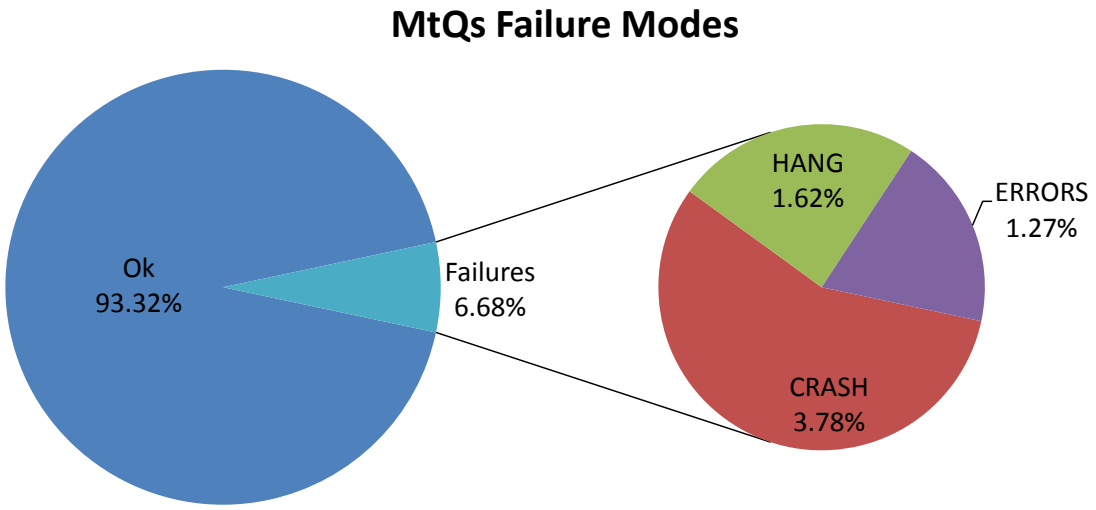


Figure 6-10 – Failure modes of MtQs experiments.

Concerning the strategy analysis phase of the approach and in order to choose a subset of software fault targets, and, consequently, decrease the number of injected faults and the resulting total experimentation time, without restricting the benchmark usefulness, we analyzed, as mentioned, 6 different approaches: LOC, Vg, B, Mi, and Fc metrics, and RandSF, a random selection of software faults, following a uniform distribution. Still for the random selection, it is important to notice that for each group of randomly chosen OS kernel functions (from 1 to 1,153), there were executed 2,000 experiments, and analyzed the resulting maximum, minimum, average and standard deviation values. That is, 2,000 experiments for each one of the combinations of n functions among 1,153, for n between 1 and 1,153 (in other words, 2,000 experiments of 1 randomly chosen function among the 1,153 target functions; 2,000 experiments of 2 randomly chosen functions among the 1,153 target functions; etc.). This selection method mainly serves as a control strategy.

For each one of the 6 mentioned approaches, Table 6-3 shows the percentage of total fault injections needed to obtain a given global

deviation error limit in the WS experiments, considering the full set of targets (the two best strategies of each global deviation error limit are presented in a shaded background). Correspondent data for the MtQs experiments can be found in Table 6-4. These tables shows, for each one of approaches based on software metrics (LOC, Vg, B, Mi and Fc), two different ways of choosing the kernel target functions: based on the ascending (Asc) and descending (Desc) orders of the correspondent metric. For example, for the LOC approach, it is possible to start the software fault injection in functions with greater LOC values (LOC approach in descending order – LOC Desc) or in functions with smaller LOC values (LOC approach in ascending order - LOC Asc). Thus, we consider 11 different strategies to choose the adequate fault injection targets: two sort orders for each one of the 5 software metrics based approaches plus a random approach, as explained above. The presented values show that, for example, for the WS experiments, using LOC Asc (i.e., choosing as injection order the OS functions with smallest LOC), it is necessary to inject 29.56% of the faults (6,170 faults) to achieve a global deviation error in the failure modes less than or equal to 2%.

It is very important to note that the values indicated in Tables 6-3 and 6-4 represents the worst-case scenarios. That is, possible smaller sets of faults that incidentally could produce results with a smaller error are not being considered. Instead, it is found the worst combination of faults (i.e., the largest set) needed to assure a given error limit. In other words, any form of casuistic occurrence along the experiments is eliminated, by ensuring that the indicated values are such that, for each approach, none of the remaining experiments inflict a higher global deviation value. I.e., being d_{g_i} a global deviation value for a given number of faults i,

$$d_{g_i} = min\left\{d_{g_j}, \forall j \leq i : d_{g_j} > d_{g_k}, \forall k > j\right\}$$
$$i, j, k \in \{injected\ faults\}.$$

		WS Experiments					
		Error (d_g - global deviation values)					
		0	**<=0.5%**	**<=1%**	**<=2%**	**<=3%**	**<=4%**
LOC	Asc	100.0%	**53.84%**	45.03%	29.56%	19.29%	12.19%
		20,875	**11,239**	9,399	6,170	4,027	2,544
	Desc	100.0%	90.17%	**21.95%**	**14.81%**	12.59%	9.00%
		20,875	18,822	**4,582**	**3,091**	2,628	1,879
Vg	Asc	100.0%	**59.16%**	44.73%	28.42%	19.41%	16.44%
		20,875	**12,349**	9,338	5,933	4,051	3,431
	Desc	100.0%	87.39%	**25.05%**	**18.72%**	**1.61%**	**0.80%**
		20,875	18,242	**5,230**	**3,908**	**337**	**168**
B	Asc	100.0%	62.65%	41.82%	33.94%	19.40%	12.16%
		20,875	13,078	8,730	7,085	4,049	2,538
	Desc	100.0%	90.10%	70.16%	10.09%	**1.61%**	**1.61%**
		20,875	18,809	14,646	2,106	**337**	**337**
Mi	Asc	100.0%	91.32%	32.92%	16.74%	10.87%	9.77%
		20,875	19,063	6,872	3,494	2,269	2,040
	Desc	100.0%	69.97%	39.73%	24.47%	17.30%	13.15%
		20,875	14,607	8,293	5,109	3,611	2,746
Fc	Asc	100.0%	63.99%	53.69%	40.74%	29.71%	25.71%
		20,875	13,357	11,207	8,504	6,202	5,366
	Desc	100.0%	82.63%	76.77%	10.05%	5.01%	1.62%
		20,875	17,250	16,025	2,097	1,046	338
RandSF	Max	100.0%	97.05%	85.93%	58.23%	41.89%	26.35%
		20,875	20,260	17,938	12,155	8,745	5,500

Table 6-3 - Percentage of fault injections needed to achieve a given global deviation error limit in the WS Experiments.

		MtQs Experiments					
		Error (d_g - global deviation values)					
		0	**<=0.5%**	**<=1%**	**<=2%**	**<=3%**	**<=4%**
LOC	Asc	100.0%	61.55%	47.69%	33.49%	22.62%	18.18%
		20,875	12,848	9,955	6,992	4,721	3,796
	Desc	100.0%	93.34%	70.84%	13.04%	7.87%	1.68%
		20,875	19,485	14,788	2,722	1,643	351
Vg	Asc	100.0%	63.31%	50.69%	30.29%	21.69%	17.92%
		20,875	13,215	10,581	6,322	4,528	3,741
	Desc	100.0%	87.39%	43.67%	**3.91%**	**3.03%**	**0.80%**
		20,875	18,242	9,116	**816**	**632**	**168**
B	Asc	100.0%	67.31%	46.36%	34.73%	21.56%	17.85%
		20,875	14,050	9,678	7,249	4,501	3,727
	Desc	100.0%	90.81%	70.92%	**6.06%**	**0.80%**	**0.80%**
		20,875	18,957	14,805	**1,264**	**168**	**168**
Mi	Asc	100.0%	91.75%	**14.79%**	10.59%	9.77%	1.46%
		20,875	19,152	**3,088**	2,211	2,040	304
	Desc	100.0%	**43.95%**	39.51%	26.12%	19.33%	16.62%
		20,875	**9,174**	8,248	5,452	4,036	3.469
Fc	Asc	100.0%	**53.28%**	47.30%	28.27%	17.61%	2.32%
		20,875	**11,122**	9,873	5,901	3,677	485
	Desc	100.0%	82.34%	**14.51%**	9.84%	1.62%	1.62%
		20,875	17,189	**3,028**	2,055	338	338
RandSF	Max	100.0%	95.95%	81.35%	61.79%	38.70%	22.75%
		20,875	20,030	16,982	12,900	8,078	4,748

Table 6-4 - Percentage of fault injections needed to achieve a given global deviation error limit in the MtQs Experiments.

In other words, for a fixed sort strategy, the mathematical expression of d_{g_i} assures that for a certain subset i of injected faults, no other subset of software faults that includes i produces a greater global deviation (in the limit, it would be possible that, luckily, the injection of faults in one single function, the first in a certain sort strategy, could produce a null deviation). Analogous definitions hold for individual failure modes.

Looking at the reduction of the number of faults, in both relative (percentage) and absolute terms, the following observations can be drawn based on the obtained results:

- Some of the strategies provide a good reduction of the number of faults (lower than 50%), keeping the error in the results very small (e.g., less than or equal to 1%).
- Smaller and simpler workloads, such as MtQs, seem to allow identical number to what would be necessary for more complex workloads. Particularly, it can be observed that, in order to obtain smaller errors in the results (less than or equal to 0.5%), the number of injected faults is identical to what would be needed in complex and large workloads.
- Concerning the WS experiments (that represent relatively large and complex workloads), the best strategies to select a subset of faults to inject, for errors between 3% and 4%, are Vg Desc and B Desc. However, we can state that the approach Fc Desc is very close to those ones, also showing a good convergence for that error range. For that kind of workloads, and for errors in the range between 1% and 2%, we can mention LOC Desc ad Vg Desc as the two best strategies. For small errors (lower than or equal to 0.5%), the LOC Asc and the Vg Asc showed to be the best strategies (with 53.84% and 59.16%, respectively, of the total injected faults).

- Concerning smaller and simpler systems, represented by the MtQs experiments, the best strategy to decrease the number of faults, for errors between 2% and 4% are B Desc and Vg Desc, closely followed by the Fc Desc. However if we consider errors in the range from 1% to 2%, we can mention the Fc Desc and the Mi Asc as the best strategies. For smaller errors, lower than or equal to 0.5%, the Mi Desc and the Fc Asc revealed to be the best ones.
- In general, we can state that the best strategies for errors in the range 1% to 4% are not the best ones for smaller errors (lower than or equal to 0.5%), and vice-versa. More precisely, both WS and MtQs experiments seem to show that, regardless of the strategy used, the Asc order is the best one for very low errors (lower than or equal to 0.5%). On the other hand, we can state that the Desc orders are the best ones for errors between 1% and 4%. This situation can be explained by the function-based choice used in this study. More complex functions, those with higher software complexity measures, and potentially best represented in faultload (which includes all the possible software fault locations), induces a one-step block analysis of a greater set of software fault injections. On the contrary, less complex functions (typically represented in the faultload only by a few software faults) induce a more fine and step-by-step analysis. Consequently, the former type of functions provides a faster, but rough, convergence, in opposition to the latter type, with a slower, but accurate, convergence. The criteria Mi Desc is an exception to this rule, and will be explained later.
- The random selection (RandSF) of a subset of faults is the worst strategy of all to reduce the number of software fault injections. It is worth noting that this selection strategy is, by far, the easiest fault reduction strategy concerning implementation, as all others

require some sort of previous target analysis, which is obviously not the case of the RandSF strategy.

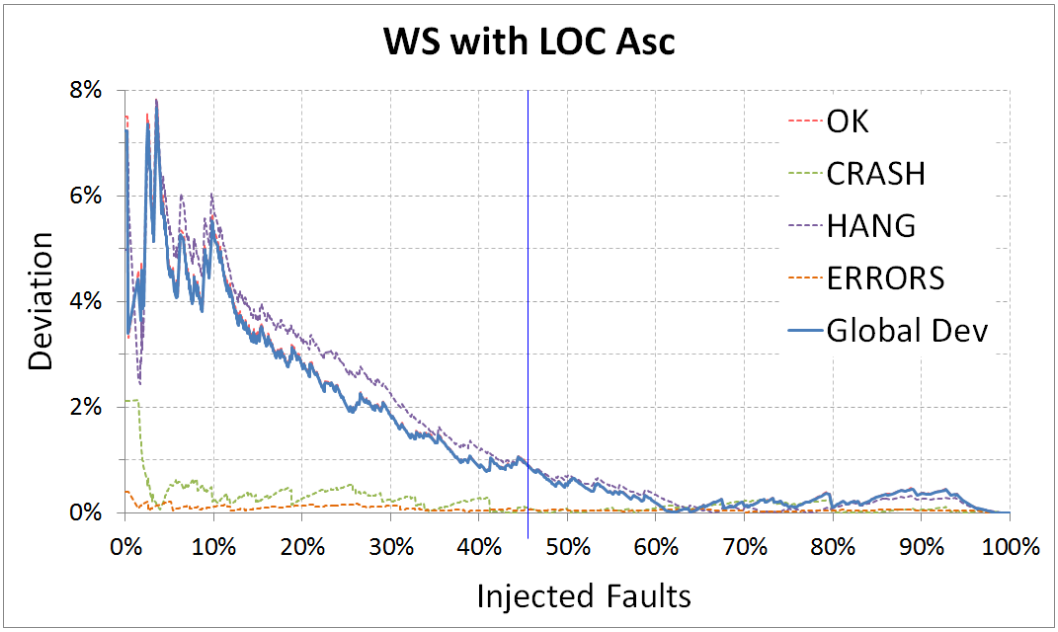
In addition to the analysis provided by the Tables 6-3 and 6-4, it is also important to analyze the error evolution in a less discrete way. The following charts (from Figure 6-11 to Figure 6-15, for the WS; and from Figure 6-16 to Figure 6-20, for the MtQs) show the error evolution df_i for each failure mode (represented by its name) and the global error d_g (represented as Global Dev), as well as their relationship. Each individual chart represents each strategy for the definition of subsets of faults. The vertical blue line indicates the percentage of injected faults needed to achieve 1% of error (global deviation) in each of the considered strategies (that value seems to be a turning value, as explained below). It is important to notice that, as described for the table values analysis, one should not consider smaller incidental errors produced by smaller sets of faults. On the other hand, besides the global deviation, it is also important to analyze the individual deviation values for each one of the failure modes considered (OK, CRASH, HANG and ERRORS), as, depending on the SUB characteristics and on some specificities of the target system, a certain failure mode can be more important and relevant than others.

Regarding the LOC approach in the WS experiments (from Figure 6-11 to Figure 6-15), for example, we can observe in the charts of Figure 6-11 that if the experiments were made starting with the functions with greater LOC values (descending order - LOC Desc), from a certain order, the global deviation value remains near zero and with minor changes. The same behavior is noticed in the LOC Asc (in ascending order). However, the convergence in LOC Desc is clearly quicker: 20% of the total injected faults induce a global error near 1.3% (it is worth to recall that we are using the worst-case scenarios). As already mentioned, this observation is explained by the function-based analysis used in this study. Despite the differences in the convergence speed, an analogous behavior holds for all the other failure mode deviations. This behavior was not so evident in the

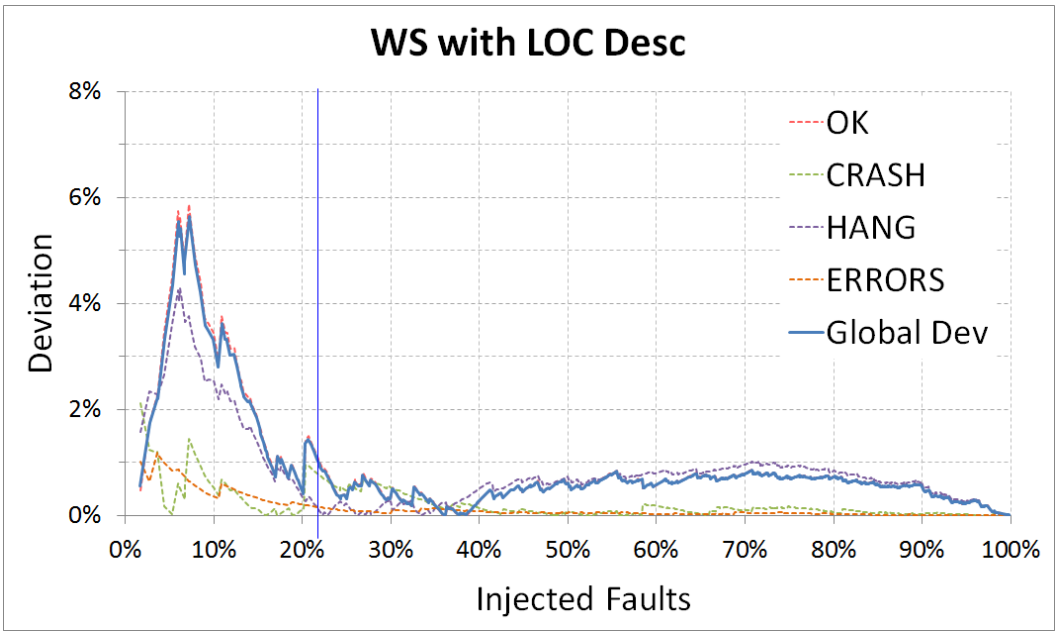
discrete values presented in previous tables. The charts also show that, considering the LOC Asc approach, injecting about 45% of the total software fault injection considered, we can obtain a global deviation of about 1%. Moreover, that value remains with minor changes in the experiments immediately following and converges to zero as we inject the remaining software faults.

One can also observe that, regardless of the metrics used to select the targets, the convergence lines of the Asc approaches present similar behaviors and the same holds for Desc orders (except for the Mi metric, as explained below). This similarity seems to induce the definition of two groups: one for each sorting option, Asc and Desc. In fact, the charts seem to confirm that, regardless of the strategy used, the Asc order is the best for very low errors (lower than or equal to 0.5%). We can state that the Desc orders are the best ones for errors between 1% and 4%. An exception to this rule is related to the Mi approach, in which an exchange of the charts can be observed. This variation is justified by the definition of the Maintainability index, Mi, which, as developed by P. Oman [Oman *et al.* 1992], is greater for smaller and less complex functions, in opposition to all the other metrics. This observation seems to indicate 1% as a turning value, where the Desc strategies start to be less efficient then the Asc ones (reversed for the Mi strategy).

Despite the referred similarity of the presented charts, a more detailed look shows that the Vg, B, and Fc approaches, in Desc order, reveal a higher convergence of global deviation values up to about 2%. On the other hand, for very low deviation values, in the order of magnitude of 0.5%, the LOC, Vg and B approaches, in Asc order, show a greater efficiency. This confirms our observations from Table 6-3 - Percentage of fault injections needed to achieve a given global deviation error limit in the WS Experiments.

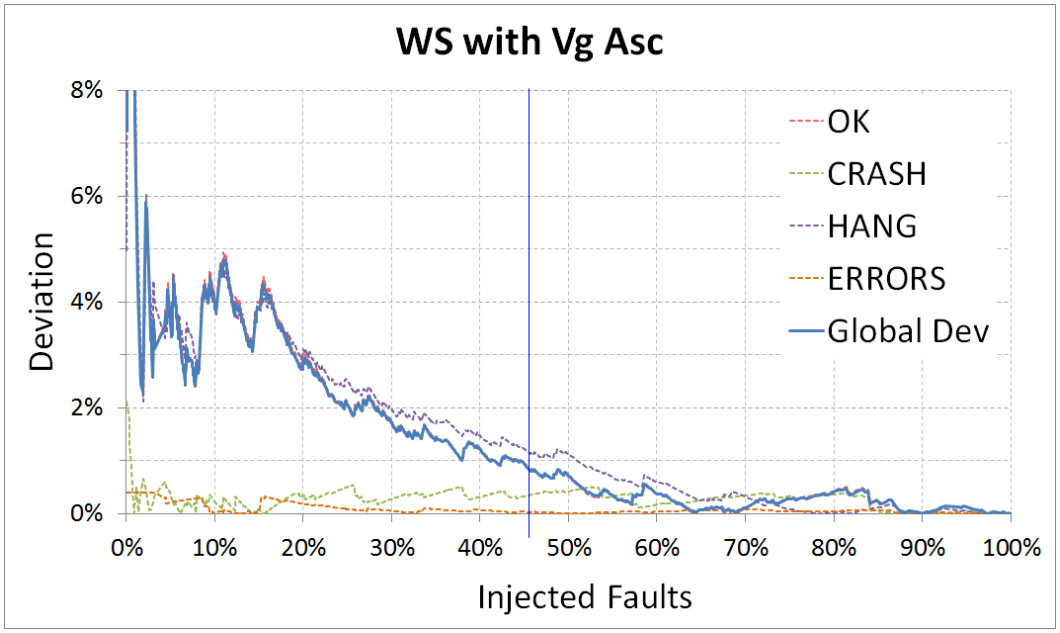


(a)

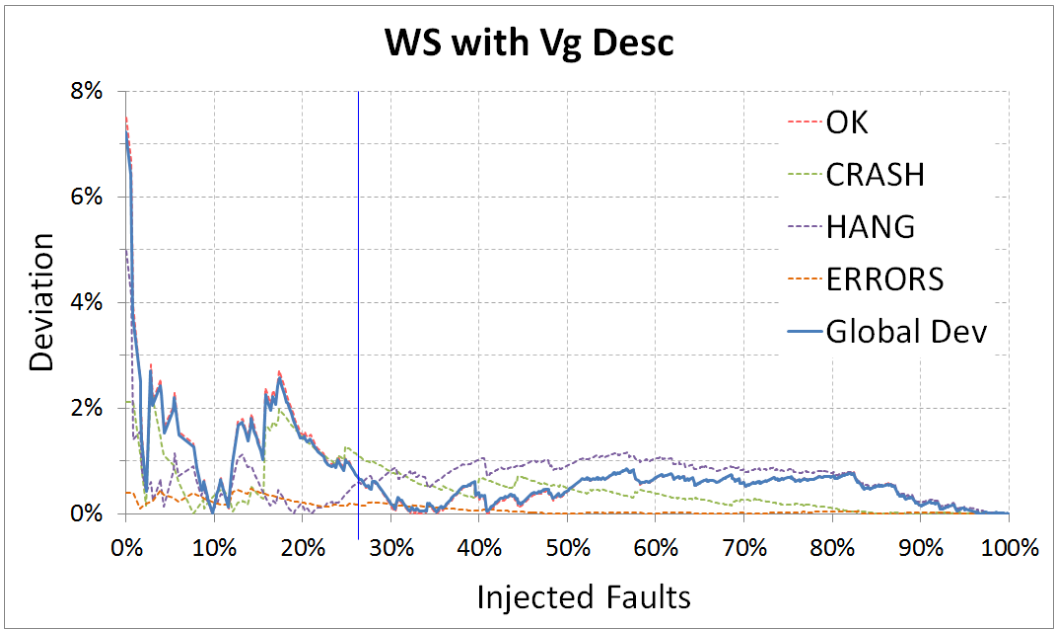


(b)

Figure 6-11 - Deviations for each failure mode in the WS experiments, considering the LOC strategy. (a) LOC Asc. (b) LOC Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

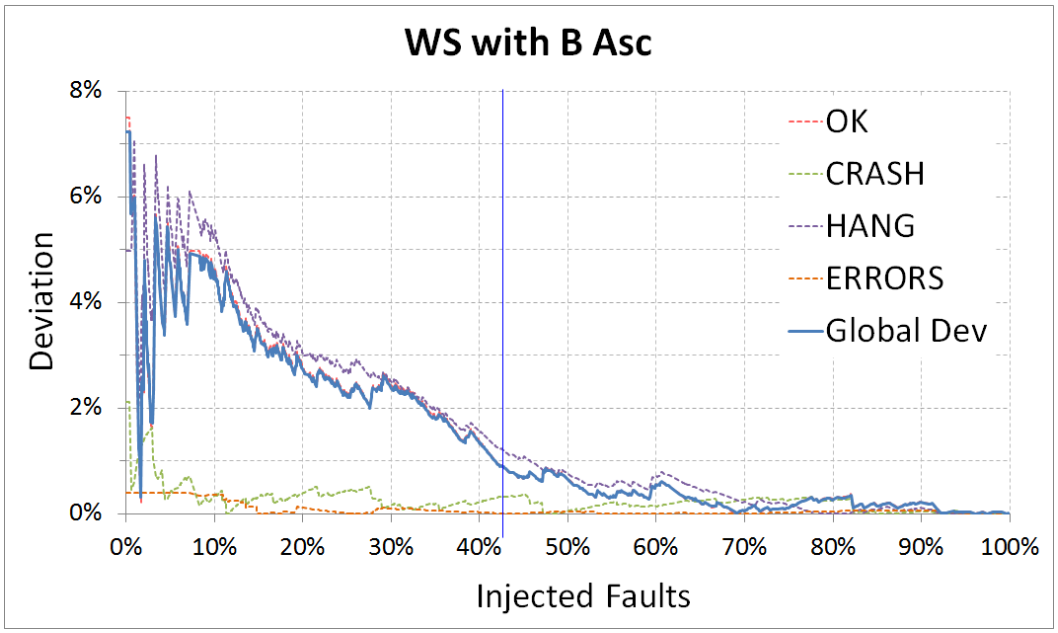


(a)

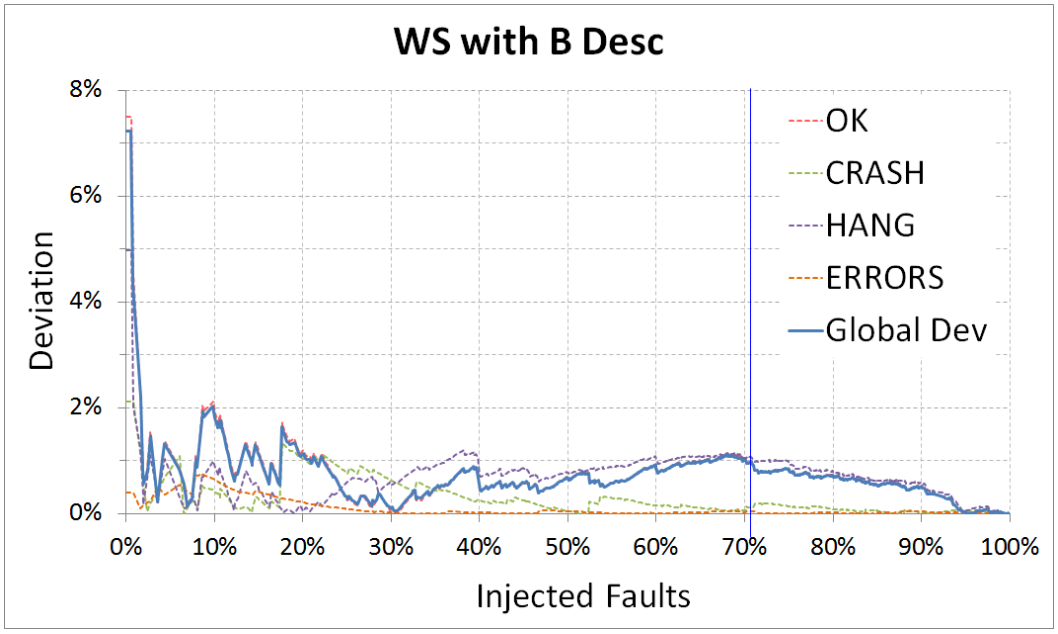


(b)

Figure 6-12 - Deviations for each failure mode in the WS experiments, considering the Vg strategy. (a) Vg Asc. (b) Vg Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

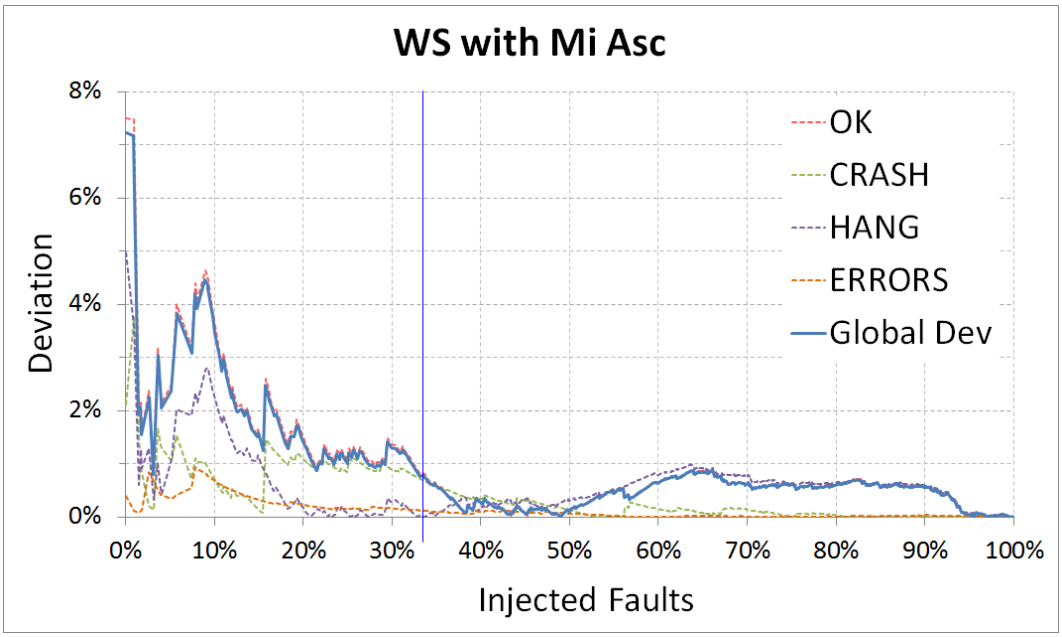


(a)

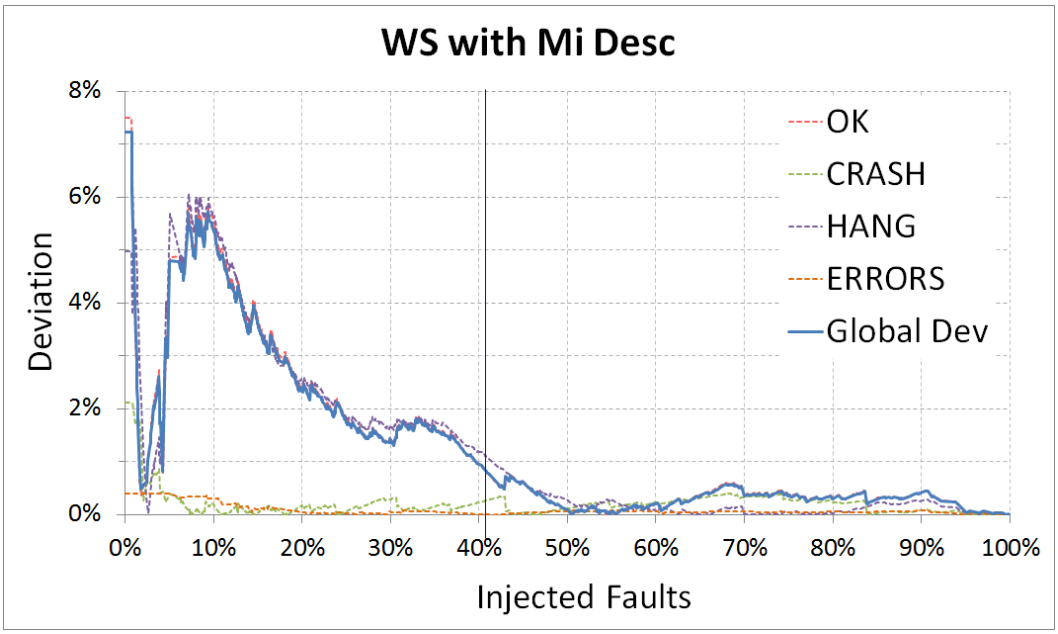


(b)

Figure 6-13 - Deviations for each failure mode in the WS experiments, considering the B strategy. (a) B Asc. (b) B Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

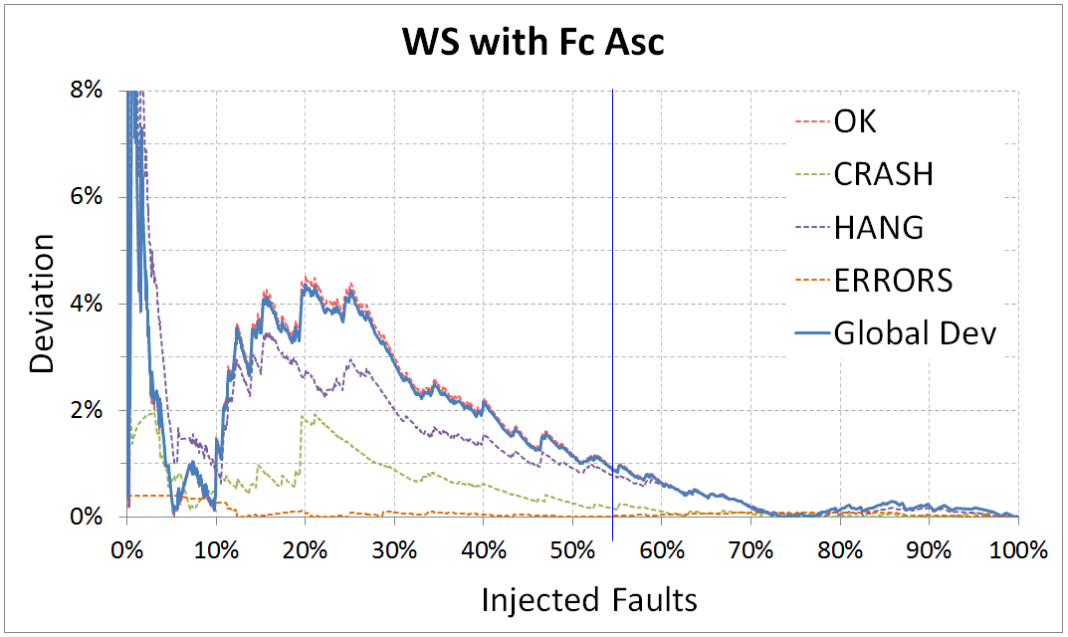


(a)

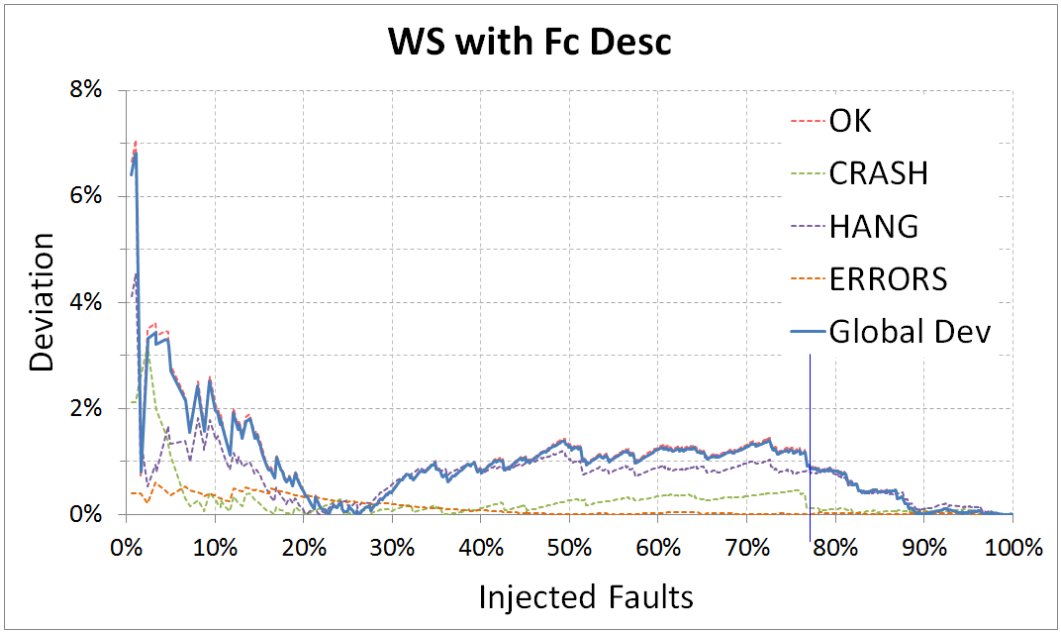


(b)

Figure 6-14 - Deviations for each failure mode in the WS experiments, considering the Mi strategy. (a) Mi Asc. (b) Mi Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.



(a)



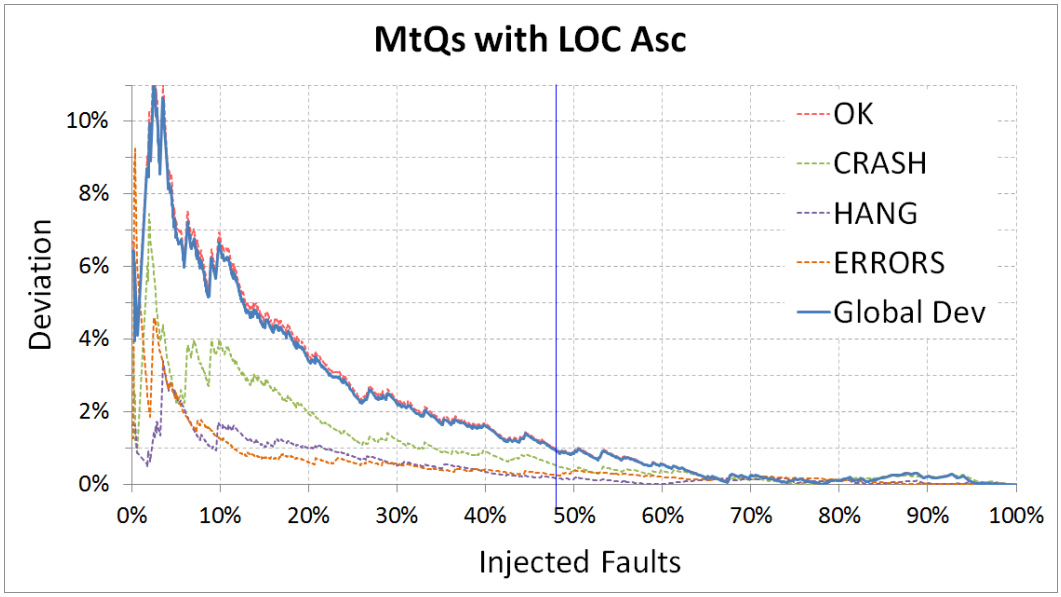
(b)

Figure 6-15 – Deviations for each failure mode in the WS experiments, considering the Fc strategy. (a) Fc Asc. (b) Fc Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

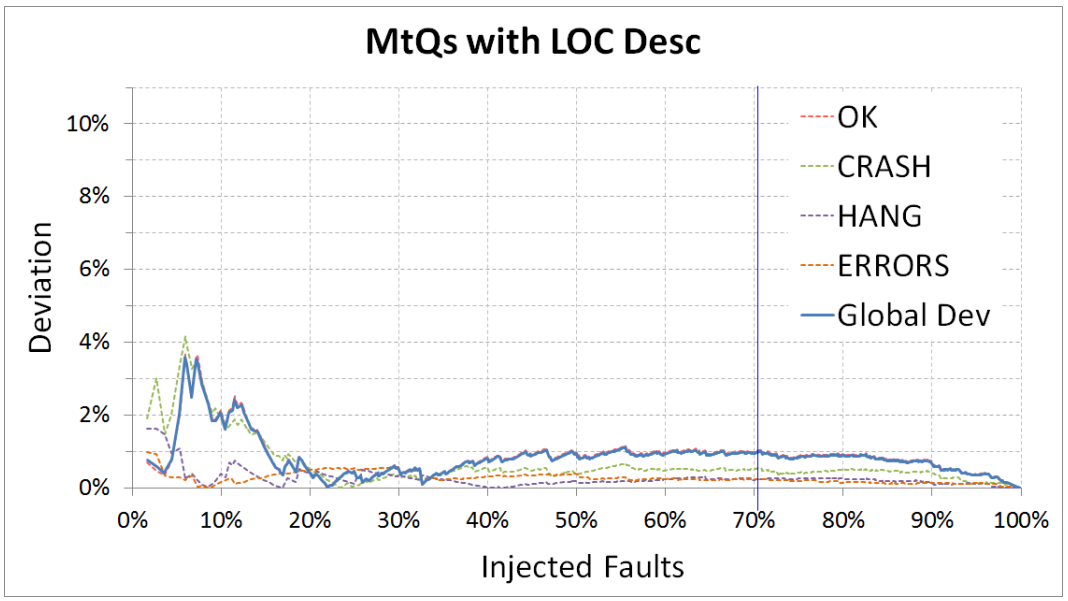
Concerning the MtQs experiments, like in the WS, we can observe from the charts from Figure 6-16 to Figure 6-20, that all approaches lead to convergence lines with similar behavior, considering their respective orders (Asc or Desc), except for the Mi approach (justified by the definition of the Maintainability index, Mi, as explained above). In these experiments, a more detailed analysis of the charts confirms that the best strategy is still the Vg Desc, for errors up to 2%. For errors less than 0.5%, the Mi Desc criterion is the best choice to select the subset of faults.

Considering both benchmark systems, these charts confirm that the best strategies for higher errors (greater than 2%) are those that have a worse performance considering lower errors (around 0.5%), and vice-versa. On the other hand, considering the behavior similarities of all the approaches, even with different types of SUBs (the WS, representing relative large and complex systems, and MtQs, representing a much smaller benchmark system) the charts and the data suggest that the Vg criteria (Asc, for errors lower than 0.5%, and Desc for greater errors) is a good global choice to answer our initial question: how to choose an adequate fault injection target, and thus reduce the total software fault injection experiments, without restricting the benchmark scope.

Despite the better performance of the Vg strategy, the LOC approach (in machine code) still shows to be a good strategy (in Asc order, for errors lower than 0.5%, and Desc for greater errors). The LOC approach is of particular importance because it is easier to obtain than all the other software measures (though always more complex than the random selection) and it does not require the availability of the target source code. Furthermore, unlike the other software metrics, the LOC strategy does not require the use of any complementary tool in order to analyze the code, as it can be obtained directly from the analysis of the OS kernel binary.

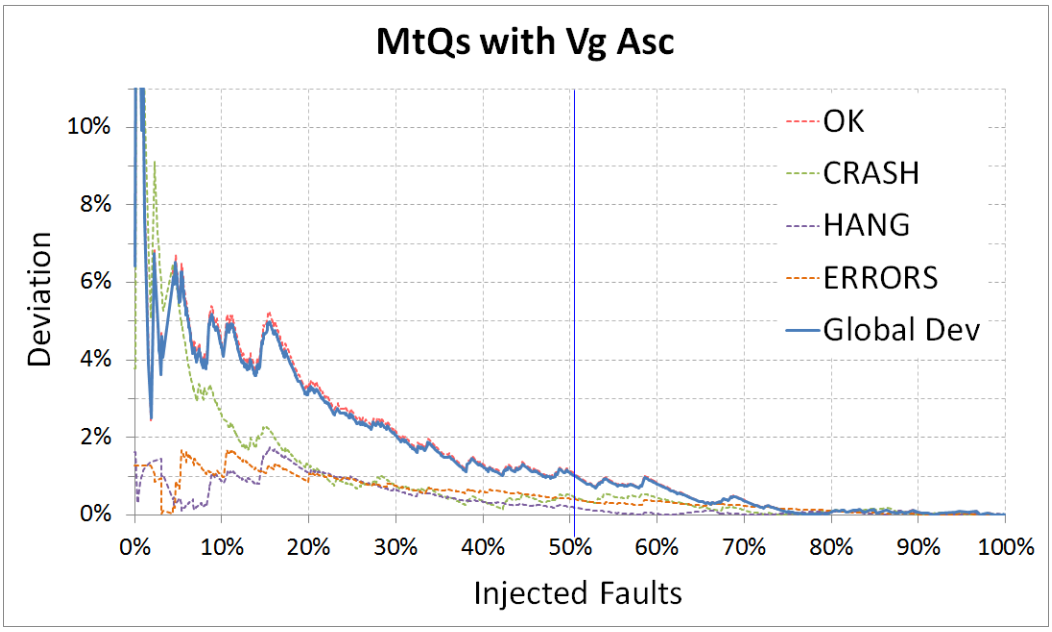


(a)

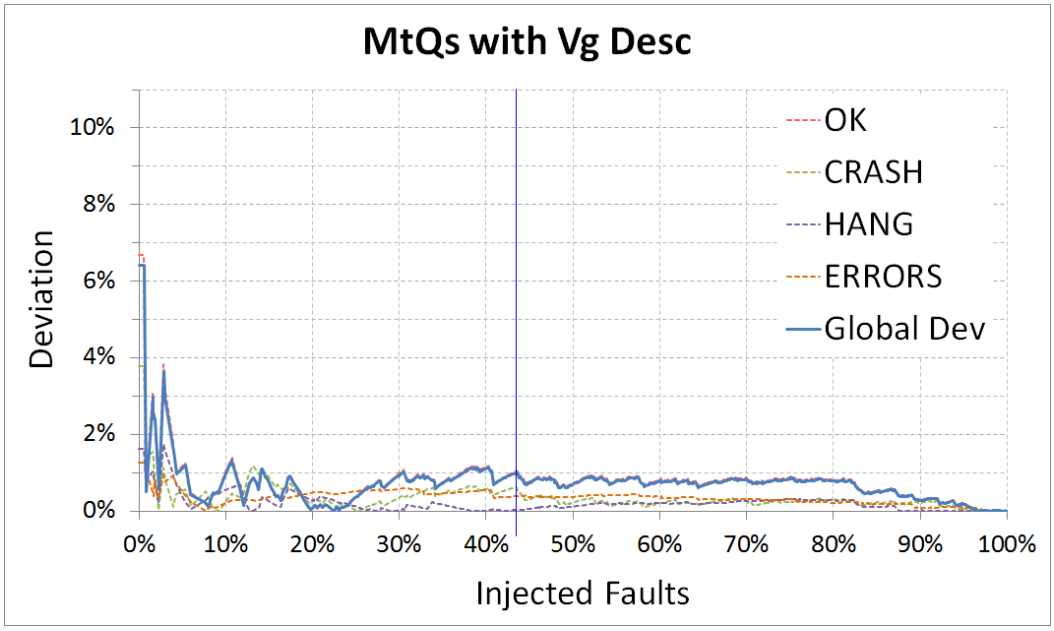


(b)

Figure 6-16 - Deviations for each failure mode in the MtQs experiments, considering the LOC strategy. (a) LOC Asc. (b) LOC Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

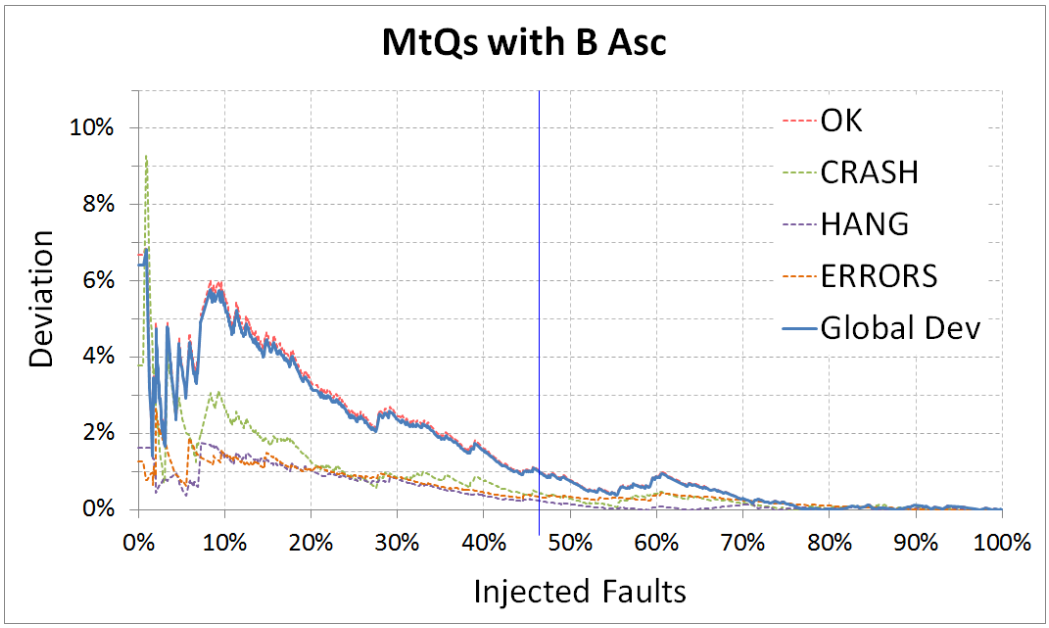


(a)

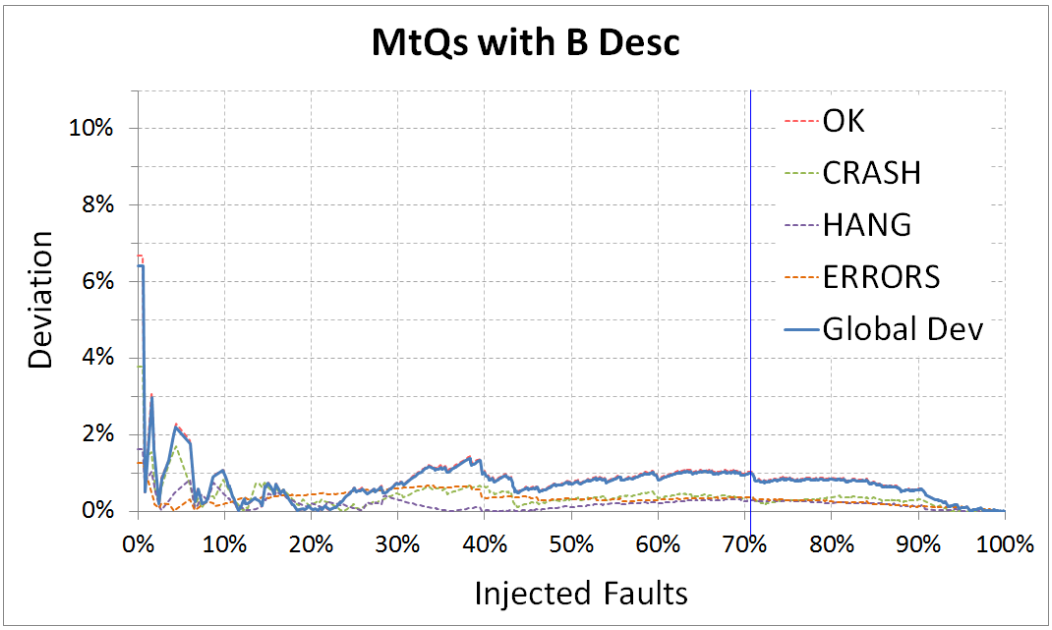


(b)

Figure 6-17 - Deviations for each failure mode in the MtQs experiments, considering the Vg strategy. (a) Vg Asc. (b) Vg Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

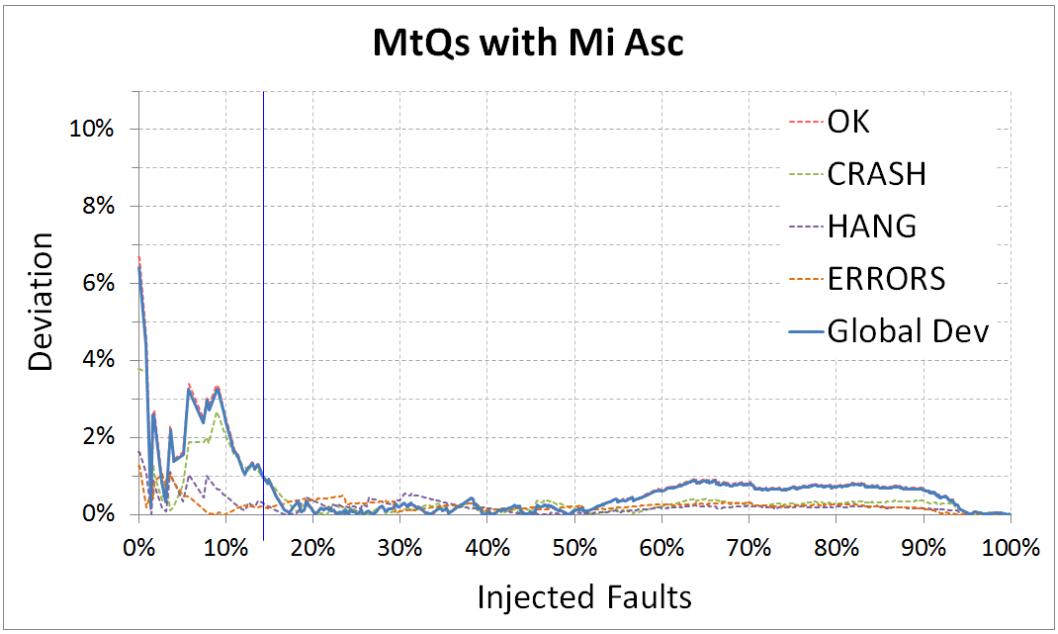


(a)

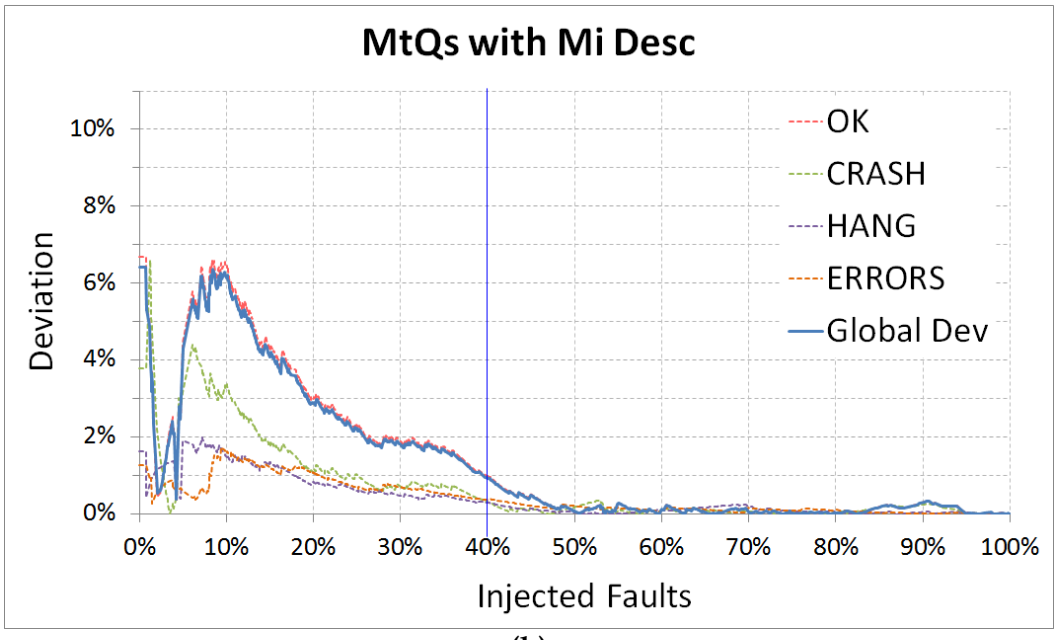


(b)

Figure 6-18 - Deviations for each failure mode in the MtQs experiments, considering the B strategy. (a) B Asc. (b) B Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

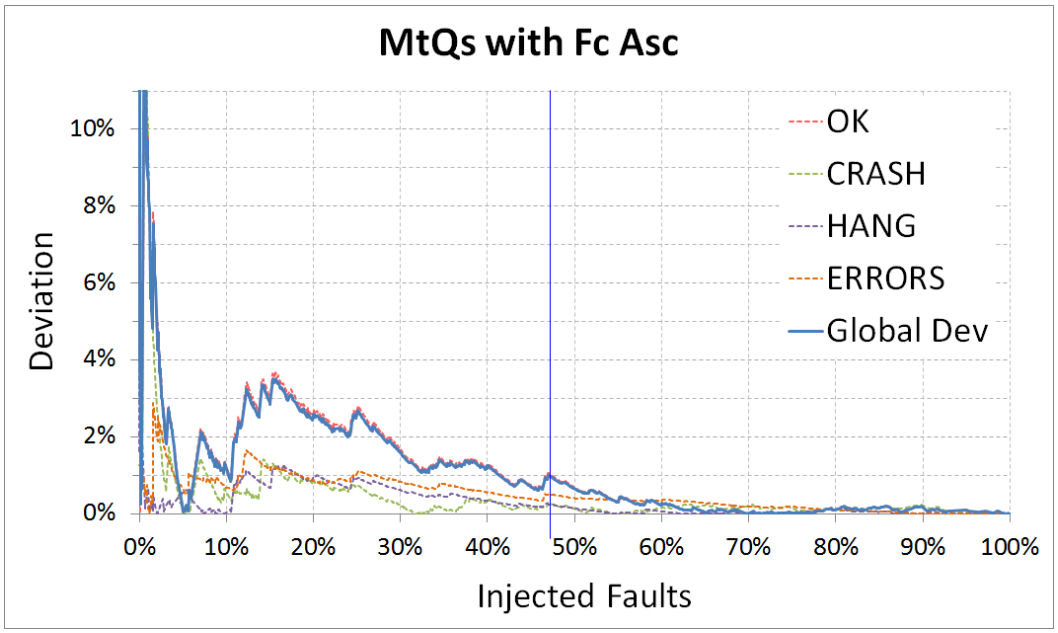


(a)

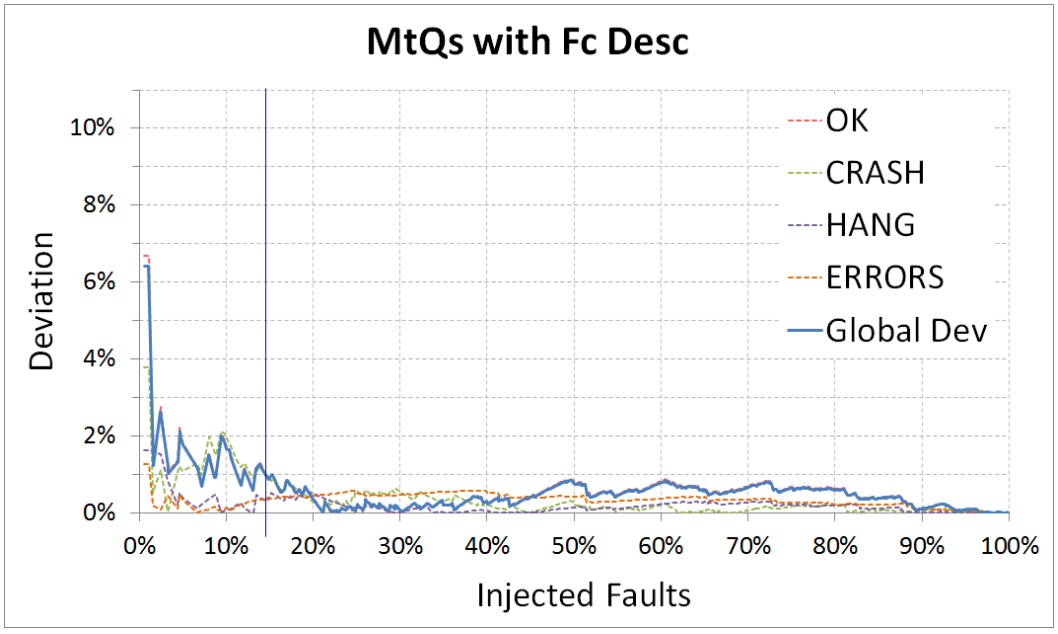


(b)

Figure 6-19 - Deviations for each failure mode in the MtQs experiments, considering the Mi strategy. (a) Mi Asc. (b) Mi Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.



(a)



(b)

Figure 6-20 - Deviations for each failure mode in the MtQs experiments, considering the Fc strategy. (a) Fc Asc. (b) Fc Desc. The vertical blue line indicates the percentage of injected faults needed to achieve a global deviation of 1%.

The results show that if we choose to start the WS experiments by the OS kernel functions with lower Vg (Vg Desc approach), after the injection of 25.05% of the total faults (5,230 fault injections), we obtain a d_g (Global Dev) value less or equal than 1%. The same happens for each one of the df_i (deviations of individual failure modes). In this way, we can reduce the fault injection experiments by approximately 75%, representing an enormous save of time in carrying out the benchmark experiments. Considering the total time needed to inject all the 20,875 faults in the WS experiments, we can estimate the reduction time of the total experiments in, approximately, 5,020 hours.

6.4 Proposal strategy for faultload reduction

Considering the results of the experimental evaluation carried out (presented in previous sections), a generic approach can be followed in order to solve the problem of the large size of the faultload, which arises in benchmarking the dependability of large and complex systems.

The proposed approach consists in the generation of an accurate faultload, specifically created for a given target system, and encompasses the following steps:

1. Obtain the complete list of target functions that should be considered as targets of the software fault injection (the OS kernel functions were considered in the conducted experimentation study).
2. Analyze all the functions listed in the previous step in order to obtain the correspondent software metrics (Vg or LOC, according to the results obtained in the experimental evaluation study). It is worth pointing out that, despite the better global performance of

the Vg strategy, the LOC approach also proved to be a good strategy. Moreover, the LOC software metrics is a lot easier to obtain than the Vg (even when compared with the other software metrics), and it is does not require the availability of the source code. The LOC metric can be directly obtained from the target functions binary code, which make this software metric especially adequate for COTS and COTS-based systems.

3. Sort the list of functions based on the selected software metric and in the intended order. According to the results obtained in the experimental study, for a greatest reduction on the size of the faultload, the descending order should be used. On the other hand, if it is accepted to have a faultload with a greater number of faults, the ascending order should be chosen instead.

4. Generate the faultload using the G-SWFIT (Generic Software Fault Injection) methodology [Durães *et al.* 2006] to determine the set of software faults that can be injected in each of the functions listed in the previous step (the presented research work used a tool provided by the author of the G-SWFIT methodology). The G-SWFIT technique consists in the scanning of the target code for specific low-level instruction patterns (sequence of machine code instructions) in order to emulate high-level software faults through the modification of the ready-to-run binary code of the target software component. It uses a set of operators for software fault emulation through low-level code mutations based on an extensive collection of real software faults, found in field.

5. Tailor the whole set of faults generated in order to obtain a reduced size faultload containing a given number of faults. The error imposed by the reduction of the number of faults can be estimated, according to our research. In other words, the faultload is calibrated for a given error bound. According to the conducted experimental study, using the Vg Desc or the LOC Desc

approaches, the complete faultload can be reduced to merely 4,000 software faults in order to obtain an expected maximum error of 2% (which seems to be a reasonable error for dependability benchmarks). Moreover, this faultload is adequate for dependability benchmarks, regardless of the complexity of the BT system (as evidenced by the values presented in Tables 6-3 and 6-4).

It should be noticed that the faultload generated using the proposed approach is specifically generated for the selected target system. Different targets systems should originate different faultloads.

As a result of the presented study, two ready-to-use calibrated faultloads are made available in http://eden.dei.uc.pt/~pncosta/. They were specifically generated for the target system used in the fault injection campaign carried out on this research work – the Linux RedHat 7.3 operating system (kernel version 2.4.18-3). The faultloads were generated according to the mentioned approach, using the Vg Desc and the LOC Asc strategies and contain 4,000 and 13,000 software faults, respectively. Concerning the errors induced by the use of the provided faultloads, our research study suggests that it is lower than 2% for the faultload based on the Vg Desc (the smaller faultload) and lower than 0.5% for the faultload based on LOC Asc (the larger faultload).

The faultloads generated with the proposed approach are especially useful for dependability benchmarks, as the error induced by the reduction of the number of faults was estimated on the presented experimental evaluation and measured against the results obtained with the complete faultload.

6.5 Summary

This chapter described the testbed used to evaluate different strategies to guide the fault injection target selection of dependability benchmarks and reduce the required fault injection experiments, without restricting the benchmark scope and keeping accurate results. It presents and analyzes the results obtained with an exhaustive set of fault injection experiments using a comprehensive faultload, which includes all possible software target locations of an operating system kernel (the complete set of the kernel OS functions, referred in kernel symbols table), resulting in one of the most extensive fault injection studies ever reported. More than 41 thousand of continuous fault injection experiments, carried out in more than 2 years, show that the fault injection experiments of a dependability benchmark can be reduced by more than 75%, maintaining the induced error below 1%. The effectiveness of the innovative approach is demonstrated with two real and different systems: a web-server dependability benchmark and a large-scale integer vector sort application extended with performance and quality measures.

The proposed methodology allows answering the problem of extending the use of dependability benchmarks to large and complex systems, making them feasible and practicably applied. It is worth pointing out that such benchmarks usually take several months or even years due to its large faultload size, which means that, in practice, it is not possible to execute them.

Chapter 7

Conclusion

This is the last chapter of this thesis and it provides an overview of the research work carried out in recent years, in the field of dependability benchmarking, at the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra.

7.1 Overview and future work

Dependability benchmarks should provide generic, cost-effective and reproducible ways for characterizing the behavior of components and computer systems in the presence of faults, allowing the quantification of dependability attributes or the characterization of system into well-defined dependability classes.

A key element in dependability benchmarks is the existence of a suitable fault injection tool to support the experiments. Dependability benchmarks must include fault injectors with very specific features: (i) they should be very easy to install and use, without the need for any complex setup or installation procedure; (ii) have high level of portability; (iii) have very low intrusiveness; (iv) be capable of injecting faults in both user and

system spaces; (v) and in code and data segments of any process, irrespective of their complexity; (vi) be independent of the availability of any source code of any system component or user process, (vii) be dynamically linked into a target system; and (viii) be compatible with the latest and most advanced software fault models.

Despite all the developments, none of the existing fault injection tools (presented in section 2.4.3) satisfied these requirements. In order to fulfill this gap, this work presents a pioneering SWIFI tool, named DBench-FI (Dependability Benchmarking Fault Injector), specially developed for dependability benchmarking. It has a unique set of features, required by that type of application: very low intrusiveness, capable of injecting both in user and system space, does not require application source code to be available, can be dynamically loaded into a system, and can inject even on applications that are already running when it is installed.

The methodology used in its design, based on the OS kernel schedule upgrading algorithm, together with a carefully crafted integration with the scheduler and memory management functions, constitutes the main innovation of this SWIFI tool, and is responsible for the unique characteristics presented by the fault injector. The DBench-FI enables a breakthrough in the areas of fault injection and dependability benchmarking, opening new perspectives hardly achievable with existing methods and making it one of the most versatile fault injectors available.

The current version of DBench-FI is adequate for the injection of hardware faults (intermittent and transient faults) in the systems memory, as well as for software faults, according to the G-SWFIT model - the state-of-the-art in software faults model. It is a central tool for the experimental evaluation presented in this thesis (chapter 6). Future versions of DBench-FI can be easily extended to include the majority of the existing fault models of Xception fault injector [Carreira *et al.* 1998b], such as spatial fault triggers and the capability to inject faults in processor resources.

Another major challenge in the design of dependability benchmarks is the definition of the faultload. Concerning software faultloads, that difficulty is further increased because of the known difficulties in assuring fault representativeness and the need of complex fault emulation methods.

Faultloads based on software faults had already been proposed. However, in order to assure the necessary representativeness, they require a large number of fault injection locations and, consequently, a huge number of experiments. That problem is even more dramatic in large and complex systems, where the execution time of those dependability benchmarks can take months or years due the mentioned faultload size.

This thesis presents the results of comprehensive fault injection experiments performed during more than two years of continuous fault injection runs in two completely different applications: a real web-server dependability benchmark and a large-scale integer vector sort client-server application extended with performance and quality metrics. The goal was to define the best strategy to reduce the number of faults while keeping accurate dependability benchmark results.

The reduction of the number of faults is achieved by an approach to guide the fault injection target selection in the code of the target systems. The goal is to identify the software fault target locations that assure good accuracy in the dependability benchmarks experiments while reducing dramatically the time needed to run the benchmark (because the number of faults is highly reduced).

The fault reduction strategy is based on measures of the target code, namely, Lines Of Code (LOC), the Extended Cyclomatic Complexity (Vg), Halstead's Delivery Bugs (B), Maintainability Index (Mi) and Functional Complexity (Fc). A randomly chosen subset of targets among the full set of injection targets, following a uniform distribution, is also studied (RandSF). In this case, for each subset, percentage of the full set, 2000 experiences have been carried out.

The results presented in this thesis extends our initial experimental study [Costa *et al.* 2009] (presented in section 5.3.1- Preliminary assessment study), as we consider the whole operating system kernel of the SUB (referred by the exported kernel symbols table) as the set of targets to establish the benchmark reference results instead of just the OS system calls used by the benchmark.

A study of the quality and usefulness of the dependability benchmark results for each approach is presented, and we can conclude that, in what concerns software fault injection, using the Vg criteria to choose the target functions for fault injection, allow a faster achieving of identically results, with respect to failure modes, globally and individually considered. The results show that we can reduce the fault injection experiments by approximately 75%, maintaining the induced error (global deviation) below 1%. This represents an enormous save of time in carrying out the benchmark experiments, especially in large and complex systems.

Despite this choice, the LOC approach (in machine code) also proved to be a valid and interesting strategy, especially if we consider that it is easier to obtain than all the other measures. Moreover, it is highly suitable for systems where the source code is not available for analysis or whether the tool for the software metrics analysis is unavailable. Furthermore, without being the best approach, random subsets of the software fault injection targets have also showed to be a valid strategy.

Besides these conclusions, some other relevant observations should be taken into account:

- In order to guide the target selection and reduce the number of faults, the best strategies for higher errors (within the range of 1% to 4%) are the worst ones when errors are intended to be smaller (lower than 0.5%), and vice- versa.
- The experiments performed with either a complex and large workload or a smaller and simpler one show that, regardless of

the strategy used, the ascending order (Asc) is the best one for very low errors (lower or equal to 0.5%). It can be stated that the descending orders (Desc) are the best for errors between 1% and 4%. The Mi criteria is an exception to this rule, since, contrariwise to what happens with the other metrics, Mi is greater for smaller and less complex functions.

- In order to keep the error lower than 0.5%, the number of injected faults is identical in both benchmark systems, despite the great differences in their workloads. This reveals independence between the number of faults and the complexity of the benchmark target, for very low errors.

It should be noticed that the complete workload-faultload space is in fact huge and testing the complete space is truly impossible. Thus, as performing a large set of experiments covering many points in the space workload-faultload is unfeasible, this study uses a worst case example, which is a completely different workload, concerning the workload complexity and the required computer resources. In fact, the integer vector sort application is very different from the real web-server dependability benchmark. The similarity of the obtained results in these two completely different systems seems to indicate that is reasonable to assume that the reduced fault set is a good approximation of the comprehensive fault set. It is worth mentioning that the faults are applied to the operating system and the different workloads represent different points in the workload space.

Future implementations of dependability benchmarks may encompass compact and representative faultloads generated according the approach presented in this research study. The presented methodology can be used in the future with new fault injection targets in order to generate accurate and specific faultloads. New applications can also be used as benchmark targets in order to evaluate the impact of the injected faults.

7.2 Contributions

Taken as a whole, the main contributions of this work can be summed up in the following items:

- To provide a software fault injector compatible with the demanding requirements of dependability benchmarks. Namely, it should be very easy to install and use, have very low intrusiveness, be capable of injecting faults in both user and system spaces, and in code and data segments of any process, irrespective of their complexity, be independent of the availability of any source code of any system component or user process, be dynamically linked into a target system and be compatible with the latest and most advanced software fault models. Concerning this last requirement, it was considered essential the compatibility of the fault injector with the Generic Software Fault Injection Technique (G-SWFIT) [Durães *et al.* 2006] – the state-of-the-art in software faults model. G-SWFIT is based on a set of operators for software fault emulation through low-level code changes in the target executable code, mimicking the most common types of real software faults. These operators resulted from a field study based on the analysis and classification of more than 600 software faults found in real software applications. The developed tool consists in one of the most versatile software fault injectors currently available.
- To define and evaluate different hypothesis for the reduction of the number of software fault injection experiments. The evaluation is based on the analysis of the error obtained in consequence of the reduction of the fault injection experiments. This study uses the results obtained with a comprehensive faultload that includes all possible software target locations (the complete set of the kernel OS functions, referred in kernel

symbols table), resulting in one of the most extensive fault injection studies ever reported.

- To present a strategy to guide the fault injection target selection of dependability benchmarks and to reduce the required number of software faults, thus decreasing the execution time of the benchmark, maintaining, simultaneously, their usefulness and representativeness. The proposed methodology is especially useful in large and complex systems, where the experimentation time can be severely reduced without compromising the dependability benchmark results. Conducted experiments showed that the fault injection experiments can be reduced by more than 75%, maintaining the induced error below 1%. This method will open the possibility to extend the dependability benchmarks to large and complex systems, making them feasible and practicably applied (such benchmarks usually take several months or even years due to its large faultload size).
- To provide accurate and ready-to-use faultloads, compatible with a given target system. These faultloads can be used as the faultload component of dependability benchmarks, as the error introduced by the reduction of the number of faults was measured against the results obtained with the complete faultload. This strategy allows us to provide reduced sized faultloads that assure an error lower than a given limit.

Bibliography

[Aidemark *et al.* 2001] Aidemark, J., Vinter, J., Folkesson, P., Karlsson, J., "GOOFI: Generic Object-Oriented Fault Injection Tool", IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2001, pp. 83-88, 2001.

[Albinet *et al.* 2004] Albinet, A., Arlat, J., Fabre, J.-C., "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel", Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2004, pp. 867-876, 2004.

[Almeida *et al.* 2010] Almeida, R., Poess, M., Nambiar, R., Patil, I., Vieira, M., "How to Advance TPC Benchmarks with Dependability Aspects", Proc. of the Second TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems, TPCTC 2010, pp. 57-75, Sep. 2010.

[Andrews *et al.* 2005] Andrews, J.H., Briand, L.C., Labiche, Y., "Is Mutation an Appropriate Tool for Testing Experiments?", Proc. 27th Int. Conference on Software Engineering, ICSE 2005, pp. 402-411, May 2005.

[Arlat *et al.* 1990a] Arlat J., Crouzet, Y., Laprie, J.-C., "Fault Injection for the Experimental Validation of Fault Tolerance", LAAS Report 90415, 1990.

[Arlat *et al.* 1990b] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., Martins, E., Powell, D., "Fault Injection for the Dependability Validation: A Methodology and Some

Applications", IEEE Transactions on Software Engineering, Vol. 16, No. 2, pp. 166-182, Feb, 1990.

[Arlat *et al.* 1993] Arlat, J., Costes, A., Crouzet, Y., Laprie, J.-C., Powell, D., "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems", IEEE Transaction on Computers, Vol. 42, No. 9, pp. 913-923, Aug. 1993.

[Arlat 2002] Arlat, J., "From Experimental Assessment of Fault-Tolerant Systems to Dependability Benchmarking", Workshop on Fault-Tolerant Parallel and Distributed Systems, FTPDS'02, joint organized with the International Parallel and Distributed Processing Symposium, IPDPS'02, April 2002.

[Arlat *et al.* 2002] Arlat, J., Fabre, J.-C., Rodríguez, M., Sales, F., "Dependability of COTS Microkernel-Based Systems", IEEE Transactions on Computers, Vol. 51, No. 2, pp. 138-163, Feb, 2002.

[Avizienis 1985] Avizienis, A., "The N-Version Approach to Fault-Tolerant Software", IEEE Trans. on Software Engineering, Vol. SE-11, No. 12, pp. 1491-1501, Dec. 1985.

[Avizienis *et al.* 2000] Avizienis, A., Laprie, J.-C., Randell, B., "Fundamental Concepts of Dependability", Proc. 3rd Information Survivability Workshop, ISW, pp. 7-12, Oct. 2000.

[Avizienis *et al.* 2004] Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Trans. on Dependable and Secure Computing, Vol. 1, No. 1, pp. 11-33, 2004.

[Avresky *et al.* 1996] Avresky, D., Arlat, J., Laprie, J.-C., Crouzet, Y., "Fault Injection for Formal Testing of Fault Tolerance", IEEE

Transactions on Reliability, Vol. 45, No. 3, pp. 443-455, Sept. 1996.

[Barbosa *et al.* 2001] Barbosa, E.F., Maldonado, J.C., Vincenzi, A.M.R., "Toward the Determination of Sufficient Mutant Operators for C", Software Testing, Verification and Reliability, vol. 11, no. 2, pp. 113-136, May 2001.

[Barbosa *et al.* 2011] Barbosa, R., Karlsson, J., Yu, Q., Mao, X., "Toward Dependability Benchmarking of Partitioning Operating Systems", Proc. of the IEEE/IFIP 41st Int. Conf. on Dependable Systems and Networks, DSN 2011, pp. 422-429, June 2011.

[Basso *et al.* 2009] Basso, T., Moraes, R., Sanches, B., Jino, M., "An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults", Workshop de Testes e Tolerância a Falhas, pp. 1-13, 2009.

[Blair *et al.* 1992] Blair, M., Obenski, S., and Bridickas, P., "Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia", Tech. Report GAO/IMTEC-92-26, U.S. General Accounting Office, Feb. 1992.

[Bovet *et al.* 2005] Bovet, D.P., Cesati, M., Understanding the Linux Kernel, 3rd Ed., O'Reilly Media, Nov. 2005.

[Brown *et al.* 2000] Brown, A., Patterson, D.A., "Towards Availability Benchmark: A Case Study of Software RAID Systems," Proc. USENIX Ann. Technical Conf., pp. 263-276, June 2000.

[Brown *et al.* 2001] Brown, A., Patterson, D.A., "To Err is Human", First Workshop on Evaluating and Architecting System Dependability, EASY 2001, 2001.

[Brown *et al.* 2002] Brown, A., Chung, L.C., Patterson, D.A., "Including Human Factor in Dependability Benchmarks", International

Conference on Dependable Systems and Networks, DSN 2002, 2002.

[Brown *et al.* 2004a] Brown, A., Chung, L., Kakes, W., Ling, C., Patterson, D.A., "Dependability Benchmarking of Human-Assisted Recovery Processes", IEEE/IFIP Int. Conf. Dependable Systems and Networks, DSN 2004, Florence, Italy, pp. 405-410, June 2004.

[Brown *et al.* 2004b] Brown, A., Hellerstein, J., Hogstrom, M., Lau, T., Lightstone, S., Shum, P., Yost, M. P., "Benchmarking Autonomic Capabilities: Promises and Pitfalls", Proc. Int. Conf. on Autonomic Computing (ICAC'04), 2004.

[Brown *et al.* 2004c] Brown, A., Hellerstein, J., "An Approach to Benchmarking Configuration Complexity", Proc. of the 11thACM SIGOPS European Workshop, Leuven, Belgium, September 2004.

[Brown *et al.* 2005] Brown, A., Redlin, C., "Measuring the Effectiveness of Self-Healing Autonomic Systems", Proc. 2nd Int. Conf. on Autonomic Computing (ICAC'05), 2005.

[Buchacker *et al.* 2003] Buchacker, K., Tschaeche, O., "TPC Benchmark-C Version 5.2 Dependability Benchmark Extensions", http://www.faumachine.org/papers/tpcc-depend.pdf, 2003.

[Budd 1981] Budd, T., "Mutation Analysis: Ideas, Examples, Problems, and Prospects", Computer Program Testing, (Eds., Chandrasekaran, B., Radicchi, S.), pp. 129-134. North Holland, 1981.

[Budd *et al.* 1982] Budd, T., Angluin, D., "Two Notions of Correctness and Their Relation to Testing", Acta Informatica, vol. 18, no. 1, pp. 31-45, Mar. 1982.

[Carey *et al.* 1993] Carey, M., Witt, D., Naughton, J., "The OO7 Benchmark", Proceedings of the 1993 ACM SIGMOD international conference on Management of data, SIGMOD 1993, pp. 12-21, May 1993.

[Carreira *et al.* 1995] Carreira, J., Madeira, H., Silva, J. G., "Xception: Software Fault Injection and Monitoring in Processor Functional Units", 5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5), Sep. 1995.

[Carreira *et al.* 1998a] Carreira, J., Silva, J.G., "Why do Some (weird) People Inject Faults?", ACM Software Engineering Notes, pp. 42-43, Jan. 1998.

[Carreira *et al.* 1998b] Carreira, J., Madeira, H., Silva, J.G., "Xception: A technique for the Experimental Evaluation Dependability in Modern Computers", IEEE Trans. Software Eng., Vol. 24, Nº2, pp. 125-136, Feb. 1998.

[Carreira *et al.* 1999] Carreira, J., Costa, D., Silva, J.G., "Fault injection spot-checks computer system dependability", IEEE Spectrum, vol. 36, no. 8, pp. 50-55, Aug. 1999.

[Carter 2006] Carter, P.A., PC Assembly Language, http://www.drpaulcarter.com/pcasm/, July 2006.

[CASDCST 1992] Committee to Assess the Scope and Direction of Computer Science and Technology of the National Research Council, "Computing the Future", Communications of ACM, vol. 35, no. 11, pp. 30-40, Nov. 1992.

[Chidamdber *et al.* 1991] Chidamber, S., Kemerer, C., "Towards a Metrics Suite for Object Oriented Design", 6th Annual ACM Conference on Object Oriented Programming Systems, Languages and Applications, OOPSLA'91, Oct. 1991.

[Chillarege *et al.* 1991] Chillarege, R., Kao, W., Condit, R., "Defect Type and its Impact on the Growth Curve", Proc. 13th Intl. Conf. on Software Engineering, pp- 246–255, 1991.

[Chillarege *et al.* 1992] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., Wong, M., "Orthogonal Defect Classification – A Concept for In-Process Measurements", IEEE Transactions on Software Engineering, Vol. 18, No. 11, pp. 943-956, Nov. 1992.

[Chillarege *et al.* 1995] Chillarege, R., Biyani, S., Rosenthal, J., "Measurement of failure rate in widely distributed software", Proc. 25th IEEE Int. Symposium on Fault Tolerant Computing, FCTS-25, pp. 424-433, 1995.

[Chillarege 1996] Chillarege, R., "Orthogonal Defect Classification", Handbook of Software Reliability Engineering, (Ed., Lyu, M.), IEEE Computer Society Press, McGraw-Hill, Chapter 9, 1996.

[Choi *et al.* 1992] Choi, G., Iyer, R., "Focus: An Experimental Environment foo Fault Sensitivity Analysis", IEEE Transactions on Computers, Vol. 41, No. 12, pp. 1515-1526, Dec. 1992.

[Chou 1997] Chou, T., "Beyond Fault Tolerance", IEEE Computer, Vol. 30, No. 4, pp. 47-49, April 1997.

[Chrissis *et al.* 2003] Chrissis, M., Konrad, M., Shrum, S., CMMI: Guidelines for process integration and product improvement, Addison-Wesley Professional, 2003.

[Christmansson *et al.* 1996a] Christmansson, J., Chillarege, R., "Generation of an Error Set that Emulates Software Faults Based on Field Data", Proc. 26th IEEE Fault Tolerant Computing Symp. (FTCS-26), pp. 304-313, June 1996.

[Christmansson *et al.* 1996b] Christmansson, J., Santhanam, P., "Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms – Criteria for Error Selection Using Field Data on Software Faults", Proc. 7th IEEE Int. Symposium on Software Reliability Engineering, ISSRE'96, Nov. 1996.

[Clark *et al.* 1995] Clark, J., Pradhan, D., "Fault Injection: A Method For Validating Computer-System Dependability", IEEE Computer, vol. 28, no. 6, pp. 47-56, June 1995.

[CMT] Testwell Oy Ltd, CMT++ Tool, Version 5.0, http://www.testwell.fi/cmtdesc.html, 2011.

[Constantinescu 2005a] Constantinescu, C., "Neutron SER characterization of Microprocessors", IEEE Conferecnce on Dependable Systems and Networks, DSN 2005, pp. 754-759, 2005.

[Constantinescu 2005b] Constantinescu, C., "Dependability Benchmarking using Environmental Tools", IEEE Annual Reliability and Maintainability Symposium, pp. 567-571, 2005.

[Costa *et al.* 2003] Costa, P., Vieira, M., Madeira, H., Silva, J.G., "Plug and Play Fault Injector for Dependability Benchmarking", Proc. of First Latin-American Symposium on Dependable Computing, LADC 2003, pp. 8-22, Oct. 2003.

[Costa *et al.* 2009] Costa, P., Silva, J.G., Madeira, H., "Dependability Benchmarking Using Software Faults: How to Create Practical and Representative Faultloads", Proc. 15th IEEE Pacific Rim International Symp. on Dependable Computing, PRDC-15, Shanghai, China, pp. 289-294, 2009.

[Cristian 1982] Cristian, F., "Exception Handling and Software Fault Tolerance", IEEE Trans. on Computers, Vol. c-31, No. 6, June 1982.

[Cukier *et al.* 1999] Cukier, M., Powell, D., Arlat, J., "Coverage Estimation Methods for Stratified Fault-Injection", IEEE Trans. on Computers, Vol. 48, No. 7, pp. 707-723, July 1999.

[Daran *et al.* 1996] Daran ,M., Thévenod-Fosse, P.: Software Error Analysis: A Real Case Study Involving Real Faults and Mutations, Proc. 3rd Symp. on Software Testing and Analysis, ISSTA-3, San Diego, USA, January, pp. 158-171, 1996.

[DBENCH] DBench – Dependability Benchmarking Project, Information Society Technology, IST-2000-25425, http://www.laas.fr/DBench/.

[DBENCH 2004] DBench - Dependability Benchmarking Project, European IST Program, IST-2000-25425, Final Report, May 2004.

[Delamaro *et al.* 1996] Delamaro, M.E., Maldonado, J.C., "Proteum-A Tool for the Assessment of Test Adequacy for C Programs," Proc. Conf. Performability in Computer Systems, pp. 79-95, July 1996.

[DeMillo *et al.* 1978] DeMillo, R., Lipton, R.J., Sayward, F.G., "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, vol. 11, no. 4, pp. 34-41, Apr. 1978.

[DeMillo *et al.* 1979] DeMillo, R., Lipton, R.J., Perlis, A., "Social Processes and Proofs of Theorems and Programs", Communications of the ACM, vol. 22, no. 5, pp. 271-280, May 1979.

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

[Dijkstra 1972] Dijkstra, E., "The Humble Programmer", Communications of the ACM, vol. 15, no. 10, pp. 859-866, 1972.

[]Dimov *et al.* 2010] Dimov, A., Chandran, S., Punnekkat, S., Nasir, A., Azam, N., "Mutation Testing Framework for Software Reliability Model Analysis and Reliability Estimation", Proc. Of the 6th Central and Eastern European Software Engineering Conference (CEE-SECR), Moscow, Russia, pp. 163-169. Oct. 2010.

[Do *et al.* 2006] Do, H., Rothermel, G., "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," IEEE Trans. on Software Engineering., vol. 32, no. 9, pp. 733-752, Sept. 2006.

[Dowson 1997] Dowson, M., "The Ariane 5 Software Failure". ACM SIGSOFT Software Engineering Notes, vol. 22, no. 2, pp. 84, Mar. 1997.

[Durães *et al.* 2002a] Durães, J., Madeira, H., "Characterization of Operating Systems Behavior in the Presence of Faulty Device Drivers Through Software Fault Emulation", Proc. 2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC-02), Tsukuba, Japan, pp. 201-209, 2002.

[Durães *et al.* 2002b] Durães, J., Madeira, H., "Emulation of Software Faults by Educated Mutations at Machine-Code Level", Proc. of the 13th IEEE Int. Symposium on Software Reliability Engineering, ISSRE'02, pp. 329-340, Nov. 2002.

[Durães *et al.* 2003a] Durães, J., Madeira, H.: "Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior", IEICE (Institute of the Electronics, Information and Communication Engineers) Transactions on Information and Systems, vol. 86, part 12, pp. 2563-2570, 2003.

[Durães *et al.* 2003b] Durães, J., Madeira, H.: "A Definition of Software Fault Emulation Operators: A Field Data Study", Proc. Int. Conf. Dependable Systems and Networks, DSN 2003, San Francisco, USA, 2003 (W. Carter Award).

[Durães *et al.* 2004a] Durães, J., Madeira, H., "Generic Faultloads Based on Software Faults for Dependability Benchmarking", Proc. Int. Conf. on Dependable Systems and Networks, DSN2004, Florence, Italy, IEEE CS Press, 2004.

[Durães *et al.* 2004b] Durães, J., Vieira, M., Madeira, H., "Dependability Benchmarking of Web-Servers", 23rd Int. Conf. on Computer Safety, Reliability and Security, SAFECOMP 2004, Potsdam, Germany, 2004.

[Durães *et al.* 2006] Durães, J., Madeira, H., "Emulation of Software Faults: A Field Data Study and a Practical Approach", IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 849-867, November 2006.

[Elling *et al.* 2008] Elling, R., Pramanick, I., Mauro, J., Bryson, W., Tang, D., "Analytical Reliability, Availability and Serviceability Benchmarks", Dependability Benchmarking for Computer Systems, (Eds. Kanoun, K., Spainhower, L.), Wiley-IEEE Computer Society Press, 2008.

[Folkesson *et al.* 1998] Folkesson, P., Svensson, S., Karlsson, J., "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection", Proc. 28th Int. Symposium on Fault-Tolerant Computing, FCTS-28, pp. 284-293, June 1998.

[Friginal *et al.* 2011] Friginal, J, Andrés, D., Ruiz, J.-C., Moraes, R., "Using Dependability Benchmarks to Support ISO/IEC SQuaRE", 17th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2011, pp. 28-37, Dec 2011.

[Fujita *et al.* 2012] Fujita, H., Matsuno, Y., Hanawa, T., Sato, M., kato, S., Ishikawa, Y., "DS-Bench Toolset: Tools for Dependability Benchmarking with Simulation and Assurance", Proc. of 42nd IEEE/IFIP Int. Conf. on Dependable Systems and Networks, DSN 2012, pp. 1-8 , June 2012.

[Ganek *et al.* 2003] Ganek, A.G., Corbi, T.A., "The dawning of the autonomic computing era", IBM Systems Journal, Vol. 42, Issue 1, pp. 5-18, January 2003.

[Garber 1996] Garber, L., "AOL Blackout Indicates Need for Reliable on-Line Systems", IEEE Computer, vol. 19, no. 9, pp. 16-18, September 1996.

[Geist *et al.* 1992] Geist, R., Offutt, A., Harris Jr., F., "Estimation and Enhancement of Real-Time Software Reliability through Mutation Analysis," IEEE Trans. on Computers, vol. 41, no. 5, pp. 550-558, May 1992.

[Goswami *et al.* 1997] Goswami, K., Iyer, R., Young, L. "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis", IEEE Transactions on Computers, Vol. 46, No. 1, pp. 60-74, Jan. 1997.

[Guerra *et al.* 2004] Guerra, P., Rubira, C., Romanovsky, A., Lemos, R., "A Dependable Architecture for COTS-Based Software Systems using Protective Wrappers", Architecting Dependable Systems II, volume 3069 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 144-166, 2004.

[Gray 1985] Gray, J., "Why Do Computers Stop and What Can Be Done About It?", Tandem Technical Report 85.7, June 1985.

[Gray 1990] Gray, J., "A Census of Tandem Systems Availability between 1985 and 1990," IEEE Trans. Reliability, vol. 39, no. 4, pp. 409-418, Oct. 1990.

[Gray *et al.* 1991] Gray, J., Siewiorek, D., "High-Availability Computer Systems", Computer, vol. 24, no. 9, pp. 39-48, Sept. 1991.

[Grottke *et al.* 2007] Grottke, M., Trivedi, K.S., "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate", IEEE Computer, vol. 40, no. 2, pp. 107-109, Feb. 2007.

[Halstead 1977] Halstead, M., "Elements of Software Science", Operating and Programming Systems Series, Volume 7. New York, Elsevier, 1977.

[Hamlet 1977] Hamlet, R., "Testing Programs with the Aid of a Compiler", IEEE Transactions on Software Engineering, Vol. SE-3, No.4, pp. 279-290, July 1977.

[Han *et al.* 1993] Han, S., Rosenberg, H., Shin, K., "DOCTOR: An Integrated Software Fault Injection Environment", Technical Report, University of Michigan, 1993.

[Henry *et al.* 1981] Henry, S., Kafura, D., "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, vol. SE-7, no. 5, pp. 510-518, Sept. 1981.

[Hosmer *et al.* 1989] Hosmer, D., Lemeshow, S., Applied Logistic Regression, John Wiley & Sons, 1989.

[Hsueh *et al.* 1997] Hsueh, M.-C., Tsai, T.K., Iyer, R.K., "Fault Injection Techniques and Tools", IEEE Computer, Vol. 30, No. 4, pp. 75-82, 1997.

[Hutchins *et al.* 1994] Hutchins, M., Foster, H., Goradia, T., Ostrand, T., "Experiments on the Effectiveness of Dataflow- and

Controlflow-Based Test Adequacy Criteria," Proc. Int. Conf. on Software Engineering, pp. 191-200, May 1994.

[IEEE 1994] IEEE Standard Classification for Software Anomalies, IEEE Std 1044-1993, 1994.

[IEEE 2010] IEEE Standard Classification for Software Anomalies, IEEE Std 1044-2009, Revision of the IEEE Std 1044-1993, 2010.

[IBMACI] IBM Autonomic Computing Initiative, http://www.research.ibm.com/autonomic, 2012.

[ISODIS 2009] International Organization for Standardization, "Product development: software level", ISO/DIS 26262-6, 2009.

[ISOIEC 2005] International Organization for Standardization, "ISO/IEC 25000: Software engineering-Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE", 2005.

[ISOIEC 2010] International Organization for Standardization, "ISO/IEC 25045: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) - Evaluation module for recoverability", 2010.

[Iyer 1995] Iyer, R., "Experimental Evaluation", 25th IEEE Int. Symposium on Fault Tolerant Computing, FCTS-25, Special Issue Silver Jubilee, pp. 115-132, Jun1 1995.

[Jalote 1994] Jalote, P., Fault Tolerance in Distributed Systems, Prentice Hall, 1994.

[Jarboui *et al.* 2002] Jarboui, T., Arlat, J., Crouzet, Y., Kanoun,K., "Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques", International Conference on

Dependable Systems and Networks, DSN 2002, pp. 331-336, June 2002.

[Jarboui *et al.* 2003] Jarboui, T., Arlat, J., Crouzet, Y., Kanoun, K., Marteau, T., "Impact of Internal and External Software Faults on the Linux Kernel," IEICE (Institute of the Electronics, Information and Communication Engineers) Transactions on Information and Systems, Special Issue on Dependable Computing, vol. E86-D, no. 12, pp. 2571-2578, December 2003.

[Jenn *et al.* 1995] Jenn, E., Arlat, J., Rimén, M., Ohlsson, J., Karlsson, J., "Fault Injection into VHDL Models: The MEFISTO Tool", Predictably Dependable Computing Systems, (Eds. Randell, B., Laprie, J.-C., Kopetz, H., Littlewood, B.), Springer-Verlag, pp. 329-346, 1995.

[Jia *et al.* 2011] Jia, Y., Harman, M, "An Analysis and Survey of the Development of Mutation Testing", IEEE Trans. on Software Engineering, vol. 37, no. 5, Sep./Oct. 2011.

[Johnson 1989] Jonhson, B. W., Design and Analysis of Fault-Tolerant Digital Systems, Addison Wesley, 1989.

[Kalakech *et al.* 2004] Kalakech, A., Kanoun, K., Crouzet, Y., Arlat, J., "Benchmarking The Dependability of Windows NT4, 2000 and XP", Proc. Int. Conf on De-pendable Systems and Networks, DSN 2004, Florence, Italy, IEEE CS Press, 2004.

[Kalyanakrishnam *et al.* 1999] Kalyanakrishnam, M., Kalbarczyk, Z., and Iyer, R., "Failure Data Analysis of a LAN of Windows NT Based Computers," Proc. Symp. Reliable Distributed Database Systems (SRDS-18), pp. 178-187, 1999.

[Kanawati *et al.* 1995] Kanawati, G., Kanawati, N., Abraham, J., "FERRARI: A Flexible Software-Based Fault and Error Injection System", IEEE Trans. Computers, Vol. 44, Nº 2, Feb. 1995.

[Kanoun *et al.* 1996] Kanoun, K., Borrel, M., "Dependability of Fault-Tolerant Systems - Explicit Modeling of the Interactions between Hardware and Software Components", IEEE Int. Computer Performance & Dependability Symposium, IPDS'96, pp. 252-261, 1996.

[Kanoun *et al.* 1997] Kanoun, K., Kaâniche, M., Laprie, J.-C., "Qualitative and Quantitative Reliability Assessment", IEEE Software, Vol. 14, Nº 2, pp. 77-87, March 1997.

[Kanoun *et al.* 2001] Kanoun, K., Arlat, J., Costa, D., Cin, M.D., Gil, P., Laprie, J.-C., Madeira, H., and Suri, N., "DBench: Dependability Benchmarking," Proc. Supplement of the IEEE/IFIP Int'l Conf. Dependable Systems and Networks, DSN 2001, 2001.

[Kanoun *et al.* 2002] Kanoun, K, Madeira, H., Arlar, J., "A Framework for Dependability Benchmarking", DSN Workshop on Dependability Benchmarking, jointly organized with DSN-2002, June 2002.

[Kanoun *et al.* 2005] Kanoun, K., Crouzet, Y., Kalakech, A., Rugina, A.-E., Rumeau, P., "Benchmarking the Dependability of Windows and Linux using PostMark Workloads", Proc. International Symposium on Fault-Tolerant Computing, pp. 11-20, IEEE Computer Society, 2005.

[Kanoun *et al.* 2006] Kanoun, K., Crouzet, Y., "Dependability Benchmarks for Operating Systems", International Journal of Performance Engineering, 2(3), pp. 275-287, 2006.

[Kanoun *et al.* 2008] Kanoun, K., Spainhower, L., Dependability Benchmarking for Computer Systems, Wiley-IEEE Computer Society Press, 2008.

[Kao *et al.* 1993] Kao, W., Iyer, R., Tand, D., "FINE: A Fault Injection and Monitoring Environment for Tracing UNIX System Behavior Under Faults", IEEE Trans. Software Eng., Vol. 19, No. 11, pp. 125-136, Nov. 1993.

[Kao *et al.* 1994] Kao, W., Iyer, R., "DEFINE: A Distributed Fault Injection and Monitoring Environment", Proc. Workshop Fault-Tolerant Parallel and Distributed Systems, June 1994.

[Katcher 1997] Katcher, J., "PostMark: A New File System Benchmark", Technical Report TR-3022, Network Appliance Inc., October 1997.

[Kerrisk 2010] Kerrisk, M., The Linux Programming Interface: A Linux and UNIX System Programming Handbook, No Starch Press; Oct. 2010.

[King *et al.* 1991] King, K., Offutt, A., "A Fortran Language System for Mutation-Based Software Testing", Software - Practice and Experience, vol. 21, no. 7, pp. 685-718, July 1991.

[Knight 2002] Knight, J.C., "Safety Critical Systems: Challenges and Directions", Proc. 24th International Conference on Software Engineering, pp. 547-550, 2002.

[Koopman *et al.* 1999a] Koopman, P., Madeira, H., "Dependability Benchmarking & Prediction: A Grand Challenge Technology Problem", 1st IEEE Int. Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems, Phoenix, Arizona, USA, November 30, 1999.

[Koopman *et al.* 1999b] Koopman, P., DeVale, J., "Comparing the Robustness of POSIX Operating Systems", 29th Intl. Symp. on Fault-Tolerant Computing, pp. 30-37, 1999.

[Koren *el al.* 2007] Koren, I., Krishna, C. M., Fault Tolerant Systems, Morgan Kaufmann, 2007.

[Krebs 2008] Krebs, B., "Cyber Incident Blamed for Nuclear Power Plant Shutdown", Washington Post, June 5, 2008.

[Laprie 1985] Laprie, J.-C., "Dependable Computing and Fault Tolerance: Concepts and Terminology", Proc. 15-th IEEE Int'l Symp. on Fault-Tolerant Computing, FCTS-15, pp. 2-11, 1985.

[Laprie 1995] Laprie, J.-C., "Dependable computing: Concepts, limits, Challenges", Invited paper IEEE 25th International Symposium on Fault-Tolerant Computing, FTCS-25, Pasadena, California, USA, pp. 42–54, June 1995.

[Laprie 1998] Laprie, J.-C., "Dependability of Computer Systems: from Concepts to Limits", IFIP International Workshop on Dependable Computing and its Applications, Johannesburg, pp. 108-126, Jan. 1998.

[Lee *et al.* 1995] Lee, I., and Iyer, R.K., "Software Dependability in the Tandem GUARDIAN System," IEEE Trans. Software Eng., vol. 21, no. 5, pp. 455-467, May 1995.

[Leveson *et al.* 1995] Leveson, N., Safeware: System Safety and computers, Addison-Wesley, 1995.

[Li *et al.* 2006] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., and Zhai, C., "Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software", Proc. 1st workshop on Architectural and System Support for Improving Software Dependability, pp. 25-33, Oct 2006.

[Lightstone *et al.* 2003] Lightstone, S., Hellerstein, J., Tetzlaff, W., Janson, P., Lassettre, E., Norton, C., Rajaraman, B., and Spainhower, L., "Towards Benchmarking Autonomic Computing Maturity," Proc. First IEEE Conf. Industrial Automatics (INDIN '03), Aug. 2003.

[Lions 1996] Lions, J.L., "Ariane 5: Flight 501 Failure – Report by the Inquiry Board", ESA, July 1996.

[Love 2010] Love, R., Linux Kernel Development, Addison-Wesley Professional, 3rd ed., July 2010.

[Lyu 1995] Lyu, M., Software Fault Tolerance, John Wiley & Sons, 1995.

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

[Lyu *et al.* 2003] Lyu, M., Huang, Z., Sze, S., Cai, X., "An Empirical Study on Testing and Fault Tolerance for Software Reliability Engineering", Proc. of 14th Int. Symposium on Software Reliability Engineering 2003, ISREE 2003, pp. 119-130, Nov. 2003.

[Madeira *et al.* 2000] Madeira, H., Costa, D., Vieira, M., "On the Emulation of Software Faults by Software Fault Injection", Proc. Int. Conference on Dependable Systems and Networks, NY, USA, pp. 417-426, 2000.

[Madeira *et al.* 2001] Madeira, H., Koopman, P., "Dependability Benchmarking: making choices in an n-dimensional problem space", Proc. of the First Workshop on Evaluating and Architecting System dependabilitY, EASY'01, July 2001.

[Madeira *et al.* 2002] Madeira, H., Kanoun, K., Arlat, J., Costa, D., Crouzet, Y., Cin, M. D., Gil, P., Suri, N., "Towards a Framework for

Dependability Benchmarking", Proc. of the 4th European Dependable Computing Conference, EDCC 2002, Oct. 2002.

[Madeira *et al.* 2003] Madeira, H., Durães, J., Vieira, M., "Emulation of Software Faults: Representativeness and Usefulness", Proc. of First Latin-American Symposium on Dependable Computing, LADC 2003, pp. 137-159, Oct. 2003.

[Masters *et al.* 2012] Masters, B., Moore, E., Pickard, J., "The upgrade that downed Royal Bank of Scotland", Financial Times, June 25, 2012.

[Mathur 1991] Mathur, A.P., "Performance, Effectiveness, and Reliability Issues in Software Testing", Proc. of the 15th Annual International Computer Software and Applications Conf., pp. 604-605, Sept. 1991.

[Mauerer 2008] Mauerer, W., Professional Linux Kernel Architecture, Wrox, Oct. 2008.

[Mauro *et al.* 2004] Mauro, J., Zhu, J., and Pramanick, I., "The System Recovery Benchmark", Proc of the 2004 Pacific Rim Int. Symposium on Dependable Computing (PRDC'04), Papeete, Tahiti, 2004.

[Maxwell 2002] Maxwell, S., Linux Core Kernel Commentary: In-Depth Code Annotation, 2nd Ed., Coriolis, 2002.

[McCabe 1976] McCabe, T., "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320, November 1976.

[Moraes *et al.* 2004] Moraes, R., Martins, E., "An Architecture-based Strategy for Interface Fault Injection", in Workshop on Architecting Dependable Systems, IEEE/IFIP Int. Conf. on Dependable Systems and Networks, DSN 2004, Italy, 2004.

[Moraes *et al.* 2005a] Moraes, R., Martins, E., Mendes, N., "Fault Injection Approach based on Dependence Analysis", Proc. 1st Workshop on Testing and Quality Assurance for Component-Based Systems, TQACBS 2005, pp. 181-188, July 2005.

[Moraes *et al.* 2005b] Moraes, R., Martins, E., Poleti, E., Mendes, N., "Using Stratified Sampling for fault injection", Proc. 2nd Latin-American Symp., LADC 2005, Salvador, Brazil, pp. 9-19, October 2005.

[Moraes *et al.* 2006a] Moraes, R., Durães, J., Martins, E., Madeira, H., "A field data study on the use of software metrics to define representative fault distribution", Proc. IEEE/IFIP Int. Conf. on Dependable Systems and Networks, DSN 2006, Workshop on Empirical Evaluation of Dependability and Security (WEEDS), Philadelphia, USA, June, 2006.

[Moraes *et al.* 2006b] Moraes, R., Barbosa, R., Duraes, J., Mendes, N., Martins, E., Madeira, H., "Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent?", Proc. 6th European Dependable Computing Conf., pp. 53-64, Oct. 2006.

[Moraes *et al.* 2007] Moraes, R., Durães, J., Barbosa, R., Martins, E., Madeira, H., "Experimental risk assessment and comparison using software fault injection", Proc. 37th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks, Dependable Computing and Communications Symp., DCCS, Edinburgh, UK, June, 2007.

[Moreira *et al.* 2003] Moreira, F., Maia, R., Costa, D., Duro, N., Rodríguez-Dapena, P., Hjortnaes, K., "Static and Dynamic Verification of Critical Software for Space Applications", Data Systems In Aerospace, DASIA 2003, 2003.

[Mresa *et al.* 1999] Mresa, E.S., Bottaci, L., "Efficiency of mutation operators and selective mutation strategies: An empirical study", Software Testing, Verification and Reliability, vol. 9, no. 4, pp. 205-232, Dec. 1999.

[Musa 1996] Musa, J., Software Reliability Engineering, McGraw-Hill, 1996.

[MySQL] MySQL Market Share, http://www.mysql.com/why-mysql/marketshare/, 2012.

[Namin *et al.* 2006] Namin, A.S., Andrews, H., "Finding Sufficient Mutation Operators via Variable Reduction", p. 5, Nov. 2006.

[Namin *et al.* 2007] Namin, A.S., Andrews, H., "On Sufficiency of Mutants", Proc. Second Workshop on Mutation Analysis, 29th Int. Conference on Software Engineering, ICSE 2007, pp. 73-74, May 2007.

[Namin *et al.* 2008] Namin, A.S., Andrews, H., Murdoch, D., "Sufficient Mutation Operators for Measuring Test Effectiveness", Int. Conference on Software Engineering, ICSE 2008, pp. 351-360, May 2008.

[Nasa 2004] NASA Software Safety Guidebook, Nasa Technical Standard, NASA-GB-8719.13, March 2004.

[Natella *et al.* 2013] Natella, R., Cotroneo, D., Durães, J., Madeira, H., , "On Fault Representativeness of Software Fault Injection", IEEE Transactions on Software Engineering, vol. 39, no. 1, pp. 80-96, Jan. 2013.

[Ng *et al.* 1996] Ng, W., Aycock, C., Rajamani, G., Chen, P., "Comparing Disk and Memory's Resistance to Operating System Crashes", Proc. 7th IEEE Symp. on Software Reliability Engineering, ISSRE 96, New York, USA, October, 1996.

[Ng *et al.* 1999] Ng, W., Chen, P., "Systematic improvement of the Fault Tolerance in the RIO file cache", Proc. 29th IEEE Fault Tolerant Computing Symp., FCTS-29, Madison, USA, 1999.

[Ng *et al.* 2001] Ng, W., Chen, P., "The Design and Verification of the Rio File Cache", IEEE Trans. on Computers, vol. 50, no. 4, April 2001.

[Oman *et al.* 1992] Oman, P., Hagemeister, J., "Metrics for Assessing a Software System's Maintainability," Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, pp. 337-344, 1992.

[Offutt *et al.* 1993] Offutt, A.J., Rothermel, G., Zapf, C., "An Experimental Evaluation of Selective Mutation", Proc. of the 15th Int. Conf. on Software Engineering, pp. 100-107, May 1993.

[Offutt *et al.* 1996] Offutt, A.J., Rothermel, G., "An Experimental Evaluation of Selective Mutat ion", ACM Trans. Software Engineering and Methodology, vol. 5, no. 2, pp. 99-118, Apr. 1996.

[Ozone] Ozone – Object Oriented Database Management System, http://www.ozone-db.org/, 2004.

[Podgurski *et al.* 1993] Podgurski, A., Yang, C., Masri, W., "Partition Testing, Stratified Sampling and Cluster Analysis", Proc. Of 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, USA, Los Angeles, pp. 169-181, 1993.

[PostgreSQL 2012] Worldwide Customers by Application Type/Workload, EntrepriseDB PostgreSQL, http://www.enterprisedb.com/customer-success/customers-by-application-workload, 2012.

[Powell *et al.* 1995] Powell , D., Martins, E., Arlat, J., Crouzet, Y., "Estimators for Fault Tolerance Coverage Evaluation", IEEE Trans. on Computers, Vol. 44, No. 2, pp. 261-274, Feb. 1995.

[Rela *et al.* 1996] Rela, M.Z., Madeira, H., Silva, J.G., "Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks", Proc. of the 1996 Symposium on Fault-Tolerant Computing, FTCS-26, pp. 394-403, June 1996.

[Rimén *et al.* 1993] Rimén, M., Ohlsson, J., Karlsson, J., Jenn, E., Arlat, J., "Design Guidelines of a VHDL based Simulation Tool for the Validation of Fault Tolerance", Proc. 1st ESPRIT Basic Research Project PDCS-2 Open Workshop, LAAS-CNRS, Toulouse, France, pp. 461-483, September 1993.

[Rodríguez *et al.* 1999] Rodríguez, M., Salles, F., Fabre, J.-C., Arlat, J., "MAFALDA: Microkernel Assessment by fault injection and design aid", 3rd European Dependable Computing Conference, EDCC-3, pp. 143-160, Sep. 1999.

[Rosenberg *et al.* 2000] Rosemberg, L., Stapko, R., Gallo, A., "Risk-based Object Oriented Testing", Proc. 13th International Software/Internet Quality Week, QW2000, San Francisco, California, USA, 2000.

[RSM] Resource Standard Metrics (RSM), Version 7.75, http://msquaredtechnologies.com/, 2011.

[RTEMS] RTEMS Real Time Operating System, http://www.rtems.org/, 2012.

[Rufino *et al.* 2007] Rufino, J., Filipe, S., Coutinho, M., Santos, S., Windsor, J., "ARINC 653 INTERFACE IN RTEMS", Proc. Data Systems in Aerospace Conference, DASIA 2007, Italy, June 2007.

[Ruiz *et al.* 2004] Ruiz, J.-C., Yuste, P., Gil, P., Lemus, L., "On Benchmarking the Dependability of Automotive Engine Control Applications", Int'l Conf. Dependable Systems and Networks (DSN'04), pp. 857-866, 2004.

[SAFE] SAFE: A Software Fault Emulation Tool, http://www.mobilab.unina.it/SFI.htm, 2012.

[Sahinoglu *et al.* 1990] Sahinoglu, M, Spafford, E.H., "A Bayes Sequential Statistical Procedure for Approving Software Products", Proc. of the IFIP Conference on Approving Software Products, ASP 1990, pp. 43-56, Sept. 1990.

[Sanches *et al.* 2011] Sanches, B., Basso, T., Moraes, R., "J-SWFIT: A Java Software Fault Injection Tool", Proc. 5th Latin American Symp. on Dependable Computing, LADC, April 2011.

[Scott 2012] Scott, J., "RBS enters fifth day of software failures", ComputerWeekly.com, June 25, 2012.

[Segall *et al.* 1988] Segall, Z., Vrsalovic, D., Siewiorek, D., Yaskin, D., Kownacki, Barton, J., Dancey, R., Robinson, A., Lin, T., "FIAT – Fault Injection Based Automated Testing Environment", Proc. 18th Int. Symp. on Fault Tolerant Computing, FCTS-18, pp. 102-107, June 1988.

[Siewiorek *et al.* 1992] Siewiorek, D.P., Swarz, R.S., Reliable Computer Systems - Design and Evaluation, Digital Press, 1992.

[Silva *et al.* 2005] Silva, J.G., Madeira, H., Dependable Computing Systems: Paradigms, Performance Issues, chapter 12: Experimental Dependability Evaluation, Wiley-Interscience, 2005.

[Skarin *et al.* 2010] Skarin, D., Barbosa, R., Karlsson, J., "GOOFI-2: A Tool for Experimental Dependability Assessment", Proc. of the 40th IEEE/IFIP Int. Conf. on Dependable Systems and Networks, DSN 2010, pp. 557-562, June/Jul. 2010.

[SPEC] SPEC – Standard Performance Evaluation Corporation, "SPECweb99 benchmark", http://www.spec.org/web99.

[Sridharan *et al.* 2010] Sridharan, M., Namin, A.S., "Prioritizing Mutation Operators based on Importance Sampling", Proc. 21st Int. Symposium on Software Reliability Engineering, pp. 378-387, Nov. 2010.

[Stafford *et al.* 1997] Stafford, J., Richardson, D., Wolf, A., "Chaining: A Software Architecture Dependence Analysis Technique", Technical Report CU-CS845-97, Department of Computer Science, University of Colorado, Sep. 1997.

[Stott *et al.* 2000] Stott, D., Floering, B., Burke, D., Kalbarczyk, Z., Iyer, R., "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors", Proc. IEEE International Computer Performance and Dependability Symposium, pp. 91-100, March 2000.

[Sullivan *et al.* 1991] Sullivan, M., and Chillarege, R., "Software defects and their impact on systems availability – A Study of field failures on operating systems", Proc. of the 21st IEEE Fault Tolerant Computing Symposium, FTCS-21, pp. 2-9, June 1991.

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

[Torres 2000] Torres-Pomales, W., "Software fault tolerance: A tutorial," NASA Technical Report NASA/TM-2000-210616, Langley Research Center, Hampton, Virginia, Oct. 2000.

[TPC] TPC – Transaction Processing Performance Evaluation Corporation, http://www.tpc.org/.

[TPCC] TPC Benchmark C (TPC-C), Transaction Processing Performance Council, http://www.tpc.org/tpcc/, 2012.

[Treanor 2012] Treanor, J., "RBS computer failure to cost bank £100m", the Guardian, August 2, 2012.

[TRHA 1993] Thames Regional Health Authority, "Report of the Inquiry into The London Ambulance Service", The Communications Directorate, South West Thames Regional Health Authority. ISBN No: 0 905133 706, February 1993.

[Trivedi *et al.* 1994] Trivedi, K.S., Haverkort, B.R., Rindos, A., Mainkar, V., "Methods and Tools for Reliability and Performability: Problems and Perspectives", 7th Intl. Conf. on Techniques and Tools for Computer Performance Evaluation, Lecture Notes in Computer Science, Vol. 794, pp. 1-24, Springer, Vienna, Austria, 1994.

[Tsai *et al.* 1996] Tsai, T., Iyer, R., "An approach towards Benchmarking of Fault Tolerant Comercial Systems", Proc. 26th Int. Symp. on Fault-Tolerant Computing, FCTS-26, pp.314-323, June 1996.

[PSOTF 2004] U.S.-Canada Power System Outage Task Force, "Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations", U.S. Energy Department, April 2004.

[Vieira *et al.* 2003] Vieira, M., and Madeira, H., "A Dependability Benchmark for OLTP Application Environments," Proc. 29th Int'l Conf. Very Large Databases, VLDB 2003, Sept. 2003.

[Vieira *et al.* 2009] Vieira, M., Madeira, H., "From Performance to Dependability Benchmarking: A Mandatory Path", TPC Technology Conference, TPCTC 2009, pp. 67-83, 2009.

[Voas *et al.* 1997a] Voas, J., McGraw, G., Kassab, L., Voas, L.,"A 'Crystal Ball' for Software Liability", IEEE Computer, pp. 29-36, June 2007.

[Voas *et al.* 1997b] Voas, J., Charron, F., McGraw, G., Miller, K., Friedman, F., "Predicting How Badly 'Good' Software can Behave", IEEE Software, Vol. 14, No. 4, pp. 73-83, Jul/Ago 1997.

[Voas *et al.* 1998] Voas, J, McGraw, G., Software Fault Injection: Inoculating Programs Against Errors, John Wiley & Sons, 1998.

[Weinstock *et al.* 1997] Weinstock, C., Gluch, D., "A Perspective on the State of Research in Fault-Tolerant Systems", Software Engineering Institute, Carnegie Mellon University, Special Report CMU/SEI-97-SR-008, June 1997.

[Weyuker 1982] Weyuker, E., "On Testing Non-Testable Programs", The Computer Journal, vol. 25, no 4, pp. 456-470, 1982.

[Weyuker 1998] Weyuker, E., "Testing Component-Based Software: A Cautionary Tale", IEEE Software, Vol. 15, No. 5, pp. 54-59, Sep./Oct. 1998.

[Wilson *et al.* 2002] Wilson, D., Murphy, B., Spainhower, L., "Progress on Defining Standardized Classes for Computing the Dependability of Computer Systems", Proc. Int'l Conf. Dependable Systems and Networks (DSN'02), pp. F1-5, 2002.

[Wong *et al.* 1995] Wong, W.E., Mathur, A.P., "Reducing the cost of mutation testing: an empirical study", Journal of Systems and Software, vol. 31, no. 3, pp. 185-196, Dec. 1995.

[Wong *et al.* 2010] Wong, W.E., Debroy, V., Surampudi, A., HyeonJeong, K., Siok, M.F., "Recent Catastrophic Accidents: Investigating How Software was Responsible", Proc. of the 2010 4th International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, pp. 14-22., June 2010.

[Xu *et al.* 2002] Xu, J., Kalbarczyk, Z., Iyer, R., "HiPerFI: A High Performance Fault Injector", Fast Abstract in Proc. IEEE Int. Conf. on Dependable Systems and Networks, June 2002.

[Zhu *et al.* 2003a] Zhu, J., Mauro, J., and Pramanick, I., "R3—A Framework for Availability Benchmarking," Proc. Int'l Conf. Dependable Systems and Networks (DSN '03), San Francisco, pp. B-86-B87, 2003.

[Zhu *et al.* 2003b] Zhu, J., Mauro, J., and Pramanick, I., "Robusteness Benchmarking for Hardware Maintenance Events", Proc. Int'l. Conf. on Dependable Systems and Networks (DSN'03), San Francisco, pp. 115-122, 2003.

[Zyl *et al.* 2006] Zyl, P., Kourie, D., Boake, A., "Comparing the Performance of Object Databases and ORM Tools", Proceedings of the 2006 annual research Conference of the South African institute of Computer Scientists and information technologists on IT research in developing countries, SAICSIT 2006, pp. 1-11, South Africa, 2006.