DBench

Dependability Benchmarking

IST-2000-25425

State of the Art

Report Version: Deliverable CF1

Report Preparation Date: 30 August 2001

Classification: Public Circulation

Contract Start Date: 1 January 2001

Duration: 36m

Project Co-ordinator: LAAS-CNRS (France)

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

Sponsor: Microsoft (UK)

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

Table of Content

State of the Art

Authored by: J. Arlat*, K. Kanoun*, H. Madeira^{++}, J. V. Busquets◆◆, T. Jarboui*, A. Johansson** and R. Lindström**

With the contribution of: S. Blanc◆◆, Y. Crouzet*, J. Durães^{++}, J.-C. Fabre*, P. Gil◆◆, M. Kaâniche*, J. J. Serrano◆◆, J. G. Silva^{++}, N. Suri** and M. Vieira^{++}

* LAAS   ** Chalmers $^{++}$ FCTUC   ◆ FAU   ◆◆ UPVLC

August 30, 2001

Abstract

The DBench (Dependability Benchmarking) project aims at formulating a conceptual framework for benchmarking the dependability of COTS and COTS-based systems. As a starting point of the project, this report surveys and comments on the state-of-the-art techniques in selected areas that we have currently identified to be pertinent for the activity of conducting the benchmarking process.

The areas covered encompass defining the context of benchmarking and its conjunction with issues of dependability. These include aspects of dependability assessment (such as analytical modelling, fault injection and field measurement), performance benchmarking and robustness benchmarking. For each area, several techniques and problems relevant to dependability benchmarking are identified, including areas requiring more research. This report compiles our current opinions on the topics and establishes the background for further DBench activities.

The survey identifies that representativeness is a key feature, since it is important both for fault injection experiments as well as for performance benchmarks.

Another insight derived from the survey is the importance of considering dependability in association with functionality and performance attributes of the systems — a stand-alone interpretation of dependability is probably not an objective property of any system. Furthermore, past experience has shown using a combined experimental and modelling approach to been successful in assessing computer system dependability. Accordingly, applying this to dependability benchmarking is also of interest, and warrants further research.

Finally, the problem of presenting relevant results to the users of a benchmark is an important facet. Performance benchmarks have typically employed a policy of presenting a few easy to understand metrics to the benchmark user. This has been a successful approach as benchmark users are not required to be experts on the subject. However, the basic issue here is ascertain the technical value of a benchmark with regards to its technical accuracy, and also for its implied utility to the user, where the user covers both system designers and end users.

1 Introduction

As the use of either commodity and/or specialized computers pervasively extends to virtually all facets of life, our consequent dependency on their correct behaviour leads to correspondingly growing impact as these computers incur failures (i.e., lack of dependable behaviour). In the past, the term “safety critical” suggested connotations of flight control, nuclear reactor type of high visibility critical systems; at present, the use of computers in cars, the use of high-performance servers driving the internet (transaction processing, e-commerce, etc.) infrastructure, all constitute “critical” systems whether for their impact on life or €-critical connotations for businesses.

Thus, the classical notion of relating the utility of a computer system based on its performance attributes now needs to be extended to involve “dependability” in the provision of performance and related functionality.

Also, given the enhanced performance and the consequent bundling of services on given computing platforms, these systems tend to get larger and more complex as they evolve. This increase in complexity poses a threat to the dependability of such systems. Thus, the awareness of dependability aspects has increased within the computer industry. The recent trends in deploying web-based services, and the pervasive use of embedded computer systems in numerous facets of life has also contributed to this awareness of the fallibility of these systems.

Traditionally, dependable computer systems were built using special hardware and software components, by designers aware of the special dependability requirements. Knowledge about the application could also be used to improve the dependability of these components. However, the cost associated with special purpose components is making this approach very unattractive and in many cases infeasible. Therefore, the trend is to use both hardware and software Commercial Off-The -Shelf components (COTS). Perhaps this has been most apparent in the hardware design of dependable computer systems, which are now largely using COTS components. For software systems the use of COTS components such as microkernels, operating systems, databases and different kinds of middleware, is increasing in an attempt to improve software quality and increase the productivity of software developers. However, very little is known about the dependability of COTS software. Thus, there is a need to find new methods for evaluating and comparing the dependability of such components. One approach that has been proposed to accomplish this is to use benchmarks for dependability assessment. Developing a framework for such a benchmark is the purpose of the DBench project.

The field of dependable systems has matured over the past few decades. This leads us to base the dependability benchmarks on the many years of experience gathered from assessing the dependability of computer systems as well as conducting performance benchmarks – both being well established and standalone fields. This report provides a documentation and survey of state of the art techniques and best practices in the above-mentioned disciplines. Apart from revealing commonality and perhaps complementary facets across them, the survey will also be important to future developmental work in the DBench project, as features from both of these disciplines will be incorporated into the final dependability benchmarking framework. Careful examination of current state of the art techniques may reveal many important research areas as well as principles that can be directly incorporated into the dependability benchmarking framework.

When an effort is made to assess the dependability of a computer system, the measures of interest are usually obtained by using analytical models in combination with fault injection experiments, consequently both of these techniques will influence the results. For this reason, both fault-injection techniques and methods for analytical modelling will be considered for the dependability benchmarking framework. The use of models may also require field measurements to determine the value of certain parameters. Results from field measurements may also be useful to evaluate the representativeness of different faults. Fault-injection will be used in the dependability benchmark to evaluate how the target system behaves in the presence of faults and give an understanding of how different categories of faults affect the system. This knowledge can be used to forecast faulty behaviour in operation as well as to evaluate and possibly improve any fault tolerance mechanisms. It is important to notice that fault-injection techniques are already being used for systems where the internal structure is unknown, which is often the case when COTS components are used. One important example of this is robustness testing that has been used to reveal deficiencies by injecting faults into the interfaces of COTS components.

The DBench project is focusing on applying benchmarking techniques to assess the dependability of both transaction processing and embedded systems. Currently, the only benchmarks available for systems in these categories are performance benchmarks. Therefore, much knowledge can be obtained by examining the history and current state of performance benchmarking. When studying the evolution of performance benchmarks the importance of the workload becomes very apparent. Having a representative workload is important for all benchmarks, from small benchmarks targeting embedded systems to large benchmarks targeting transaction processing systems. Also, how the workload is specified varies between different performance benchmarks. For some benchmarks the workload is simply the source code to an application that is to be compiled and executed on the target system, while for others the workload is a set of specifications that have to be implemented before the benchmark can be run. Which approach to choose for dependability benchmarks will have to be considered since both have their benefits and drawback. The same issue will have to be considered for the faultload.

The rest of the document is organised in the following way. Section 2 contains a survey of current techniques for dependability assessment, both model-based as well as measurement-based. Tools and techniques for robustness benchmarking are covered in Section 3. Benchmarks for embedded systems are covered in Section 4, while Section 5 is devoted to performance benchmarks for transactional systems. Included in these sections are some notes on general principles for performance benchmarks. The document is concluded with Section 6.

2 Assessment Techniques

Dependability assessment of computer architectures encompasses both model-based and measurement-based techniques. Model-based assessment includes both analytical and simulation models. The difference between the two approaches concerns mainly: i) the abstraction level usually considered for describing (modelling) the behaviour of the target system and ii) the assumptions attached to the distributions of the stochastic processes governing the parameters of the model. In this document we will primarily refer to analytical modelling. Measurement encompasses both the observation of a real-life system in operation (termed as field measurement here) and controlled experimentation where faults are deliberately injected into the target system so as to accelerate the characterization of its faulty behaviour, usually referred to as fault injection experiments.

The three techniques are complementary and are well-suited for various life-cycle phases of computer systems:

- Analytical modelling is very useful and popular to support the selection of a dependable architecture for a computer system during the design phase. Also, once the system architecture is selected, detailed modelling provides a powerful tool for the evaluation of the dependability measures (availability, reliability, etc.) of the system under consideration (in development or in operation). In the latter case, modelling needs the support of fault injection and field measurement.

- Fault injection on a prototype system is usually performed during system validation. It provides valuable information on specific behaviours of the system (or components of the system) in presence of faults. In particular, it allows understanding of the effects of faults on the target system and the assessment of the efficiency of fault-tolerance mechanisms. Fault injection is recommended for newly developed systems or for Commercial-Of-The-Shelf (COTS) components for which no (or not enough) dependability information is available from the field.

- Field measurement provides helpful support for understanding real phenomena (information on actual error/failure behaviour and on possible system bottlenecks) and for quantifying dependability measures. Even if there is no better way to understand the system dependability than by field measurement, analysis of field data can be performed only when the system is already in operation, which could be considered as too late as only little improvements can be performed at this stage in case of identification of any problem. Indeed, feedback from field data from previous systems is very helpful for the development of current products.

Each of these techniques — namely: analytical modelling, fault injection and field measurement— are addressed successively in the sequel. Finally, some aspects of the cross fertilization among these techniques are exemplified.

2.1 Analytical Modelling

Analytical modelling relies on the description of the system behaviour taking into account failure and repair of hardware and software system components and interactions between them. Measures of dependability are assessed by allocating stochastic probabilities or processes to model parameters. Analytical models have long been recognized as a determining factor for rational decision making when considering different possible architectures or maintenance policies during the design of hardware fault-tolerant systems.

The main types of models extensively used are reliability block diagrams, fault tree and state-space models. Given the ease of modelling they provide, particularly with respect to stochastic dependencies between system components, state-space models constitute the prevailing type of model for evaluating dependability measures. Markov chains are the most commonly used state-space models to model system dependability as they also allow evaluation of various measures related to dependability and performance (i.e., performability measures) based on the same model, when a reward structure [Howard 1971] is associated to them. The resulting model is referred to as Reward Markov model

To facilitate the generation of large state-space models, high-level specification languages such as Generalised Stochastic Petri Nets (GSPN) and their off-springs are generally used. In particular, the association of a reward structure leads to Generalised Stochastic Reward Petri Net (GSRPN) that can be automatically converted to Reward Markov models [Trivedi *et al.* 1994]. GSRPNs allow a compact representation of the behaviour of systems involving synchronisation, concurrency and conflict phenomena. Also, they provide means for structural verification of the model.

Evaluation can be broken down into three main closely related phases:

- The *choice of the dependability measures* to be evaluated.
- The *construction* of one (or several) model(s) describing the behaviour of the system.
- The *processing* of the model(s) to evaluate dependability measures.

In the following, the most salient trends related to the choice of dependability measures and to model construction are briefly described as numerous software packages have been devised over the last twenty years to assist model processing (e.g., see [Trivedi *et al.* 1994]). Surveys of the problems related to techniques and tools for dependability and performance evaluation can be found for example in [Reibman & Veeraraghavan 1991] and [Trivedi *et al.* 1994].

2.1.1 Dependability Measures to be Evaluated

Dependability covers a wide range of measures: reliability, availability, maintainability, safety, etc.. The measures to be evaluated greatly depend on the field of application of the computing system considered (for example: availability for a telecommunication system, reliability for a space probe, safety for the on-board control in a transportation system, etc.). To identify dependability measures, the behaviour of a computing system can be schematically depicted by taking into consideration two states of the service delivered: proper and improper. Transitions between these states are governed by the *failure* processes (from a proper service to an improper one) and the *restoration* processes (from an improper service to a proper one).

The main measures are aimed at characterizing the time of proper service delivery. Two main categories of measures are distinguished [Laprie 1995]:

- Measures that characterize the sojourn time in the state where the proper service is being delivered (before reaching the improper service state): these correspond for example to reliability and MTTF, that measure the time of proper service delivery prior to a failure.
- Measures that characterize the delivery of proper service with respect to the alternation of proper and improper services: these encompass the various forms used to measure availability (time instant, interval-of-time, or asymptotic).

Most current computing systems feature several performance levels and thus, several modes of services (proper and improper) can be distinguished. According to the viewpoints considered to evaluate dependability, there exist two main (extreme) cases for which the system features:

- several modes of proper service completion and a single mode of improper service;
- a single mode of proper service delivery and several modes of improper service.

Typical examples of systems involving several levels of performance correspond to multi-processor systems through their ability to support degraded modes of service delivery. These systems gave rise to numerous studies to define and evaluate a measure generalizing the conventional measures of dependability and associating performance measures generally referred to as *performability* [Meyer & Sanders 1993].

A particularly interesting case pertaining to the second category of systems is that of systems exhibiting two modes of improper service following failures with different levels of severity (benign and catastrophic). This allows the measures linked to the evaluation of safety of these systems to be obtained within the very same framework. Thus, safety represents the measurement of time in the safe states (proper service delivery and benign failure) prior to a catastrophic failure. A hybrid measure can be defined that measures the delivery of a proper service relative to the alternation “proper service-improper service” following a benign failure. The advantage of this measure lies in that it allows for the system availability prior to the occurrence of a catastrophic failure to be quantified, and hence, supports the assessment of the usual trade-off between reliability (or availability) and safety [Essamé *et al.* 1997]**.**

2.1.2. Model Construction

Modelling requires the knowledge of the system architecture (i.e., system composition and interactions between the various components), error detection and fault-tolerance mechanisms (if any) and maintenance policies. At the model level, the associated phenomena are represented by their occurrence rates (failure, repair, error propagation) or conditional probabilities (error detection coverage, recovery coverage, maintenance efficacy).

The main problem posed by the establishment of a Markov chain truly representative of the behaviour of a complex system is that of controlling the explosion in number of states. Several techniques have been published to address this problem; they can be grouped into two categories: “largeness avoidance” and “largeness tolerance” techniques [Trivedi *et al.* 1994].

Largeness Avoidance Techniques try to circumvent the generation of very large models. The basic idea is to construct small sub-models that can be processed in isolation. The results of the sub-models are integrated in a single overall model that is small enough to be processed. Among these techniques, we have for example: the behavioural decomposition technique [Balbo *et al.* 1988], the hybrid hierarchical modelling technique [Balakrishnam & Trivedi 1995] and the data structure technique for the Kronecker solution of GSPNs [Ciardo & Miner 1996, Ciardo & Miner 1999]. From a practical point of view and to the best of our knowledge, most of these techniques are efficient when the sub-models are loosely coupled and become hard to implement when interactions are too complex. Also, largeness avoidance by means of truncation of the least important states (i. e., states with very small probabilities) can be used to complement efficiently largeness tolerance techniques as in [Muppala *et al.* 1992].

The main objective of **Largeness Tolerance Techniques** is to master the complexity of the generation of the global system model through the use of *concise specification methods* and *automated generation* of the model. The specification consists of a set of rules allowing an easy construction of the Markov chain. These rules are based on either a) in-house formalisms or b) well-known formalisms such as GSPNs or their off-springs. Example of in-house formalisms include: Kronecker's algebra [Amoia *et al.* 1981], PERT diagrams [Sahner & Trivedi 1987]**,** production rules in METFAC [Carrasco & Figueras 1986]), SAVE language [Goyal *et al.* 1986] and FIGARO language [Bouissou 1993].

GSPNs and their off-springs appear as a general-purpose approach to the specification and construction of a complex system in a modular way. The basic idea is to generate the model of a modular system by composition of the sub-models of its components; they are referred to as *model composition techniques*. In addition to the GSNP formalism, these techniques make use of composition rules for sub-model interfacing and integration to facilitate model generation, master the complexity and preserve the formalism properties. Several model composition techniques have been published (e.g., see [Meyer & Sanders 1993], [Rojas 1996], [Kanoun & Borrel 1996], [Fota *et al.* 1999b], [Bondavalli *et al.* 1999], [Rabah & Kanoun 1999]) and numerous evaluation tools using GSPNs and their offsprings have been developed (e.g., SPNP [Ciardo *et al.* 1989], SURF-2 [Béounes *et al.* 1993], SHARPE [Sahner & Trivedi 1987], UltraSAN [Sanders *et al.* 1995] and DEEM [Bondavalli *et al.* 2000]).

GSPNs and their off-springs have been used to model real-life systems such as air traffic control systems [Fota *et al.* 1999a, Kanoun *et al.* 1999], space applications [Bondavalli *et al.* 1997] and RAID storage system [Santonja *et al.* 1996].

The evaluation of real-life systems requires the knowledge of numerical values of the model parameters that can be provided either from field data (i.e., failure and repair rates) or from controlled experiments (i.e., coverage factor, various proportions of failure modes). Sensitivity analyses allow identification of the most significant parameters to be estimated from the field or from controlled experiments.

2.2 Fault Injection

Fault injection corresponds to the artificial insertion of faults into a real[1] target computer system [Voas & McGraw 1998, Carreira *et al.* 1999]. Actually, fault injection experiments are intended to yield three benefits:

1) Understanding of the effects of real faults and thus of the related behaviour of the target system.

2) Possible enhancement and correction of the fault tolerance mechanisms included in the target system, in case of identification of design faults in these mechanisms.

[1] Depending on the phase during the development when fault injection experiments are being carried out, the target system might well be a prototype or even an early simulated version.

3) Forecasting of the faulty behaviour of the target system, in particular encompassing a measurement of the efficiency (coverage) provided by the fault tolerance mechanisms.

Initially applied to centralised systems (especially dedicated fault-tolerant computer architectures) [Avizienis & Rennels 1972], fault injection has then intensively addressed distributed computer systems [Martins *et al.* 1990, Kao & Iyer 1995, Dawson *et al.* 1996, Blough & Torii 1997, Cukier *et al.* 1999] , and also, the Internet [Labovitz *et al.* 1999]. Also, the various layers of a computer system (ranging from hardware [Martínez *et al.* 1999] to software: executive [Kao *et al.* 1993], middleware [Pan *et al.* 2001], application [Tsai & Singh 2000]) can be targeted by fault injection.

It is important to note that such an approach, based on controlled experiments, is very much suitable when non-development items (e.g., COTS components) are integrated and interact into the system being considered, which is particularly the case for the kind of operating systems addressed by the DBench project. Indeed, COTS components exhibit usually little information on their development process, on their architecture and on their actual failure behaviour, to allow for a detailed analysis to be carried out. Accordingly, robustness testing [Siewiorek *et al.* 1993] is often one important objective of fault injection experiments for such components. This will investigated in more detail in Section 3. Indeed, fault injection allows for objective data to be obtained in particular on their failure modes so as to be able to elaborate suitable architectural solutions — such as fault containment and error processing mechanisms (e.g., wrappers) — to better detect/tolerate their errors. Such insights and measurements are also useful to feed any higher-level models that may be developed for dependability verification or evaluation purpose.

Hereafter, we further exemplify the role of fault injection into the dependability validation process. Two main kind of questions arise when dealing with fault injection experiments:

1) What kinds of faults are to be injected, how to inject them and what type of target system activity is to be considered?

2) What observations, measurements are to be made and how to process them?

After presenting a brief introduction of the main attributes that characterise fault injection-based evaluation, we address these two questions in the subsequent paragraphs.

2.2.1 The Fault Injection Attributes

Fault injection experiments correspond to a form of *test sequence*, characterized by an *input* domain and an *output* domain [Arlat *et al.* 1990].

The input domain corresponds to a set of injected *faults* ***F*** and a set ***A*** that specifies the *activity* (workload) of the target system and thus, the activation profile for the injected faults.

The output domain corresponds to a set of *readouts* ***R*** that are collected to characterize the target system behaviour in the presence of faults and a set of *measures* ***M*** that are derived from the analysis and processing of the ***FAR*** sets.

Together, the ***FARM*** sets constitute the major attributes that can be used to fully characterise a fault injection test sequence. A fault injection test sequence consists of a series of *experiments*;

each experiment specifying a particular point in the ***{FxAxR}*** space. In practice, it is often the case that the definition of ***M*** has an impact on the selection of the other attributes.

During each *experiment* in a fault injection test sequence, a fault from the ***F*** set is injected that, in conjunction with the activity of the target system (***A*** set), determines an error pattern that constitutes a test input for the target system to be validated. For increased confidence in the estimates obtained, it is necessary to carry out a large number of experiments. For minimum bias in the estimation, it is further recommended to select both ***F*** and ***A*** sets by *statistical sampling* among the expected operational fault and activation domains of the target system.

An important issue to deal with concerns the combination of the ***F*** and ***A*** sets to produce error patterns. In particular, this has to do with respect to the efficiency of the tests being carried out: indeed, in the case when fault injection experiments are meant to reveal deficiencies in fault tolerance mechanisms (FTMs), one would like to minimize the number of non-significant experiments, i.e., experiments where the FTMs are not sensitised (e.g., see [Arlat *et al.* 1999]). This has also to do with the assessment of the representativeness of the fault injection experiments on the basis of error patterns actually provoked into the target system.

2.2.2 On the Input Domain

To date, two main categories of faults have been targeted by fault injection activities: physical (hardware) faults and design (software) faults. Indeed, seldom work has been devoted to devise controlled experiments addressing human-related interaction faults (being they accidental or intentional [Xu *et al.* 2001]).

As is also the case for other efforts made in dependable and fault-tolerant computing, most mature advances and techniques concerning fault injection deal with physical fault injection. Moreover, supported by the fault statistics available, transient faults have dominated this effort (over permanent faults). Dedicated tools were developed to flip bits at the pins of an IC (MESSALINE [Arlat *et al.* 1989], RIFLE [Madeira *et al.* 1994], AFIT [Martínez *et al.* 1999], just to name a few), alter the power supply or even bomb the system/chips with EMI or heavy-ions [Karlsson *et al.* 1998]. Injecting software faults is a more difficult task. Although significant work has been carried out dealing with software mutation [Thévenod-Fosse *et al.* 1995, Voas & McGraw 1998], software bug injection gave rise to much less investigations and is still largely an open research issue. Also, although they are the result of a (permanent or hard) bad design, their perceived faulty behaviour is often transient (or soft) [Gray 1986].

Nevertheless, it is important to point out that what really matters when probing the behaviour of a target system in presence of faults is much less the faults themselves than the consequences provoked/observed, via the ***A*** set attribute. Indeed, in practice, similar error patterns often originate from various distinct causes (faults). Besides it makes it possible to avoid the high costs and practical restrictions (e.g., intrusiveness, repeatability, controllability, etc.) attached to the use of physical techniques with recent hardware technologies (e.g., high density chip packing and high clock speed), such a matter of fact is one main driver for the emergence of the so-called software implemented fault-injection (SWIFI, in short) technique. Typically, SWIFI flips bits in processor register cells or in memory, thus allowing for simulating the consequences of either transient hardware faults and to some extent software faults as well, [Madeira *et al.* 2000]. In particular, several studies have shown that such simple bit flips allows for generating errors similar to those provoked by physical injection (e.g., see [Kanawati *et al.* 1995]). Recently several powerful and generic tools have been developed that support this technique; FERRARI [Kanawati *et al.* 1995], Xception [Carreira *et al.* 1998] and MAFALDA [Rodríguez *et al.* 1999] are examples of such tools. Moreover, SWIFI is generic, as it can be applied at the various layers of the target system (processor, microkernel, OS, middleware, application).

The preceding techniques assume that at least a prototype of the target system is available. An alternative, usable as early as the design phase, is to apply fault injection on a simulation model. This way, the main design choices and architectural solutions, including fault tolerance mechanisms, can be assessed in the early phases of the development process. Simulation-based fault injection is also generic as it can be applied at various levels (device, gate, RTL, PMS, network, etc.). Of course, there is an inevitable trade-off between i) the level of detail/abstraction of the simulation model and ii) the length (duration) of the simulation runs that can be carried out. Several studies have been reported that match various levels. Among these, DEPEND is a versatile environment based on a collection of generic objects (injectors, servers, links, etc.) that has been used for assessing the dependability of several large scale systems. In particular, DEPEND was recently used for the analysis of a complex RAID storage system [Kaâniche *et al.* 1998]. Furthermore, as VHDL is of widespread use, significant efforts were carried out on fault injection into VHDL simulation models both in the US (e.g., see, [DeLong *et al.* 1996]) and in Europe [Sieh *et al.* 1997, Gil *et al.* 1999], in particular within the framework of the ESPRIT PDCS and DeVa projects, both from the evaluation and verification viewpoints (e.g., see [Jenn *et al.* 1995] and [Arlat *et al.* 1999], respectively). In addition, simulation also provides useful support for the objective characterisation of fault models to be applied with the SWIFI technique (e.g., see [Yount & Siewiorek 1996]).

An alternative technique that is aimed at providing to the SWIFI technique, a “reachability” level close to what is allowed by physical techniques, still avoiding for the associated intrusiveness, is recently developing. It makes use of the testing facilities (internal scan-chain and boundary scan standard IEEE 1149.1, also known as JTAG) incorporated into recent VLSI components and hardware boards. Such testing supports are used to increase both the controllability and observability of the fault injection process. A recent study [Folkesson *et al.* 1998], showed that such a technique (also called SCIFI: Scan-Chain Implemented Fault Injection) consistently allowed to provoke errors similar to those generated when injecting into VHDL simulation models.

Concerning the ***A*** attribute, the selection of the workload is very much dependent on the objective of the fault injection experiments: fault forecasting or fault removal. In the case of *fault forecasting*, the main issue is to rate, by *evaluation*, the *efficiency* of the operational behaviour of the FTMs. This type of test thus constitutes primarily a test of the FTMs with respect to their overall behavioural specification. In practice, this means estimating the parameters that characterize the operational behaviour of the FTMs: coverage factors, dormancy, latency, etc. Accordingly, the application of the operational profile of the target system is preferred. For what concerns the *fault removal* objective, fault injection is explicitly aimed at reducing, by *verification*, the presence of FTM design and implementation faults. Since such faults can cause incorrect behaviour of the FTMs when they are faced with the faults they are intended to handle, they can be identified as *fault-tolerance deficiency* faults. From the verification viewpoint, fault injection therefore aims at revealing such faults and accordingly, to determine appropriate actions to correct the design or implementation of the FTMs. Here again, as for the ***F*** attribute, the pertinent perception of the impact of the ***A*** set, should be made through the errors patterns that are provoked into the target system.

2.2.3 On the Output Domain

As already mentioned, the outcomes of the fault injection experiments concern the assessment of the behaviour of the target system in presence of faults. Besides the identification and characterization of the failure modes, one very common outcome is the evaluation of the efficiency of fault tolerance. The main quantitative characteristic of fault tolerance is the notion of coverage [Bouricius *et al.* 1969]. In a fault-tolerant system, coverage is typically defined as the *conditional probability that, given a fault in the system, the system can tolerate it*. In practice, the notion of coverage can equally be applied to errors (in particular, to characterize the efficiency of different error processing steps: detection, recovery, etc.).

The readouts collected in ***R*** during an experiment contribute to the characterisation of the state of the target system. This is achieved by way of the assertion or not of a set of predicates that are meant to abstract the specification of the behaviour of the target system and thus of the FTMs under test. Typical examples of such predicates are: {fault_activated}, {fault_activated & error_signalled}, {error_signalled & proper service delivered}. Such predicates or their combinations define the set of vertices of a graph that models the behaviour of the target system (or of the FTMs) in the presence of faults. This graph can be either established *a priori* to describe anticipated behaviours (e.g., part of an analytical model) or obtained *a posteriori* from the processing of the ***R*** set, which is a form of model extraction from the experimental results (see Section 2.4.2). The ***R*** set can also be extended by predicates involving the outputs of the target system, as well as by the recording of execution traces so as to provide a better visibility on the faulty behaviour. Another major important dimension consists in taking into account the dynamics of the fault/error process (*latency*), leading to the definition of coverage in the form of a distribution function of the random variable characterizing the time when the monitored predicate is being asserted (e.g., see [Arlat *et al.* 1993]).

As the input space corresponding to the different "fault-activity" pairs likely to affect the target system can seldom be fully covered, tests can only be incomplete.[2] Indeed, raw experimental readouts can be typically expressed as follows:

x % of injected faults of type *f* for an activity mode *a* lead to an error detection.

Hence, statistic methods are needed to derive measurements estimate the coverage from the readouts of a fault injection test sequence.

Such an estimation raises two main concerns:

1) How to minimize the estimation bias for the tested part of the target system?

2) How to extend the range of the estimation to the non-tested part?

[2] Except for very limited assumptions, for example, injection of permanent faults on a small part of the target system.

These two questions are general in purpose and cover a wide range of fault injection experiments. However, for simplicity, we consider a target system consisting in a prototype (hardware and software) subjected to physical injection on the pins of the ICs that compose it.

With respect to item 1), the risk of bias can mainly originate from three sources:

- non-representativity of the target system activity (and therefore, of the errors induced);
- non-significant experiments related to injected faults that were not activated as errors (dormant faults);
- all injected faults do not have the same probability of occurrence.

The risk of bias linked to the activity of the target system can be partly worked out in practice by considering several modes of activation during the test sequence and by injecting faults independently with random delays at the beginning of each experiment.

Taking into account the second risk of bias is more demanding since it requires identifying whether faults have been activated as errors to remove them from the statistical studies. It is not always easy to determine why no reaction of the target system has not been observed; indeed, this may result from: i) a dormant fault, ii) a fault activated locally as an error, but not propagated, iii) a propagated error remaining latent, or iv) an error effectively masked by the fault tolerance mechanisms of the target system. Of course, this problem becomes particularly acute when physical faults are injected on a prototype. Nonetheless, techniques allow reduction of the uncertainty associated: a) the MESSALINE system implements electronic devices monitoring whether or not a fault has been activated at the injection point, b) RIFLE permits this information to be refined by taking into account the non-significant experiments resulting from the purging of erroneous data contained in the microprocessor “prefetch” registers (e.g., after executing of a branch statement). Although fault injection on a simulation model allows more accessibility to the target system and therefore, enhanced controllability and observability, even in that case, it is not always easy to differentiate in practice between an error with high latency and a masked error.

The third and last risk of bias reflects the fact that the probability of occurrence of each fault has to be taken into account in the computation of the coverage estimator. Indeed, if these probabilities could not all be considered while computing the sample of injected faults; only considering the proportion of errors detected on the number of significant experiments can lead to a biased evaluation of coverage. Such an issue, previously identified in [Rennels & Avizienis 1973], was formally analysed in [Powell *et al.* 1995] and then refined in [Cukier *et al.* 1998] by considering both frequentist and Bayesian estimates.

Finally, with respect to item 2), that is related to the use of the coverage estimator for the tested part to infer coverage for a complete system, the bounds that can be directly derived for the interval of trust of the complete coverage system are generally too loose to be of practical interest. A partial solution to this problem has been proposed in [Powell *et al.* 1995]. In particular, in the case of injection of physical faults using the “forcing” technique, it was shown that the test range can be dramatically broadened *a posteriori* by including structural information concerning the wiring diagram and by partitioning the results of experiments as a function of the equipotential lines rather than at the IC level. Another form of “fault collapsing” technique aimed at combining all faults that would lead to the same readouts into a single fault injection experiment was presented in [Smith *et al.* 1996], where error propagation is analysed in order to reduce the set of equivalent fault injection tests.

2.3 Field Measurement

Measuring a real-life system means recording naturally occurring errors and failures in the system while it is running under user workloads. Analysis of such field data can provide valuable information on actual error/failure behaviour, quantify dependability measures and identify system bottlenecks. There is a wide variety of research related to the analysis of error and failure data collected from operational computer systems. The main issues addressed are summarised hereafter.

2.3.1 Steps in Field Measurement

Field measurement involves three main steps: data collection, data validation and data processing.

Data collection consists in the definition of *what* to collect and *how* to collect the data. The kind of data to be collected is directly linked to the kind of behaviour to be analysed and to the quantitative measures to be evaluated to characterise such behaviour. There are two ways to obtain the data: online automatic logging and human manual logging. Many computer systems such as IBM mainframes, Unix/Solaris and Windows NT based systems include an event-logging software in the operating system. This software records events occurring in the various components of the system including detected errors as well as other system events such as reboots and shutdowns. The main advantage of automatic logging is its ability to record a large amount of information, in particular with respect to transient errors that cannot be achieved manually. Disadvantages are that an automatic log does not usually include information about the cause and propagation of the error or about offline diagnosis or maintenance. Also under some crash scenarios, the system may fail too quickly for any error messages to be recorded. Therefore, other sources of information are generally needed in conjunction with online logging to provide the missing data.

Data validation consists in analysing the collected data for correctness, consistency, and completeness. This consists in particular in filtering-out invalid data and in coalescing redundant or equivalent data. Usually, the collected data contains a large amount of redundant and irrelevant information, as well as incorrect or incomplete information. Such problems have been observed in several studies, e.g. see [Kaâniche *et al.* 1990, Levendel 1990, Buckley & Siewiorek 1995, Thakur & Iyer 1996]. Thus, preliminary investigation of the data must be performed to classify this information and to facilitate subsequent analyses. In automatic logs, a single fault in the system can result in many repeated errors occurring close in time. Indeed, as the effects of the fault propagate through a system, hardware and software detectors are triggered resulting in multiple error records. To ensure that the subsequent analyses will not be biased by these repeated reports, related events should be coalesced into a single event. This observation led to the development of the tuple, or data-clustering concept [Tsao & Siewiorek 1983]. Several techniques have been proposed for event tupling, e.g., see [Tsao & Siewiorek

1983, Iyer *et al.* 1986, Hansen & Siewiorek 1992]. A comparative analysis of some of these techniques is reported in [Buckley & Siewiorek 1996].

Once invalid data is filtered-out and data is coalesced, the basic dependability characteristics of the target system can be identified through data processing.

Data processing consists in performing statistical analyses on the validated data to identify and analyse trends and to evaluate quantitative measures that characterise dependability. Descriptive statistics can be derived from the data to analyse the location of faults, errors and failures among system components, the severity of failures, the time to failure or time to repair distribution, the impact of the workload on the system behaviour, the coverage of error detection and recovery mechanisms, etc. Commonly used statistical measures in the analysis include frequency, percentage, probability distribution, and hazard rate function. Basic statistical techniques can be applied to estimate the mean, variance, and confidence intervals of the parameters characterising these measures (e.g., see [Kendall 1977] for comprehensive statistical methods). More sophisticated analyses can also be performed using trend tests [Kanoun & Laprie 1996] and analytical modelling (e.g., see Section 2.4.2).

2.3.2 Study of Failures from the Field

There is a large body of literature related to the analysis of failures occurring in operational computer systems. These analyses cover several aspects including:

- Investigation of the classes of errors/failures reported, their relative importance, and the correlation among errors;
- Analysis of error/failure inter-arrival times and recovery time distributions;
- Analysis and modelling of software and hardware error detection and recovery mechanisms.

Hereafter, we outline some of the results and lessons learnt from these studies. Some of these results are detailed and discussed in [Iyer & Tang 1996].

Early dependability-related experimental studies based on field data focused on the measurement, analysis and modelling of transient or intermittent faults in digital systems. In particular, the analysis of errors collected on DEC computer systems reported in [Siewiorek *et al.* 1978, McConnel *et al.* 1979] showed that transient failures occur at least an order of magnitude higher than permanent failures. These studies also found that the inter-arrival time of transient errors follows a Weibull distribution with decreasing error rate, instead of the traditional exponential distribution generally assumed for modelling permanent failures. This distribution was shown to fit the software failure data collected from an IBM operating system [Iyer & Rossetti 1985]. The predominance of transient failures has been observed also for the software. In [Gray 1986], an analysis of operational failures in Tandem computers revealed that most software faults are soft faults that can be tolerated by simply restarting the failed process with different input conditions.

The dependence of system failure behaviour on system activity has been pointed out in the early 1980s. An early study focussed at the analysis of system failures and workloads on IBM machines, showed that the average system failure rate was strongly correlated with the average workload on the system [Butner & Iyer 1980, Iyer & Rossetti 1985]. Similar results were presented in [Castillo & Siewiorek 1981], based on measurements at Carnegie-Mellon University, and in [Moran *et al.* 1990]; they are based on error data collected on VAX8600 computers.

Correlation among system component failures is another source of stochastic dependency that has a significant impact on computer system dependability. Such correlation might exist among software failures, hardware failures as well as among both. For example, in [Iyer & Velardi 1985], nearly 35% of software failures observed on the MVS/SP operating system running on an IBM 3081 machine were hardware related. Measurements on VAXclusters [Tang *et al.* 1990, Wein & Sathaye 1990] and Tandem GUARDIAN machines [Lee *et al.* 1991] found that correlated failures exist significantly in distributed systems. Most of correlated failures observed on the VAXcluster study were related to the network interconnecting the VAX machines.

Software faults, and more generally design faults, have become the major dependability bottleneck during the last fifteen years. This is confirmed by field data collected on largely deployed systems,, e.g., see [Gray 1990, Moran *et al.* 1990, Cramp *et al.* 1992, Wood 1995]. For example, the analysis of field failures in Tandem computer systems between 1985 and 1990 [Gray 1990] revealed that more than 60% of system failures reported in 1989 were due to software. Accordingly, many experimental studies focussed at the analysis of software-related errors. The analysis and modelling of software errors to provide feedback to the development process have been addressed in several papers, e.g., [Musa *et al.* 1987, Lyu 1995, Kanoun *et al.* 1997, Murphy 2000]. Several experimental studies have been published to support the analysis of software error characteristics and the modelling of the impact of software failures on dependability, e.g. [Kanoun & Sabourin 1987, Levendel 1990, Kenney & Vouk 1992, Sullivan & Chillarege 1992, Kaâniche *et al.* 1994, Chillarege *et al.* 1995]. The issues addressed in the above mentioned work include: i) categorisation of software errors; ii) monitoring of software processes and products through the use of trend tests and statistical quality control, and iii) evaluation of quantitative measures characterising the software failure intensity and time to failure using reliability growth models.

Although there is a wide variety of research that is based upon the analysis of error and failure data collected from operational computer systems, very few studies have addressed interconnected systems. Distributed and network-based computing systems are notoriously difficult environments in which to detect errors and diagnose faults. Dependability analysis of networked applications in distributed environments requires the definition of representative fault models and meaningful dependability measures that characterise the system behaviour as perceived by the users. An example is provided in [Wood 1995] where a framework for analysing and measuring availability as perceived by the users in client-server distributed environments is defined and illustrated with experimental data collected from different sources (customers, network component providers, university studies, etc.) The lack of real data collected from the monitoring of networked applications and failures is one of the reasons for the lack of published results on dependability analysis and modelling of interconnected systems. Examples of such real data can be found in [Thakur & Iyer 1996], where an environment for collecting and analysing the failures in a network of Unix workstations is presented, and in [Kalyanakrishnam *et al.* 1999], where failure data from a network of 70 Windows NT mail servers is analysed. Both studies concern measurements performed in local area network environments and are based only on event logs recorded by the hosts. In this context, it is difficult to diagnose the cause of a failed client request: it might result from a server failure, a network component failure, or simply due to network performance degradation. The use of data collected with network management tools for the monitoring of network components and systems, in addition to the events recorded by the operating system, should help improve failure and dependability analyses in networked environments, (e.g., see [Orfali *et al.* 1996, Harnedy 1998] for a presentation of network management functions, standards, and tools).

2.4 Links Between Modelling and Measurement

2.4.1 Modelling and Fault Injection

Two major components have to be considered when constructing a model, based on the distinction between processes concerning: a) the failure of a component, i.e., the occurrence of a fault in the system (and possibly, its repair), and b) the processing of the errors (and if necessary, the treatment of the faults) by means of the fault tolerance mechanisms (FTM). Thus, both types of processes have to be taken into consideration to evaluate dependability measures.

From a conceptual point of view, analytical evaluation may be looked at as a form of extreme abstraction of fault injection: faults are introduced in axiomatic models by means of failure rates, while for fault injection experiments *per se*, stimuli are applied to either simulation models (empirical) or physical models (prototypes). However, if an axiomatic model actually permits evaluation of dependability measures (reliability, availability, etc.) by explicitly taking into account fault occurrence and failure treatment processes, basically, fault injection experiments only allow *conditional measures* to be evaluated, because they focus on the fault activation and error propagation processes, and their treatment by fault tolerance mechanisms. In addition, as already pointed out, taking into account and estimating fault tolerance parameters in the axiomatic models are essential steps and must rely on the results of fault injection experiments. Accordingly, dependability assessment is an iterative approach that must closely associate analytical and experimental evaluations.

Figure 1 depicts the main steps and relations that characterize the analytical and experimental approaches based on the various types of models involved, i.e., axiomatic (left part: blocks 2, 4 and 6), empirical or physical (right part: blocks 3, 5 and 7).

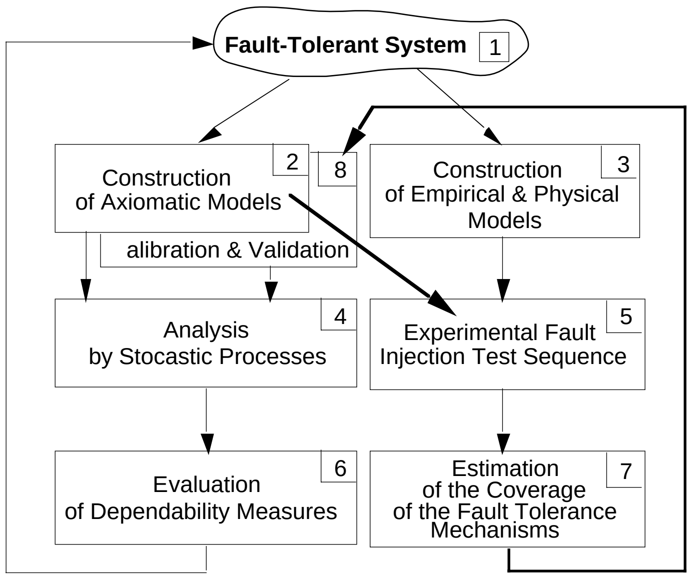

Fault-Tolerant System 1
Construction of Axiomatic Models 2
8
Construction of Empirical & Physical Models 3
alibration & Validation
Analysis by Stocastic Processes 4
Experimental Fault Injection Test Sequence 5
Evaluation of Dependability Measures 6
Estimation of the Coverage of the Fault Tolerance Mechanisms 7

Figure 1 — Relations between analytical and experimental approaches

Of course, each of these two series of steps can be carried out separately to (re)act on the target system. This concerns first, the possibility of making major architecture choices from an analysis of sensitivity of the axiomatic models to their parameters (this would correspond to a sequence of the type 1-2-4-6-1). Second, the impact of the results of the fault injection experiments on the target system, in particular aimed at removing deficiencies in the fault tolerance mechanisms could lead to the inclusion of transition 7-1 following a sequence of the type 1-3-5-7. In addition to the two conventional sequences identified and materialized in the figure by thin arrows, two main interactions (thick-line arrows) are identified.

The experimental approach (sequence 1-3-5-7) allows experimental measures typical of FTMs to be obtained. Transition 2-5 depicts the impact of parameters accounting for the coverage of these mechanisms in the axiomatic models on the specification of the fault injection test sequence (particularly with respect to the types of faults to be injected and the observations to be made). At the end of this sequence, transition 7-8 symbolizes two aspects of the interaction between both approaches:

1) *Impact of the analytical approach on the experimental approach*: the use of fault occurrence processes usually described in axiomatic models is needed to ensure a) the derivation of dependability measures and b) the unbiased and accurate estimation of coverage.

2) *Impact of the experimental approach on the analytical approach*: this includes a) the calibration of the coverage parameters of the initial axiomatic models and b) the validation and possible refinement of these models [Arlat *et al.* 1993] or even the partial extraction of models starting from the records of fault injection experiments [Choi & Iyer 1992, Kalyanakrishnam *et al.* 1999].

Then, using the stochastic process theory to solve the models permits the evaluation of the dependability measures (see Section 2.1). Then, in case the target dependability requirements are not met, the system architecture might be modified accordingly (sequence 8-4-6-1).

Although such a necessary complementarity is well established and widely recognized, at least at the conceptual level, these two approaches have seldom been used in the literature to evaluate real systems (nonetheless, see for example the SIFT and FTMP systems in [Siewiorek & Swarz 1992]. Such an evaluation approach combining analytical evaluation and fault injection has been — along with the protocol verification activity, the focal point of the validation of the fault tolerant distributed architecture in the ESPRIT Delta-4 Project [Kanoun *et al.* 1991].

In the subsequent paragraphs, we provide examples of the two main facets of links that exists between analytical modelling and experimental evaluation. The first one exemplifies how simple analytical models can be derived from experimental measurements, while the second one shows how experimental measurements can be integrated into analytical models.

2.4.1.1 Using Experimental Measurements to Derive Analytical Models

We provide here a simple example of exploitation of the measurements obtained during a fault injection campaign in order to derive a model describing the behaviour of the target system in presence of faults.

Figure 2 gives a typical example of such an experimental graph. This graph was obtained when submitting an instance of the distributed fault-tolerant (hardware and software) architecture Delta-4 to physical transient faults injected on IC pins with the MESSALINE tool [Arlat *et al.* 1990].

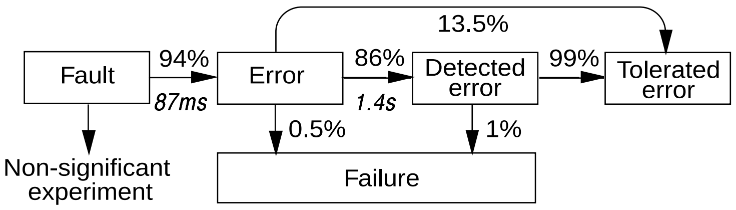


Figure 2 — Typical experimental graph

The percentages indicate the coverage values associated to *predicates* characterising the behaviour of the system in presence of faults (i.e., when `Fault` has been asserted): `Error` (activation of the injected fault), `Detected error` (the error is detected by the hardware mechanisms, and thus the faulted node is self-extracted), `Tolerated error` (the error is tolerated by the inter-node communication protocol), `Failure` (the error cannot be tolerated[3], being it detected or not). For example, 94% of the injected faults were actually activated as errors[4] (which reveals a very high testing power for pin-level fault injection); indeed, only 6% of the injections result in non-significant experiments. The associated time measurement (87 ms) defines the average *fault dormancy*. The graph shows also that 86% of these errors were detected and the associated time measurement (1.4 s) characterizes the average *latency for*

[3] Besides the invalidation of the properties (atomicity, order, etc.) to which the internode atomic multicast protocol (AMp) should comply [Powell 1994], non tolerated errors also encompasses the cases when the non-faulted nodes from the group membership are unexpectedly extracted.

[4] This information is obtained by means of current sensors attached to each injection device connected to the faulted pins.

`error detection`. One main feature of this graph concerns the inclusion of a transition that might have been omitted in an *a priori* model of system behaviour. This transition corresponds to the 13.5% of errors that were not detected, but still were tolerated at system level. Such a transition depicts a form of "extra-coverage" that results from the intrinsic resiliency of any real design due to non-deliberate redundancies — in that case, of the Delta-4 communication protocol (AMp). These experiments complemented the other efforts (formal verification and analytical modelling) used to validate the architectural designs [Kanoun *et al.* 1991].

The graph provides a high-level form of behavioural model exploiting the results obtained from fault injection experiments. Similar graphs were also exhibited in related studies (e.g., see [Chillarege & Bowen 1989]). More extensive analyses of field data can also lead to derive more detailed behavioural models. This issue will be addressed further in Section 2.4.2. It is also worth noting that similar transition models for error propagation derived from fault injection experiments were reported in [Choi & Iyer 1992] and [Fabre *et al.* 1999]. More recently, another study has proposed a conceptual framework for analysing the error propagation in modular software systems to derive the error *permeability* measure [Hiller *et al.* 2001].

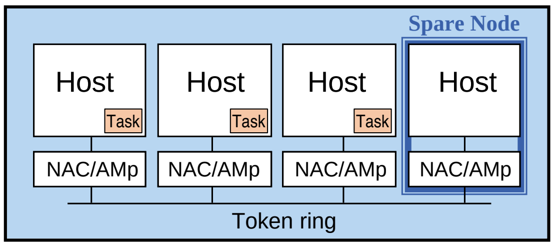
2.4.1.2 Using Experimental Measurements in Analytical Models

In order to illustrate how experimental measurements can be integrated in a modelling framework for the derivation of dependability measures, we refer again to the work carried out in the Delta-4 project (see also [Arlat *et al.* 1993]).

Figure 3 summarizes the main aspects of this work. In particular, it emphasizes transition 7-8 depicted in the diagram of Figure 1.

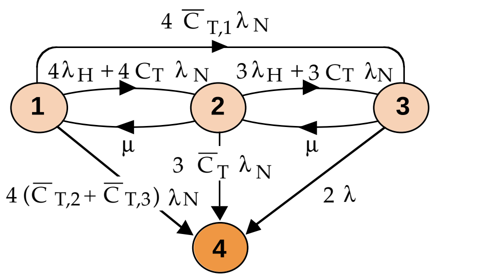
As depicted by Figure 3-a, let us consider a target architecture featuring 4 nodes each made up of a COTS host computer and of a custom network connection device — identified as he network attachment controller (NAC) — embodying error detection and confinement mechanisms, and also supporting the execution of the atomic multicast protocol (AMp).[5] Such an architecture may for example correspond to the case of a dependability requirement for triplex task execution with a 4th node acting as a "spare" in order to tolerate 2 consecutive faults. During the study, two distinct versions of the NAC hardware architecture were considered, namely: i) a standard NAC architecture with only limited self-checking capabilities (Std NAC) and ii) a NAC featuring enhanced self-checking capabilities provided by a duplex architecture (Duplex NAC). The fault injection experiments that were carried out — in particular on the Std NAC (featuring a low coverage) — had a significant impact in the debugging of the AMp software and several versions and releases of the AMp software (denoted AMp Vx.y) were therefore tested.

[5] This is actually the architecture of the target system used for the fault injection test sequence already reported (see Figure 2).



- NAC: Network Attachment Controller
  - Standard architecture (limited error detection capability)
  - Duplex architecture (enhanced error detection capability)
- AMp: Atomic Multicast protocol
  - Several successive versions and releases

a) Target architecture


$4\,\overline{C}_{T,1}\lambda_N$
$4\lambda_H + 4\,C_T\,\lambda_N$
$3\lambda_H + 3\,C_T\,\lambda_N$
1
2
3
μ
μ
$3\,\overline{C}_T\,\lambda_N$
$4\,(\overline{C}_{T,2} + \overline{C}_{T,3})\,\lambda_N$
$2\,\lambda$
4

Parameters:

λ:failure rate of one node (Host + NAC): $\lambda = \lambda_H + \lambda_N$

$\lambda_H = h \times \lambda$ and $\lambda_N = (1\text{-}h) \times \lambda = \overline{h} \times \lambda$

μ: repair rate of one node

$\overline{C}_T$: coverage of predicate T; $\overline{C}_T = 1 - C_T = \sum_{i=1}^{3} \overline{C}_{T,i}$

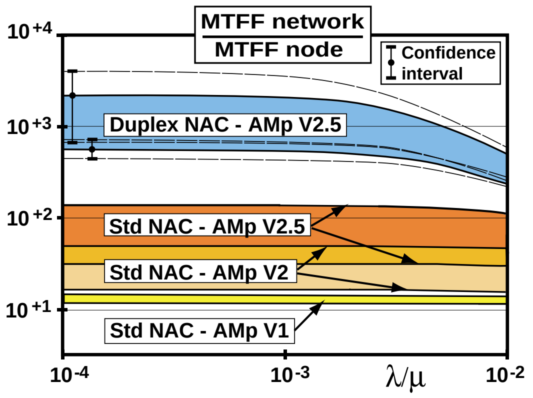
$\overline{C}_{T,i}$: non-confinement of multiplicity i, $i = 1,\dots,3$

States:
1: All 4 nodes are operational
2: A single node (the faulty node) is (self-)extracted
3: Exactly 2 nodes are extracted
4: Failure state: more than 2 nodes are extracted

b) Markov model

Target System	C_T	$\overline{C}_{T,1}$	$\overline{C}_{T,2}$	$\overline{C}_{T,3}$
NAC Std– AMp V 1	79.08%	2.32%	11.77%	6.83%
NAC Std – AMp V 2	85.02%	8.73%	2.80%	3.45%
NAC Std – AMp V 2.3	90.32%	7.79%	1.05%	0.84%
Upper Coverage Limit	*99.70%*	*0.51%*	*0.07%*	*0.26%*
NAC Duplex– AMp V 2.5	99.55%	0.32%	0.00%	0.12%
Lower Coverage Limit	*96.92%*	*0.21%*	*0.00%*	*0.06%*

c) Experimental results



d) MTFF improvement (h = 90%)

Figure 3 — Example of link between modelling and experimental results (The Delta-4 architecture)

Figure 3-b shows the Markov model that describes the behaviour of such a system. It is considered that the host computer corresponds to a fraction **h** of the failure rate associated to a node, the remaining (**1-h**) thus characterizing the NAC. In the model, parameter C_T accounts only for the coverage associated to the tolerance predicate of the NACs (`Tolerated error` on Figure 2).[6] As the experiments clearly revealed cases of non-confinement of the errors (i.e., extraction of non faulted nodes as the result of the extraction of the faulted node within the active triplex), the model also includes parameters $C_{T,i}$ accounting for the multiplicity of such undesired extractions. Indeed, such multiplicities have different consequences on the dependability behaviour of the target system being considered. For some application domains, the fault tolerance strategy depicted by the model (for which it is assumed that it is possible to tolerate up to two simultaneous extractions) might be felt as somewhat *optimistic*. Accordingly,

[6] Indeed as a high coverage majority voting decision is applied among the applications running on the host computers, the coverage of the faults in the host computers can be considered as perfect.

it is also interesting to consider another extreme (pessimistic) case for which any multiple extraction results in the failure of the network. This can be simply considered by the shift of the rate associated to transition 1-3 with transition 1-4 on the model of Figure 3-c. These two respective can thus be interpreted as *upper* and *lower* bounds on dependability improvement procured by the redundant architecture. As the sub-model of the "up states" (1-3) is strongly connected, approximations of the failure rates and MTFF can be easily obtained (e.g., see [Pagès & Gondran 1986]).

Finally, Figure 3-d compares the ranges of variations observed on the dependability gain measure for the configurations considered. The curves plotted identify the normalized MTFF improvement[7] as a function of the ratio of the failure (λ) and repair (μ) rates, for both the optimistic (upper bound) and pessimistic (lower) fault tolerance, thus identifying areas of variation of the improvement. Note that the areas associated with configurations Std NAC - AMp V2 and Std NAC - AMp V2.3 overlap.

The best *nominal upper and lower bounds*, obtained for configuration Duplex NAC - AMp V 2.5, indicate an MTFF improvement factor of 2000 and 500, respectively. However, the limits shown for each bound indicate *how the uncertainty in the estimation of the coverage may affect these dependability predictions*. As could be expected, the influence is stronger for the upper bound; the *lower/upper confidence limits* are respectively *800/4000* for the *upper bound* and *400/800* for the *lower bound*. This shows that, even in the most conservative case, the Delta-4 architecture still provides a substantial dependability improvement.

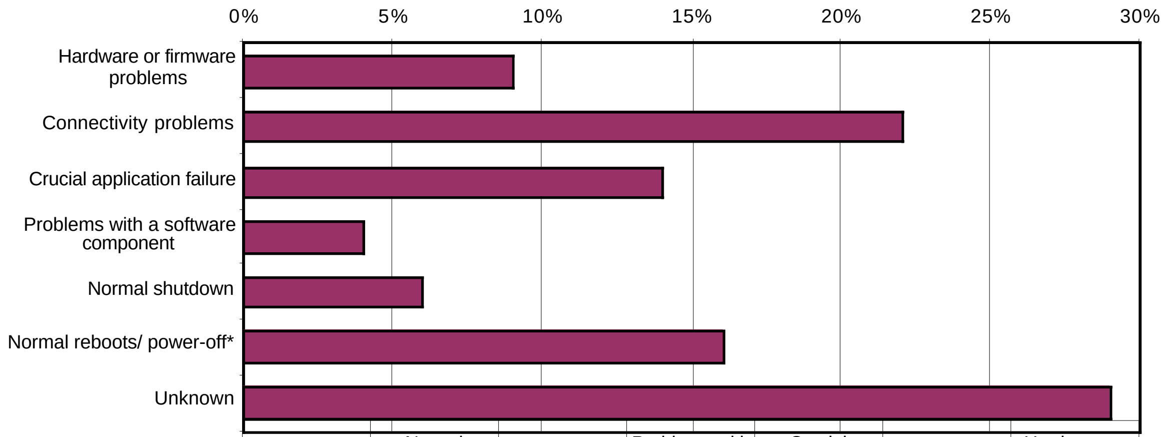
2.4.2 Modelling and Field Measurement

We focus here on the already mentioned study [Kalyanakrishnam *et al.* 1999] that was aimed at analysing failure data for a LAN of about 70 Windows NT e-mail servers. The data was obtained from event logs collected over a six-month period from the mail routing network of a commercial organisation. This work is interesting from the point of view of the processing of the experimental data to derive stochastic behavioural models of the target system.

The failure data analysis carried out on the LAN Windows NT machines focused on the analysis of the causes of the reboots and the failure behaviour of the individual machines. For example, Figure 4 (adapted from [Kalyanakrishnam *et al.* 1999]) shows a classification of the 1100 reboots observed during the monitoring period.

A large proportion of the "unknown" category was observed. This uncertainty results from the fact that at the time when the error logs were collected (1997), the version of Windows NT considered did not explicitly log an event that signals the shutdown of the machine. The results also reveal a large ratio of connectivity problems; this suggests that it is likely that an error has propagated in the domain made up of the LAN. Moreover, most problems are software-related: hardware-related causes are lower than 10%. Finally, it is worth noting that almost 50% of the reboots observed are abnormal reboots.

[7] This corresponds to the ratio of the MTFF of the redundant multiple node system architecture with respect to the MTFF of a single node (i.e., $1/\lambda$).



	Unknown	Normal reboots/ power-off*	Normal shutdown	Problems with a software component	Crucial application failure	Connectivity problems	Hardware or firmware problems
Number	319	178	63	42	152	241	105
Percentage	29%	16%	6%	4%	14%	22%	9%

* The reboot was not preceded by a shutdown and no warning nor error message appeared in the event log before the reboot.

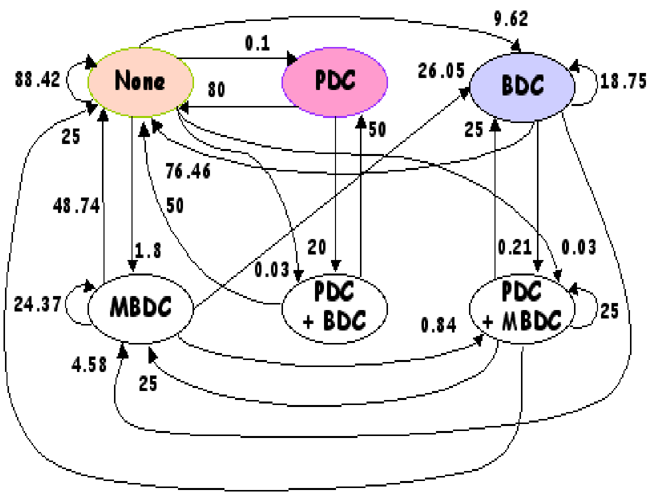
Figure 4 — Break-up of reboots into categories (1100 reboots)

These basic statistics allowed for a model to be derived that describes the behaviour of the LAN. The network features two types of machines: one machine is configured as a primary domain controller (PDC), the others operating as back-up domain controllers (BDC). The quality of service provided by the LAN is dependent upon the status (operational or not) of the PDC and BDCs. It is thus interesting to characterise the behaviour of the domain with respect to this criterion (Figure 5).

State	Reboot(s)*
None	No reboot
PDC	Primary domain controller
BDC	One back-up domain controller
MBDC	Many BDCs
PDC+BDC	The PDC and one BDC
PDC+MBDC	the PDC et many BDC

* One-hour observation window

a) Domains states



b) Domain state transitions

Figure 5 — Transition graph

Towards these ends, the observation period has been broken into 1-hour windows and the state of the domain has been determined by identifying the number of machines that rebooted during each time window; this was the basis to characterize the various states of the domain (Figure 5-a). The resulting graph is shown on Figure 5-b. The labels on the arcs indicate the various changes observed as a function of the state of the domain for the whole duration of the observation period.

Such a graph provides many useful insights. In particular, it can be seen that more than 80% of the transitions from the operational state (None) — except for the self-loop — end in the BDC state. Thus, if it is observed that such transitions result in service disruption, then the overall availability could be significantly improved just by tolerating single machine failures. Moreover, although a large ratio of transitions from the PDC state end up to the "None" state, the fact that 20% are made to the (PDC + BDC) state is likely to be an evidence of errors propagating between the machines.

2.5 Conclusion

This section presented the state-of the-art in dependability assessment based on analytical modelling, fault injection and field measurement. First, we have focused on salient trends for each technique considered alone then we have presented some examples of cross fertilisation among the three techniques

3 Robustness Benchmarks

Robustness benchmarks aim at characterizing the behaviour of a system in presence of erroneous and/or stressful input conditions. A robustness benchmark is defined as a suite of robustness tests, or stimuli in [Mukherjee & Siewiorek 1997].

Research in the domain of robustness benchmarking began to evolve in the early 90s. One of the first works in robustness benchmarking was reported in [Siewiorek *et al.* 1993]. It defines modular benchmarking. Indeed, benchmarks are constructed by regarding the system as a collection of isolated modules. Many relevant studies, [Miller *et al.* 1995, Carrette 1996, Koopman *et al.* 1997, Fabre *et al.* 1999], try to evaluate the robustness of software systems, ranging from executive kernels to simple utilities, follows-up. For example [Miller *et al.* 1995] examines the behaviour of Unix and Windows utilities when they are supplied with randomly generated input data. *Crashme* [Carrette 1996] is also a simple publicly available test of robustness for Unix and Windows systems. This program fills an array with random data and tries to execute it as if it were code. The last two studies have led to the development of two major tools (respectively, Ballista and MAFALDA) that support the conduct of robustness testing experiments.

This Section is organised in two parts. The first one details work related to robustness testing of operating systems and is focusing on a short description of the features of Ballista and MAFALDA. The second one briefly presents some current work aiming at characterizing the robustness of CORBA-based middleware.

3.1 Operating System Robustness

We focus here on two recent tools dedicated to robustness testing of operating systems (OSs), namely Ballista and MAFALDA. The former characterizes the exception handling effectiveness of software modules, while the second is mainly meant to help i) the microkernel provider in tuning its system to fit the dependability requirements set by the user, and also ii) the integrator in making architectural choices to host the mikrokernel in its design.

3.1.2 Ballista

Ballista was developed at Carnegie-Mellon University (Pittsburgh, PA, USA). The main goal of the tool is to test off-the-shelf software components against robustness problems [Koopman & DeVale 1999]. It is a combination of software testing and fault injection approaches. Tests are made using combinations of exceptional and acceptable input values of parameters of a kernel system calls. Each module under test is called once with a set of particular parameters. The parameter values are extracted randomly from a database of predefined tests. For each parameter is associated a set of values of a certain data type.

The robustness of the target OS is analysed according to the CRASH scale that distinguishes the following five failure modes:

- *Catastrophic*: The application causes a complete system crash that requires the reboot of the operating system.
- *Restart*: The application hangs and requires to be restarted.
- *Abort*: Abnormal termination of a task or a process as the result for example of a segmentation fault.
- *Silent*: No error code is returned, but one should have been returned.
- *Hindering*: Error code returned is not correct.

Catastrophic, *Restart* and *Abort* failures are detected automatically. However, *Silent* and *Hindering* failures require some manual treatment.

Initially, the testing approach supported by Ballista was developed for POSIX APIs including real time extensions. Recent work has been aimed at porting it to Windows operating systems and even various CORBA ORB implementations (see Section 3.2). In the sequel, we focus on the characterization of operating systems.

Figure 6 summarizes the results obtained with Ballista for robustness tests[8] carried out on 15 POSIX-compliant OSs [Koopman & DeVale 1999].

These experiments revealed that six OSs exhibited *Catastrophic* failures (*Crashes*), while the occurrence of *Restart* failures was observed for only two OSs. The figure also shows that all OSs featured a significant *Abort* failure rate, meaning that all they are actually sensitive to the tests. Nevertheless, *Abort* is a somewhat expected responsive behaviour for an OS subjected to exceptional tests cases; *Crashes* or *Restart*s are much more severe outcomes. Furthermore, it is worth noting that in the context of these robustness tests a *Silent* behaviour is not a guaranty of dependable behaviour. A multi-version comparison of the results obtained has been used also for identifying Silent robustness failures.

[8] In this work, a multi-version comparison of the results obtained was used for eliminating non-exceptional test cases.

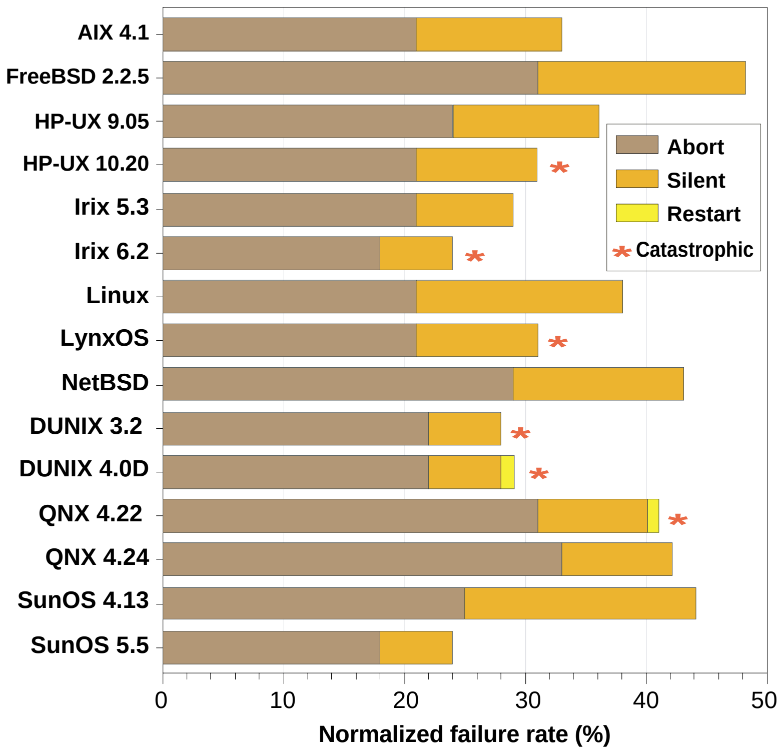


Figure 6 — Comparison of 15 POSIX operating systems

The Ballista approach was also applied to compare robustness of Windows (2000, NT, 95, 98, 98 SE and CE) and Linux (kernel 2.2.5) OSs [Shelton *et al.* 2000]. This analysis departs from the original approach, which was strictly based on a black box testing approach, and like in MAFALDA, results were detailed for the principle functionalities of the analysed operating systems. In particular, this permits a detailed interpretation of results and facilitates the definition of appropriate architectural solutions.

Figure 7 gives a synthetic view of some of the results obtained. In particular, it can be seen that while Windows CE and Windows 95, 98 and 98 SE exhibited *Crashes*, this was no longer the case for the more recent versions (which supports the claims made by Microsoft concerning the dependability improvement achieved for these versions). It is also worth pointing out that no crashes were observed during the tests carried out on Linux.

While it is irrelevant to derive strong claims concerning the respective dependability of the targeted OSs from these results alone, still interesting insights were gained from the experiments carried out. In particular, while the figure shows that, overall, the most recent versions of Windows (NT and 2000) and Linux exhibit similar behaviours, it is worth noting that Linux was found more robust on system calls, but more susceptible to *Abort* failures on C library calls compared to Windows NT.

The interest of this approach is the automatic generation of the source code of all the test cases. Ballista then executes them. The test cases consist of little programs that contain parameter declarations and the function call. This method permits detection of design errors. In fact, it can generate automatically exhaustive combinations of function parameter values.

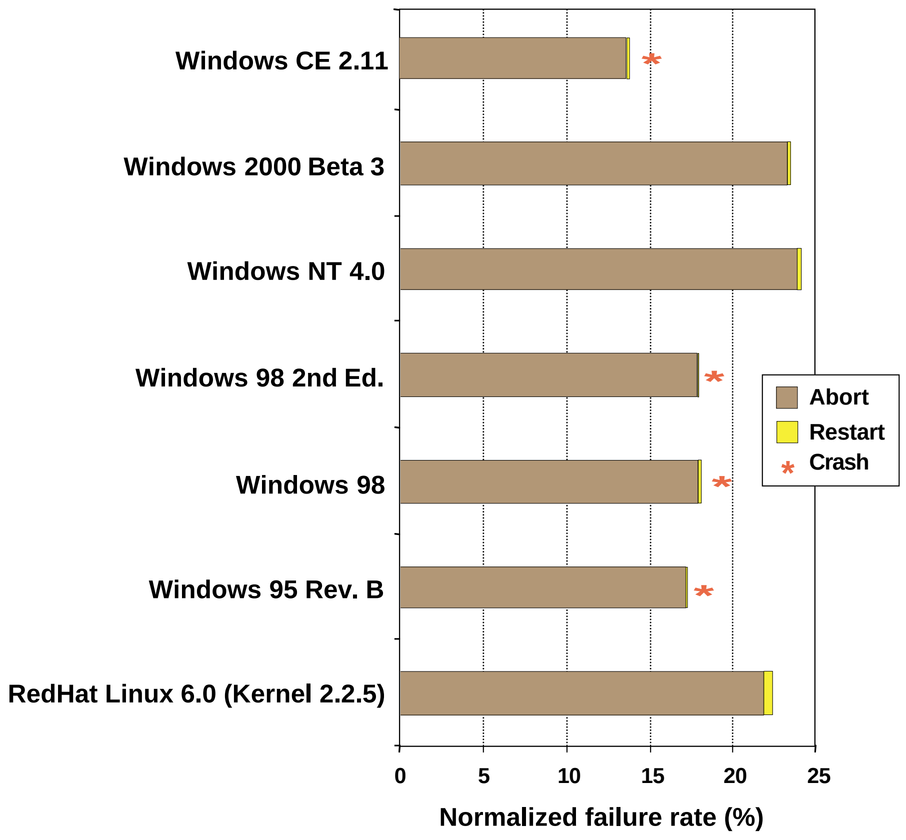


Figure 7 — Comparison of windows systems and Linux

3.1.3 MAFALDA

MAFALDA (Microkernel Assessment by Fault injection AnaLysis and Design Aid) was developed at LAAS-CNRS (Toulouse, France). It is a comprehensive tool aimed at i) characterizing the behaviour of microkernels (including COTS microkernels) in the presence of faults, and ii) facilitating the implementation of error detection mechanisms for enhancing the faulty behaviour and thus the dependability of the target microkernel. While MAFALDA supports fault injection both into the parameters of system calls and into the memory segments implementing the target microkernel, we focus here on the robustness testing capabilities offered by MAFALDA. More detailed information on MAFALDA and of the experiments conducted can be found in [Arlat *et al.* 2002]. MAFALDA runs on a test bench made up of two main entities:

- a rack of ten *Target Machines* (Intel Pentium-based PC boards) running the target microkernel,
- a *Host Machine*, (a Sun Workstation) that defines, executes, controls the experiments carried out on the target machines and finally analyses the results.

On each Target Machine, the specific workload processes are responsible for the activation of the targeted microkernel functional components (e.g., synchronization, scheduling, etc.). Two specific software modules are embedded within each target machine: the Interception and Injection modules.

The Host Machine, controls the execution and coordination of the fault injection campaigns. The parameters characterizing the campaign are stored in two input files i) the Campaign Descriptor that defines the target components and the fault set to be considered for the fault injection campaign and ii) the Workload Descriptor that contains the code for the workload processes executed during the experiments. The experiment results and measures are stored for either quick display or *a posteriori* processing.[9]

As an example, Figure 8 illustrates the types of results that were obtained when subjecting an instance of the Chorus/ClassiX microkernel (composed of basic functional components implementing basic services — such as synchronisation, memory, communication and scheduling — to a series of SWIFI experiments affecting the system call parameters.

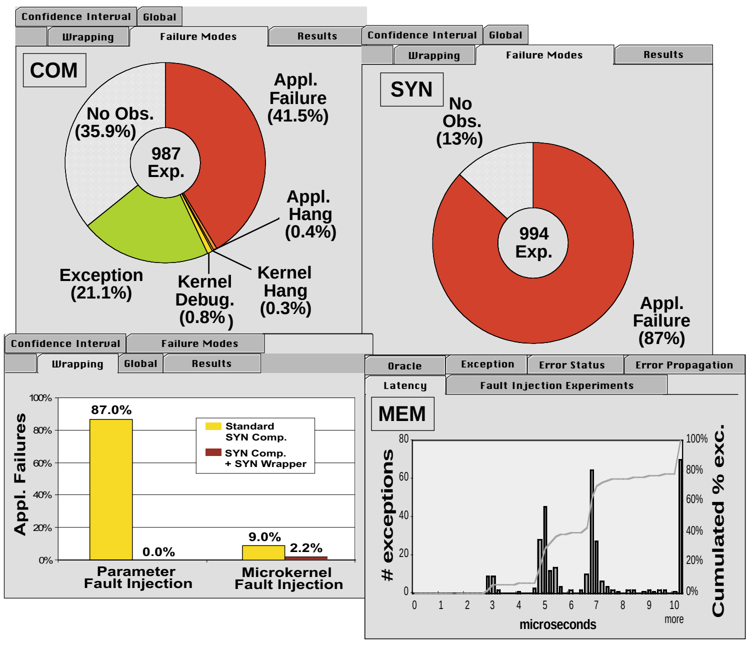


Figure 8 — Sample of results obtained with MAFALDA

The two top windows show pie diagrams concerning the behaviours observed for the communication (COM) and synchronisation (SYN) components. Regarding the COM component, about 22% of the errors were successfully detected by the microkernel error detection mechanisms ("Exception" and "Kernel Debugger", in that case), while a hang ("Kernel Hang", "Application Hang") occurred in less than 1% of the cases. Nevertheless, 40% of the errors led to an incorrect service ("Application Failure"). Finally, the "No Observation category (almost 36%) corresponds to errors that had no observable consequences although the injected faults were actually activated. The SYN component exhibited a very distinct behaviour: only application failures were observed when applying fault injection to the parameters passed to the synchronisation primitives. Moreover, the high ratio (87%) of

[9] Two additional modules called the Wrapping and Reflection modules are also provided to facilitate and support the implementation of additional on-line checks. This feature of MAFALDA is outside the scope of the issues of interest for DBench; accordingly, it will not be addressed further here. The interested reader can find more details in [Salles *et al.* 1999].

application failures is very surprising. The reason for this singular behaviour is that Chorus basic synchronisation mechanisms (essentially semaphores) do not make any verification of input parameters. This design option at the very basic microkernel layer was made on purpose in order to leave total freedom on this respect to the upper layer designer.

Due to the weakness of the SYN component, a specific wrapper (checking a basic property of semaphore-based synchronisation) was developed to improve its fault containment ability [Salles *et al.* 1999]. As shown by the histogram on the bottom left window, this simple wrapper proved highly efficient: i) it was able to eliminate all application failures in the case of corruptions of the parameter calls to the SYN component and ii) it reduced the rate of application failures from 9% to 2.2% in the case when bit flips were injected in the code segment of the SYN component. This ability to support the evaluation of various design options is of major interest for both microkernel designers (providers) and integrators (users).

Finally, the bottom right window illustrates the error latency associated to exceptions in the case of the memory (MEM) component: the distribution is characterized by three distinct modes at 3, 5 and 7 µs that concern 80% of the total number of exceptions.

As reported in [Arlat *et al.* 2002], MAFALDA was also used to analyse the LynxOS microkernel. These experiments showed a wide discrepancy of results depending on the various functional components for the Chorus and LynxOS microkernels.

A recent enhancement of the tool (MAFALDA-RT) concerns the extension of the analysis of the faulty behaviours to encompass the measurement of response times, deadline misses, etc. which are of paramount in the case of real-time systems. Such measurements are made possible thanks to the technique used for mastering of the intrusiveness related to the fault injection and monitoring processes [Rodríguez *et al.* 2001].

3.2 CORBA-based Middleware Robustness

The Ballista approach was recently adapted to test the robustness of various CORBA ORB implementations [Pan *et al.* 2001]. The target ORBs were Orbix, omniORB and VisiBroker. The failure modes were different from the original CRASH scale (see Section 3.1.2) and were adapted to better fit the CORBA context.

In the reported work, the number of functions tested varies between 17 and 23. These functions are part of the client-side interface exposed by an ORB.

As an example of the results obtained, Figure 9 shows an increase in average percentage of robustness failures from an older version of a product to a new version.

Moreover, while for Orbix 2000 the Sun Solaris version is more robust than the Linux version, for omniORB 3.0 and VisiBroker 4.0, the results are the opposite. It is worth noting that this work has only targeted client-side operations. Accordingly, interaction between a client and a server is not covered.

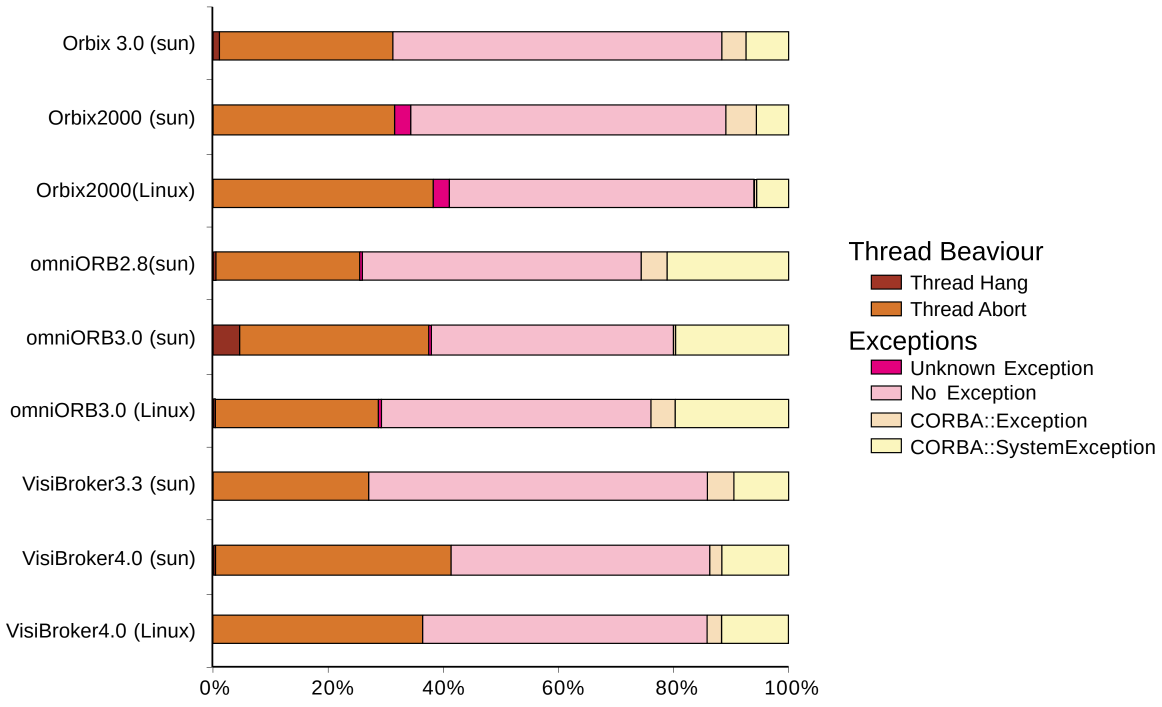


Figure 9 – Robustness test of various CORBA ORB implementations

A related study has been recently carried at LAAS-CNRS that aims at the characterization of CORBA-based middleware. The initial work presented in [Marsden & Fabre 2001] targets four implementations of the CORBA Name Service. Those provided with omniORB 2.8, ORBit 0.5.0, ORBacus 4.0.4 and the ORB bundled with version 1.3 of Sun’s Java SDK.

The characterisation of the behaviour in presence of faults is based on the results of fault injection experiments consisting of sending a corrupted request to the target (server side), and observing the resulting behaviour. The types of errors injected are single bit flips and the zeroing of two successive bytes in a message simulating transient faults in the communication subsystem as well as error propagating from remote clients. In the experiments conducted, for both fault models considered, no error propagations to the application level were observed. As an example, Figure 10 depicts the outcomes observed for the double-byte zeroing fault model.

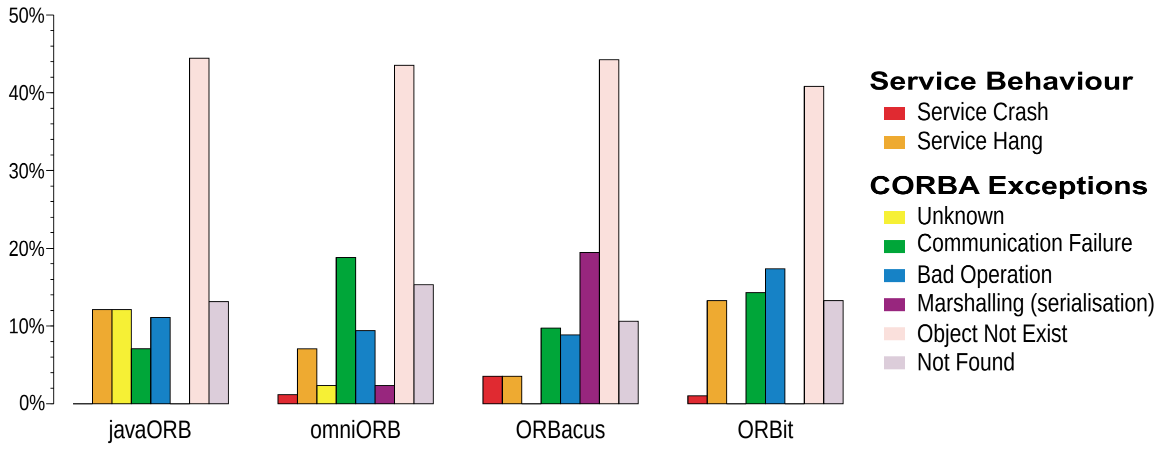


Figure 10 – Breakdown of the outcomes observed for the double-byte zeroing fault model

It is worth noting that the difference in the outcomes reported with respect to the previous study (see Figure 9) is due to distinct features of the experiments carried out i) in the target system considered (server instead of client, in this case) and ii) in the fault models used.

The experimental results show a relatively large variability of behaviour of the target candidates in the presence of faults. They also show that the *ORBacus* and *omniORB* implementations exhibit the safest behaviour in terms of the number of service failures and in the types of exceptions reported.

From the whole set of experiments, it was also found that several outcomes are not common to the two fault models used. As an example, the appearance of a *InvalidName* exception, when using single bit flip fault model, which is not provoked by the double-zero fault model.

3.3 Conclusion

Robustness testing, where the target system (software component instance or fault-tolerant computer) is subjected to erroneous and/or stressful input conditions, is one very important facet of the characterization of behaviours in presence of faults.

Actually, because in such a form of assessment fault injection is being carried out at the input interface of the target system, a clear definition of the form of tests being applied can be obtained simply from the description of the interface, thus without requiring a deep knowledge of the architecture and implementation of the target system. This is indeed a very desirable feature when considering the characterization of COTS systems for which little structural information is usually available. Moreover, the existence of standard or *de facto* interfaces (e.g., POSIX, CORBA, etc.) should significantly contribute to the attainment of an agreement on the set of robustness tests being applied and furthermore facilitate its portability from one target system to another one.

Accordingly, such a form of test is definitely a strong candidate for being included in the development of the set of dependability benchmarks that will be investigated and developed within the framework of the DBench project.

4 Performance Benchmarks for Embedded Systems

A performance benchmark is a test that measures the performance of a computer system or a component on a well-defined task or set of tasks. The task or set of tasks is normally defined by a workload and the measures are specific of each benchmark[10]. A set of rules specifies the way the test must be conducted to achieve valid benchmark results.

There are hundreds of performance benchmarks for computer systems and components today. However, as a performance benchmark represents a standard (many times an informal standard or a *de facto* standard) the relevance of a benchmark is directly related to the number of people that adhere to that standard and use the benchmark in practice. In this view, the number of performance benchmarks that might be relevant for the DBench project is substantially reduced.

[10] Actually, each performance benchmark represents a different yardstick, and results from different benchmarks cannot be compared or related one to each other.

When considering embedded systems, difficulties arise when selecting the best components for a new product. System specifications may help in these issues, but usually they are insufficient. For example, computing speed requirements, communication-integrated drivers, on-chip peripherals, memory features, implementation and debugging tools must be carefully analysed. However, depending on the working area, real-time characteristics and dependability become highly important parameters in order to select hardware and software. Benchmarking is considered as a resort with increasing acceptance to characterise system features by generating a suitable activity based upon specifications.

Workloads used in performance studies increase complexity from synthetic to real-world applications as shown in Figure 11.

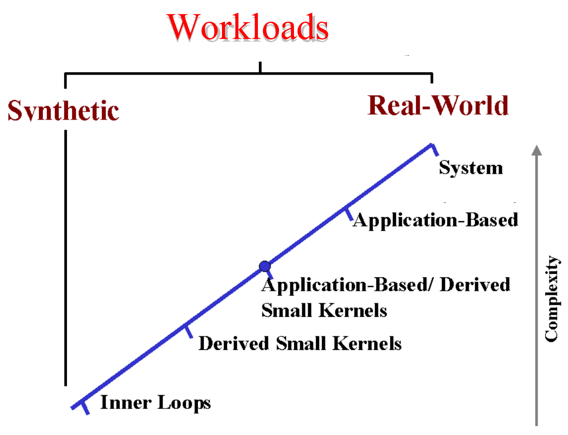


Figure 11 — Benchmark taxonomy

Synthetic workloads are artificial and manageable. Their characteristics are similar to real workloads but exhibit a higher controllability. A real workload comprises normal operations during the operational phase of the system. However, it cannot be easily repeated and requires a great amount of time for testing all possible events. If the system to be evaluated is used for a particular application area, a representative subset of functions for that application may be used. Such workloads are close to real-world applications and are generally described in terms of functions to be performed and make use of almost all resources in the system, including processors, I/O devices, networks, etc.

We find a wide variety of benchmarks devoted to embedded systems: they range i) from free software, more or less documented, ii) software developed by designers or manufacturers, specially oriented to their own products, iii) to finally software developed by specialised certification laboratories, oriented to compare different platforms. However, selecting the best workload for evaluating and validating a system depends on the expected goal for developing them.

Recent trends in the use of embedded computing systems add a new dimension to the challenge of developingsuitable benchmarks. These trends include:

- Increased use of real-time components within embedded computer systems to support voice and gesture interfaces, and to support multimedia interaction.

- Increased use of real-time computing in mass-marketed consumer electronics devices as for example electronic controller units in automotive area.
- Increased sophistication of real-time software, requiring the use of algorithms that do not necessarily execute in constant time.

"The term real-time describes any information processing activity or system that has to respond to externally generated input stimuli within a finite and specifiable amount of time" [Wells 1993]. In order to guarantee that the complete real-time system will always perform within the specified real-time constraints, developers use special software development methodologies that allow them to analyse and oversee the real-time behaviour of the system. Then, it could be desirable that a benchmark will evaluate the complete system, including the architecture, the operating system, the programming language, its compiler and whatever development and analysis integrated tool.

Usually, benchmark objectives are excessively particularised resulting in a huge range of possibilities that makes it difficult to achieve a benchmarking taxonomy. However, benchmarks can be divided according to system level. Thus, we can distinguish:

- *Microprocessor-oriented benchmarks* or benchmarks scores help designers select the right embedded microcontroller or microprocessor for their systems. They are used to assist designers in comparing embedded platforms for specific application areas.
- *OS-oriented benchmarks* are intended oriented to evaluate operating system performance .
- *Specific application-oriented benchmarks* are clearly less used for embedded systems,simply because microprocessor- and OS-oriented benchmarks already heavily rely on the application area.

The rest of this section will consider successively microprocessor-oriented benchmarks and operating system benchmarks.

4.1 Microprocessor-oriented Benchmarks

Benchmarks are presented according to the workload taxonomy presented in Figure 11.

4.1.1 Synthetic Workloads

Synthetic workloads or small real applications developed for a particular design constitute a frequent resource to test embedded systems. However, those non-portable workloads require a great amount of human effort to migrate to a particular system. Communication with the outside world helps to share those implemented workloads trying to soften their restrictive dependencies. Besides, adapting them to new tests does not always deliver a satisfactory result.

Typical examples of synthetic workloads are the Whetstone, the Dhrystone and the Tri-Dimensional Measure benchmark:

- The Whetstone is generally considered as a floating-point benchmark and is mostly representative of small engineering applications.

- The Dhrystone is a popular measure of integer performance.
- The Tri-Dimensional Measure benchmark [Rabbat 1988a, Rabbat 1988b] attempts to characterise real-time performance in terms of three important performance measures. Those are:
  - *CPU computational speed*, measured in Millions of Instruction Per Second;
  - *interrupt handling capability*, measured in Millions of Interrupts Per Second;
  - *I/O throughput*, measured in Millions of I/O transfers Per Second (Mbytes/sec).

  These three measures are not independent and should be analysed together. However, as with the other benchmarks, the Tri-Dimensional Measure benchmark lacks rigor in its description of the measurement methods and the implementation of the experimental workloads.

4.1.2 Real World-based Workloads

Consortiums, certification laboratories, private or public research departments or institutes work on developing solutions close to real world workloads. Following are some examples of those efforts.

- The National Institute of Standards and Technology (NIST) and the EDN Embedded Microprocessor Benchmark Consortium (EEMBC). The latter provides general benchmarks that are perfectly applicable to embedded systems. In 1999, EEMBC has released its first set of benchmark suites for Automotive/Industrial, Consumer, Networking, Telecommunications, and Office Automation that run on a variety of host platforms [Weiss 1999, Levy 2001]. The goal of such benchmarks is to help the designer to select an appropriate processor, controller or DSP for an application included in one of their mentioned areas.
- In 1987, the University of Michigan [Donohoe 1987] presented a Survey of Real-Time Performance Benchmarks for the Ada programming language: The Ada Embedded Systems Testbed (AEST) project at the Software Engineering Institute (SEI). It was focused on the investigation of a wide variety of issues related to software development for real-time embedded systems. The Ada language features measured by the benchmarks are subprogram calls, dynamic storage allocation, exception handling, task elaboration activation and termination, task rendez-vous, clock function resolution and overhead, and time math. The Ada runtime features measured by the benchmarks are scheduling considerations (delay statement), object deallocation and garbage collection and interrupt response time (outline only).

  The Performance Issues Working Group (PIWG) Ada Benchmarks comprise more than 130 different tests that were either collected or developed by PIWG under the auspices of the ACM Special Interest Group on Ada (SIGAda). They are grouped into three broad categories: composite benchmarks, individual timing tests and compilation tests. The composite benchmarks include a PIWG Ada version of the Whetstone and Dhrystone benchmarks. The Hennessy benchmark is a collection of well-known programming problems coded in Ada, as for example the matrix multiplication, the towers of Hanoi, the Quicksort, the Fast Fourier Transform, and so on. Besides, the prototype Ada Compiler

Evaluation Capability (ACEC) was provided with an organised suite of compiler performance tests and support software for executing the test and collecting performance statistics. The Institute constructed it for Defence Analyses for the Evaluation and Validation (E&V) team of the Ada Joint Program Office (AJPO).

- Also some specific benchmarks for comparing different platforms are developed by manufacturers. Philips' web site [PHILIPS] presents a comparative benchmark study of the CPU performance of the XA versus Motorola 68000, Intel 80C196, Philips 80C51 and Intel MCS251 carried out in 1996. Siemens' web site also includes the core of Siemens C161 [SIEMENS] in 1997. The benchmark comprises 16-by-16-bit signed multiplication, 16-by-16-bit floating point division, 24-bit variable comparison, CAN commands, linear interpolation (8*8), interrupts overhead, static routines overhead, and program overhead. However, some controversial questions remain such as the objectivity of benchmark developers, the performance to be measured and its portability to other platforms, the incentive that benchmarking offers to vendors, etc. Several manufacturers have settled this problem licensing their architectures, for example: ARM Holdings, ARC Cores, Sun Microsystems, MIPS Technologies, Motorola Cores, etc. that license the design of their CPUs and instructions set to chip makers [Turley 1999].

4.2 OS-oriented Benchmarks

As stated in [Bradley et al. 1995], current modern applications spent most of their cycles executing operating system code. Thus, it is important to measure the performance of kernel internal mechanisms.

The OS oriented benchmarks are basically synthetic workloads that exercise the operating system services. We will first address benchmarks for common OSs then we will address Real-Time OS (RTOS).

We have identified three main types of OS benchmarks:

- [Ousterhout 1990] proposes several OS benchmarks to measure system call latency, context switch time, and file system performance. The author concludes that operating system performance is not improving at the same rate as the base speed of the underlying hardware. Also, the most obvious ways to remedy this situation are to improve memory bandwidth and reduce operating system tendency to wait for disk operation to complete.

- [McVoy et Staelin 1996] introduced the LMbench micro-benchmarks package. It was developed to identify and evaluate system performance bottlenecks. It consists of a set of programs that measure operating system latency and bandwidth[11] of data movement among the processor and memory, network, file system, and disk. It is used at many sites by both end-users and system designers. LMbench has been run on AIX, BSDI, HP-UX, IRIX, Linux, FreeBSD, NetBSD, OSF/1, and Solaris. Also, part of the suite has been run on

[11] More specifically, the performance measures evaluated are: memory read/write/copy bandwidth, Inter Process Communication (related mainly to TCP and pipe) bandwidth, cached I/O bandwidth, memory read latency, context switching time, networking (connection establishment, pipe, TCP, UDP, and RPC hot potato) latency, file system latency, process creation cost, signal handling cost, system call overhead.

Windows-NT as well. The results have shown a strong correlation between memory performance and overall performance.

- Another relevant tool, detailed in [Brown et Seltzer 1997], is HBench-OS. The authors augmented the LMbench suite to increase its flexibility and precision, and to improve its methodological and statistical operation. Modification efforts were directed at making the existing benchmarks more rigorous, self-consistent, reproducible and conductive to statistical analysis. Indeed, the timing and the statistical methodologies were improved. In addition, some tests were revisited and modified. The analysis has shown that off-chip memory system design continues to influence operating system performance in a significant way.

Late 80's and early 90's is definitely a reference period on benchmarking for Real-Time Operating Systems (RTOS). Hereafter, we briefly present the main features of a few available benchmarks (the RhealStone [Kar 1989, Kar 1990], the Process Dispatch Latency Time (PDLT) benchmark [Furht 1989], the Real/Stone, the Hartstone) and of some benchmarks developed by the manufacturers:

- RhealStone consists of quantitative measurements of six critical operations that influence the performance of real-time systems. The operations are: task switching time, preemption time, interrupt latency time, semaphore shuffling time, deadlock breaking time, and datagram throughput time. A Rhealstone number is calculated as the weighted sum of the times required for implementing each of the fundamental operations.

- The Process Dispatch Latency Time (PDLT) benchmark can be defined as the length of the interval between the time when the system receives an interrupt request and the time when it begins to execute the code that is associated with the interrupt.

- The Real/Stone was properly used to global system measures. The Real/Stone is an artificial synthetic workload that was designed to simulate a real-world environment. It is a pure software benchmark that does not require any hardware test. It was based on the Vanada benchmark [Vanada 1984] and consists of three tests that measure system responsiveness, system preemptibility and system I/O throughput.

- The Hartstone benchmark [Weiderman 1989, Weiderman 1992] is a synthetic workload comprised of both periodic and sporadic tasks implemented in Ada written at Carnegie Mellon University. Rather than measuring throughput, the purpose of Hartstone evaluation is to measure the breakdown point of a real-time system. This point is defined as the point at which the computational and scheduling load causes a hard deadline to be missed. Given the well-defined set of tasks, the basic workload unit, and the interrelationships among them, neither the implementation language, nor the scheduling algorithms, nor the system synchronization primitives used matter to the Hartstone evaluation. This flexibility allows practical comparisons of systems that differ in many aspects, including hardware, operating system, and scheduling techniques. Using this benchmark, the underlying hardware details are implicitly accounted for, in that the measured system performance takes advantage of whatever cache, registers, pipeline units were available to it.

- Among *benchmarks developed by manufacturers*, we find an example in MontaVista Software Inc [MVISTA] that developed specific Linux for embedded systems: the Hard Hat Linux. It supports a variety of capabilities exclusively for embedded applications using both Intel x86/Pentium and Motorola PowerPC/PowerQUICC families. It includes a measurement-integrated tool with the operating system for measuring the interrupt blocking time as induced by the Linux kernel and device drivers. The measurement tool detects the longest blocking times at distinct points inside the Linux kernel/drivers, permits configurable number of log entries, allows user processes to view the results interactively and establishes selective range of block times for study.

It is also interesting to put emphasis on efforts providing guidelines for the development of enhanced OS benchmarks. As it is reported in [Halang 2000] the guideline VDI/VDE 3552-3553 (VDI/VDE) and, in particular, the German Standardisation Institute's DIN 19 242 standard devoted an important effort for benchmarking real-time operating systems. It defines measurable parameters of process control computers and gives programming examples. Also, the DIN 66 273 standard defines data processing, measurement and rating of data processing performance, as well as a measuring and rating method. It concentrates on visible or measurable performance quantities, for instance the response time.

The work reported in [Halang 2000] also discusses the appropriate performance criteria for measuring real-time systems. The author recommend using these criteria not in a general setting, but only in an application specific way. Criteria can be divided into three groups: qualitative binary criteria, as for example the ability to meet all deadlines, qualitative gradual criteria, as for example system dependability, and quantitative criteria, as for example the worst-case response times to occurring events. Some of those performance criteria could be also applied to non-real-time systems.

Finally, a proposal for new benchmark standard, that can be related with other areas, is suggested in [Busquets 1994] and addresses the following issues:

- The experimental workload must be similar to the actual workloads that it represents. Otherwise, the system performance measured on the experimental workload will not correlate with actual experience.

- The benchmark evaluation must be comprehensible and useful to the real-time community. A graphical representation is preferable to a single number [Ponder 1991]. It may be desirable to present the results as a function of configurable parameters, architecture and operating system parameters, and type of workload.

- The benchmark should support a wide spectrum of applicability. The benchmark must be useful to evaluate and compare architectures, compilers, operating systems, scheduling algorithms, and development methodologies.

- The benchmark should evaluate the system abilities to support both hard real-time and soft real-time applications. Static analysis techniques that guarantee compliance with real-time constraints must be recognized and rewarded by the benchmark.

- The benchmark must be well defined, easily implemented, and portable. It should consist of a carefully written definition of tasks, which use a common interface to the operating system.
- The work required to measure a particular system must be manageable.
- If vendors tune their systems in order to achieve improved benchmark performance, then it is our hope that real workloads would also benefit from the benchmark–motivated tuning.

4.3 Conclusion

In this section, we have presented a number of performance benchmarks, oriented to obtain specific measures of a microprocessor or an operating system and to a less extent to embedded systems composed of microprocessors, microcontrollers and operating systems. None of them are focused on dependability measures. However, some of them may be used in the context of the DBench project i) to measure performance degradation of the system in presence of faults and ii) as a baseline workload that exercises the system during the process of obtaining pure dependability measures.

The fact that they are many types of benchmarks suitable for embedded systems shows there is actually a strong activity in this domain that span the whole community (designers, end-users and standardisation organisations). Nevertheless, this large and highly diversified offer results in significant development expenses and also in some difficulty when one is faced to select one benchmark. A well-defined benchmark has to satisfy a clear definition of the benchmark objectives and a method to accomplish these objectives. It is also required to determine the necessary measures and how to evaluate them in order to obtain meaningful results. Also, a well-defined benchmark must satisfy some properties such as: representativeness, applicability, portability and fairness. These features and properties will be addressed further in the companion deliverable [CF2 2001], related to the preliminary definition of a dependability benchmarking framework.

5 Performance Benchmark for Transactional Systems

There are two major organisations in the performance benchmarking business[12]: SPEC (Standard Performance Evaluation Corporation) and TPC (Transaction Processing Performance Council). Both are non-profit organisations and their members include most of the major companies in the computer industry. A third organisation is the OLAP Council that provides benchmarks for on-line analytical processing technology. Although there is a clear overlap between benchmarks from OLAP Council and some TPC benchmarks, there are some differences in the philosophy of the benchmarks from the two organisations that deserve some attention. In this survey we will concentrate on the benchmarks of the three organisations mentioned above: SPEC, TPC, and OLAP Council.

[12] Excluding benchmarks for embedded systems.

5.1 SPEC Benchmarks

SPEC was founded in 1988 and has more than 60 members today, including most of the major computer industry companies.

SPEC benchmarks consist of a standardised source code taken form established applications (i.e., real-life applications) and modified by SPEC to improve portability and accommodate some specific requirements of performance benchmarking [Eigenmann 2001]. In most of the cases, the workload of a SPEC benchmark includes several applications that are run in sequence one after the other. Typically, to run a SPEC benchmark all what is required is to compile the workload for a specific system and then tune the system for the best results.

Detailed information on SPEC groups and benchmarks can be found on SPEC web site [SPEC]. SPEC is in practice an umbrella organisation encompassing three different groups: the Open System Group (OSG), the High Performance Group (HPG), and the Graphics Performance Characterisation Group (GPCG). These three groups are presented in the next three paragraphs.

5.1.1 Open System Group (OSG)

The OSG focuses on component and systems-level benchmarks for workstations and multi-user servers running Unix, Windows NT, Linux, and other operating systems (although some OSG benchmarks run only on some operating systems). Current benchmarks of OSG are as folows:

- SPEC CPU2000 is a suite of compute-intensive benchmarks that measures performance of the computer processor, memory and compiler. It includes two benchmarks suites: CINT2000 for measuring and comparing compute-intensive integer performance, and CFP2000 for measuring and comparing compute-intensive floating point performance. CINT2000 consists of twelve applications and CFP2000 includes fourteen applications. All the applications are real applications and they are written in several languages such as C, C++, Fortran77, and Fortran90. Each of these benchmark suites includes four metrics: two speed metrics, corresponding to the execution time of the applications compiled respectively with conservative and aggressive compiler optimisation choices, and two throughput metrics, corresponding to the rate of execution of the applications in a given time using again conservative and aggressive compiler optimisation choices.

- SPEC JBB2000 is a server-side Java benchmark emulating a 3-tier system with emphasis on the middle tier. It is written in Java and implements a middle tier business logic following the general model of the TPC-C benchmark(see Section 5.2.1). SPEC JBB2000 does not include a real database:TPC-C database tables are replaced by Java classes and table records are replaced by Java objects. The metrics include SPECjbb2000 ops/second that is a composite throughput measurement representing the averaged throughput over a range of application points (related to the emulated business model).

- SPEC JVM98 is a benchmark suite for comparing Java virtual machine (JVM) client platforms. It includes eight different programs, five of which are real applications or are derived from real applications. In addition to the programs used for computing performance metrics, this benchmark also includes one program to validate some of the features of Java, such as testing for loop bounds. This benchmark has two metrics that provide measures of the fastest and slowest execution times, obtained by running the benchmark a number of times. Typically, the first execution is the slowest due to the overhead of loading and verify Java classes, perform security checks, initialisation of static variables, etc, while the best time will be from a later benchmark run.

- SPEC MAIL2001 is a mail server benchmark designed to measure the performance of a system working as a mail server based on the Internet standard protocols SMTP and POP3. The benchmark metric is SPECmail2001 messages per minute, measured using a simulated email environment including network connections, disk storage, and client workloads. It is worth noting that the reported throughput (messages per minute) must meet a specified Quality of Service (QoS) criterion, which takes into account response time, errors, minimum rates, etc.

- SPEC WEB99 is benchmark for measuring the performance of World Wide Web Servers. This benchmark simulates the accesses to a web service provider, where the server supports the home page for a number of different organisations. The benchmark tries to simulate this environment in a realistic way, including all the typical features and operations of a large web server, such as static and dynamic accesses, "rotating" advertisements on a web page, large variety of file sizes, etc. The SPECweb99 metric is the maximum number of simultaneous connections that the web server is able to support while still meeting specific throughput and error rate requirements.

5.1.2 High Performance Group (HPG)

The HPG benchmarks are especially suited for evaluating the performance of parallel and distributed computer architectures, with particular emphasis on high-performance computer systems. The current benchmark suite form HPG is the SPEChpc96. This benchmark suit includes three real world programs and, in spite of being targeted to high-end systems, this benchmark can be used in a wide range of systems, such as workstations, clusters, Symmetric Multiprocessors (SMPs), vector systems and (Massively Parallel Processors (MPPs). Due to this large variety of target systems, the SPEChpc96 benchmarks can be executed in serial or parallel mode. Parallel implementations are based on programming models such as Parallel Virtual Machine (PVM) and Message-Passing Protocol (MPI) and on the directive-based OpenMP application programming interface (API). The SPEChpc96 metrics include measures of speed and throughput. The speed metric consists of the time it takes to run an application on the system being tested and the throughput metric reports how many benchmarks could be run, back-to-back, in a given 24-hour period.

5.1.3 Graphics Performance Characterization Group (GPC)

The GPC group was an independent group until 1996, when it joined the SPEC organisation. The GPC includes several autonomous project groups that develop performance benchmarks for graphics subsystems. Most of these project groups have been created outside the GPC and joined the GPC later on. In the same way SPEC is an umbrella organisation, the GPC group is also another umbrella under the SPEC organisation. The current project groups that constitute the GPC are the *Application Performance Characterization (SPECapc), OpenGL Performance Characterization (SPECopc℠),* and *SPECmedia Project Group℠*. The following points summarize these project groups and respective benchmarks:

- *Application Performance Characteriation (SPECapc)* – This group is focused on performance benchmarks for graphics-intensive applications. This project group has created several graphics performance benchmarks from real application such as SPECapc Solid Edge V9, SPECapc for 3D Studio MAX R3, SPECapc for Pro/ENGINEER™ 2000i, SPECapc's SolidWorks 99, and SPECapc for Unigraphics V15.

- *OpenGL Performance Characterization (SPECopc℠)* – This group is aimed at establishing graphics performance benchmarks for systems running under the OpenGL API. Currently the project group has two benchmarks: SPECviewperf 6.1.2 that measures the 3D rendering performance of systems running under OpenGL and the SPECglperf, that measures optimal performance of 2D and 3D graphics primitives across vendor platforms.

- *SPECmedia Project Group℠* – This project group has been formed from the Multimedia Benchmark Committee and is currently working on a first benchmark called MPEG-2. This benchmark will be a system-level benchmark that will test processor, memory, and graphic capabilities.

5.2 TPC Benchmarks

The Transaction Processing Performance Council (TPC) is a non-profit corporation formed by the major vendors of systems and software from the transaction processing and database market. The goal of TPC is to define and disseminate performance benchmarks for transaction processing systems. Detailed and latest information on TPC organisation and TPC benchmarks can be obtained from TPC's web site [TPC].

The notion of transaction used by TPC is very close to the usual business meaning of the word "transaction". In fact, from the point of view of TPC, a computer transaction is the set of operations in a computer system required to support business transactions such as common commercial exchanges of goods, services, or money. A typical transaction, as defined by the TPC, includes normally reading from, writing to, or updating a database system for things such as inventory control (goods), airline reservations (services), or banking (money). Decision support operations, which consist of reading and summarizing large amounts of historical data, are also considered as a form of transactions (read-only transactions).

A typical transactional environment consists of a number of users managing their transactions via a terminal or a desktop computer connected to a database. TPC benchmarks try to measure transaction processing and database performance in terms of how many transactions a given system can execute per unit of time. Thus, TPC benchmarks are clearly system-oriented and their measures reflect the end-user point of view.

All TPC benchmarks include two kinds of measures: performance and price/performance measures. The performance measure is a transaction rate (e.g., number of transactions per minute) and the price/performance measures are based on a detailed set of pricing rules that take into account the price of purchasing the system (hardware and software) and the maintenance costs for a given period.

Unlike SPEC benchmarks, that heavily rely on the source code of the workload, TPC transactions are defined by a very detailed specification. So, to run a TPC benchmark it is necessary to implement the specification in the target system, which means that it is necessary to program the transactions and all the aspects defined in a functional way in the benchmark specification. In practice, existing code and examples (some of them provided together with the benchmark specifications) are adapted to new target systems, which greatly simplifies the implementation of TPC benchmarks.

Conducting a TPC benchmark is a rather complex process. In addition to implementing the benchmark specification in the target system and running the benchmark, a TPC auditor must approve all the benchmark results before they can be submitted to TPC for approval. The submission of benchmark results to TPC must also include a full disclosure report containing all the relevant technical details, as defined in the benchmark specification. The benchmark results submitted to the TPC council must be analysed and approved by TPC before being publicised by vendors. This complex process associated to the fact that it is very expensive to install and tune a winning transactional systems is one of the reasons why TPC benchmarks are often referred as very expensive benchmarks. It should be noted, however, that the implementation of most of TPC benchmarks in an *"private"* way (i.e., with no intent to submit results to TPC) is straightforward and can be inexpensive if a low-end server and database are used. This makes TPC benchmarks quite useful for research purposes, as they do represent agreed workloads for the transactional area.

The TPC has currently four benchmarks: TPC-C for OLTP (On-Line Transaction Processing) systems, TPC-W for transactional Web systems such as e-commerce systems, and two benchmarks for decision support systems, the TPC-H for *ad hoc* decision support queries (queries may not be known in advance) and the TPC-R for business reporting and decision support queries (when pre-knowledge of the queries is assumed and may be used for optimization). The following subsections describe briefly each of these benchmarks. For complete description refer to the public specifications of TPC benchmarks available for download from the TPC web site [SPEC].

5.2.1 TPC-C

TPC-C simulates a complete OLTP (On-Line Transaction Processing) system such as the ones that constitute the kernel of the information systems used today to support the daily operations of most business. These systems are very database centric, as a number of users are connected to a database through a terminal or a desktop computer and submit transactions to the database. These transactions constitute small and consistent amounts of work from the application (business) point of view and normally comprise a small number of read and write operations that touch just a few records of the database. Very often, these OLTP are referred as short transactions to emphasize the fact that the system can execute a large amount of these short transactions at the same time.

The typical form of business represented by TPC-C is a wholesale supplier having a number of warehouses and their associated sales districts, and where the users submit transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. This workload does not correspond to a particular business, but tries to represent any business that must manage, sell, or distribute products or services.

TPC-C specifies the complete set of database tables, the features of the data used to populate these tables, and the mix of transactions that constitute the workload. It includes five concurrent transactions of different types and complexity, either executed on-line or in a deferred mode, and meant to exercise all the main system components. The following points, transcript from TPC-C documentation (available from the SPEC web site [SPEC]), summarize the main characteristics of TPC-C queries:

- *simultaneous execution of multiple transaction types that span a breadth of complexity,*
- *on-line and deferred transaction execution modes,*
- *multiple on-line terminal sessions.*
- *moderate system and application execution time,*
- *significant disk input/output,*
- *transaction integrity: Atomicity, Consistency, Isolation, and Durability (ACID) properties,*
- *non-uniform distribution of data access through primary and secondary keys,*
- *databases consisting of many tables with a wide variety of sizes, attributes, and relationships,*
- *contention on data access and update.*

TPC-C measures consist of a transaction rate (new-order transactions per minute – *tpmC*) and the associated price per transaction *($/tpmC*).

5.2.2 TPC-W

The TPC-W is a transactional web benchmark that simulates the activities of an Internet commerce environment. As the other TPC benchmarks, TPC-W does not represent a particular (i.e., real) business, but tries to emulate a typical e-commerce environment having browsing, shopping and web-based ordering operations. The main characteristics of TPC-W workload are (taken from TPC-W documentation available from SPEC web site [SPEC]:

- *multiple on-line browser sessions,*
- *dynamic page generation with database access and update*
- *consistent web objects,*
- *simultaneous execution of multiple transaction types that span a breadth of complexity,*
- *on-line transaction execution modes,*

- *databases consisting of many tables with a wide variety of sizes, attributes, and relationships,*
- *transaction integrity (ACID properties),*
- *contention on data access and update.*

The TPC-W measures consist of the number of web interactions processed per second (*WIPS*) and the associated price per WIPS *($/WIPS*). Several types of web interactions are used to simulate the activity of a real e-commerce, and each interaction is subject to a response time constraint.

5.2.3 TPC-H

The TPC Benchmark™H (TPC-H) is a decision support benchmark for *ad hoc* queries. The information stored in these decision support systems (also called "data warehouses") is historical in nature and can include detailed transactional data from operational databases, legacy systems, worksheets or even external data sources [Chaudhuri 1997]. Typically, operational systems such as the ones portrayed in the TPC-C and TPC-W benchmarks, that support the daily operations of the enterprises, are the natural source of data for the decision support systems. This data is periodically collected from the operational systems (i.e., the OLTP systems used to support the daily operations of the enterprise) and after being integrated and conciliated, it is stored in the data warehouse for decision support purposes. Managers analyse this data not only to characterize the past activity of the enterprise, but also to predict trends in the business and to get support for business decisions.

The decision making process is often based on a sequence of interactive queries. That is, the answer of one query immediately sets the need for a second query, and the answer of this second query raises another query, and so on in an *ad hoc* manner. Thus, efficient query processing for unpredictable queries is a critical requirement for these systems. Furthermore, the decision support queries are often very complex and generally access huge volumes of data and perform many database joins and aggregations. In spite of the *ad hoc* nature and complexity of the queries, a typical decision support system must assure interactive response times, which explains why performance is a key aspect of these systems.

The TPC-H benchmark is intended to emulate a typical decision support environment and consists of a suite of business-oriented *ad hoc* queries and some concurrent data modifications. Following the general philosophy of the TPC organisation, TPC-H is not based on a particular real case, but the whole benchmark has been designed to have broad industry-wide relevance, considering both the data used to populate the database tables and the queries used to collect the performance measures.

The performance metric reported by TPC-H is the TPC-H composite query-per-hour performance metric (*QphH@Size*), and reflects multiple aspects of the capability of the system to process queries. These aspects include the selected database size against which the queries are executed, the query processing power for a single stream, and the query throughput when multiple concurrent users submit queries. The TPC-H price/performance metric is expressed as *$/QphH@Size*. The pricing rules are similar to the ones used in other TPC benchmarks.

Another important aspect reflected in these metrics (actually it is part of the name of the measures) is that the size of the database tables has great impact on the performance results. TPC-H includes a set of scaling rules to make it possible to use the benchmark in systems with very different sizes.

5.2.4 TPC-R

The TPC Benchmark™R (TPC-R) is a decision support benchmark similar to TPC-H. The main difference is that it allows additional optimisations based on the previous knowledge of the queries. The idea is that many decision support systems are normally used for the production of extensive reports whose queries do not change and are known in advance. In these cases the database administrators can use several tuning techniques and data access structures that cannot be used when the queries are not known in advance (*ad hoc* queries). In practice, the way TPC-H and TPC-R simulates the *ad hoc* query and report environments is by allowing (in TPC-R) and not allowing (in TPC-H) the use of specific techniques and data structures such as indexes [Salzberg 1996, Chan 1998] over some key attributes and materialized views [Chauduri 1997, Roussopoulos 1998]. For practical reasons TPC decided to keep two separate benchmarks for the *ad hoc* and reporting environments. Naturally, TPC-R and TPC-H are very similar in many aspects.

The performance metric reported by TPC-R is the TPC-R composite query-per-hour (*QphR@Size*), and reflects basically the same aspects of the system capabilities already mentioned for the TPC-H. The TPC-R price/performance metric is expressed as *$/QphR@Size* and use similar pricing rules as TPC-H.

5.3 OLAP Council's APB-1 Benchmark

The OLAP Council was established in January 1995 and includes important vendors of on-line analytical processing (OLAP) technology. Unlike TPC, that covers performance benchmarking for a variety of transactional systems, the OLAP Council is focused on performance benchmarks for OLAP systems. Some examples of OLAP applications are financial modelling (budgeting and planning), sales forecasting, customer and product profitability, resource allocation and capacity planning, promotion planning, market share analysis, etc. The OLAP Council has currently one performance benchmark called APB-1. Detailed information on the OLAP Council organisation and APB-1 benchmark can be obtained from the Council's web site [OLAP].

There is a clear overlap between APB-1 and the TPC-H and TPC-R benchmarks from TPC, in the sense that all of them can be used for performance evaluation and comparison of similar systems. However, APB-1 is clearly designed having the multidimensional model in mind [Kimball 1998], which is not evident for both TPC-H and TPC-R. In some sense, APB-1 tries to reflect all the recent developments in the field of decision support and on-line analytical processing technology, while TPC-H and TPC-R reflect most the classical database culture and jargon applied to decision support applications. OLAP applications operate over a database organised according to the multidimensional model [Kimball 1998][13]. Each dimension of this model represents a different perspective for the analysis of the business. For instance, in the classical example of a chain of store business, some of the dimensions are products, stores, and time. Each cell within the multidimensional structure (a cube in this simple three dimension example) contains data (typically numerical facts) along each of the dimensions. For example, a single cell may contain the total sales for a given product in a given store in a single day.

Although the most flexible way to store the data in a data warehouse could be a multidimensional database server, most of the data warehouses and OLAP applications store the data in a relational database. That is, the multidimensional model is implemented as a star scheme [Kimball 1998] formed by a large central fact table surrounded by several dimensional tables related to the fact table by foreign keys. This is another reason for the overlap between APB-1 and the decision support benchmarks from TPC. In spite of the fact that APB-1 has been specifically designed for multidimensional systems, in practice these benchmarks became quite similar as most of the target systems implement the multidimensional database in the same way (i.e., by using a normal relational database).

The goal of the APB-1 is to measure the overall performance of the OLAP server. To achieve this, the workload is composed of a large set of operations and queries to reflect common business operations. APB-1 provides a single metric called *AQM* (*Analytical Queries per Minute*). This measure represents the number of analytical queries processed per minute including data loading and computation time.

The APB-1 implementation and conducting follow the same basic approach used by TPC benchmarks. The benchmark is defined by a detailed specification and to run the benchmark it is necessary to implement the specification in the target system (again, as for TPC benchmarks, previous examples and code can be reused to make the benchmark implementation easier). A full disclosure report is required by OLAP Council prior to publish APB-1 results. This report must be approved by certified OLAP Council auditors, which corresponds to the general proceeding used by TPC.

5.4 Discussion of Specific Aspect of Existing Benchmarks and their Relevance for DBench

This survey confirms one of the DBench premises that dependability has been largely absent of the benchmarking effort so far. In fact, even in the case of some TPC and SPEC benchmarks that address application areas where dependability represents a key role (e.g., TPC-C, TPC-W, SPEC WEB99, and SPEC MAIL2001), most the aspects and components related to dependability have been largely ignored. In particular, even in the few cases when some dependability features are specified in the benchmark, there is no attempt to measure the

[13] The decision support terminology mixes terms such as “OLAP” and “data warehouses” very often. Many times, a tool that explores data in an analytical way is called OLAP tool or OLAP application, while the actual database that stores the data that is explored is called the data warehouse. However, the term data warehouse is also applied to the entire system used for decision support. To make things even worst, the term “data mart” is also used to refer small data warehouses dedicated to a specific business area.

efficiency of these features, or to take explicitly the effect of faults into account in the performance measures.

For example, TPC-C specifies that data recovery features of the database must ensure that data can be recovered from any point in time during the running of the benchmark and that the ACID (Atomicity, Consistency, Isolation, and Durability) properties of transaction processing systems must be supported by the system under test during the running of the benchmark. However, the benchmark specification does not include any procedure to confirm that these mechanisms are working properly or to measure the impact of the recovery activation in the system performance (however, the auditors should report on this). The SPEC MAIL2 specifies a Quality of Service (QoS) criterion that the reported throughput (messages per minute) must meet, which takes into account aspects such as response time, errors, and minimum rates. Although these are very interesting benchmark features that must be accommodated in DBench, this QoS criterion is only related to workload conditions.

The different organisations involved in performance benchmark activities have quite different philosophies concerning both the way benchmarks are designed and used. While SPEC provide benchmarks almost ready to run (i.e., what is necessary is to compile the workload into the target system), TPC and OLAP Council benchmarks assume the form of a detailed specification (essentially a functional specification) that must be implemented and run on the target system. Several reasons account for these differences, but clearly the different kind of target systems and the different degree in the benchmark complexity are the major ones. For example, TPC-W and SPEC WEB99, just to mention two benchmarks for the same general application area, have huge differences in complexity and range of different target systems that can be addressed (clearly, TPC-W is much more complex). Although the advantages of a “ready to run” benchmark are quite obvious, the complexity added by the dependability metrics suggests that future dependability benchmarks will tend to follow the specification-based model. The research planned for DBench will help to clarify this point.

TPC (especially) and OLAP Council (as well) keep a strict control on the benchmark results and decide whenever the proposed result can be accepted as benchmark results or not. This is understandable, as the TPC and OLAP Council benchmarks have to be implemented and it is necessary to verify that the implementation follows the benchmark specification. This is just one of the reasons why TPC benchmarks are pointed out as being very expensive benchmarks. SPEC benchmarks, on the contrary, cannot be changed (in general)[14] as they are delivered in the form of a package containing program source code and the tools required to run the benchmark. This way, running the benchmark and getting the results is very simple and does not require SPEC approval to publish the results (although SPEC may declare some results as non valid if it detects that the results have been obtained without observing the benchmark documentation).

The number of results published is the key aspect of the success of a performance benchmark. This is true for all the organisations involved and that can be easily identified by the mere observation of the emphasis they put on the number of published results in the web pages of all the benchmark organisations. However, for the complexity reasons mentioned above, the way

[14] Still,SPEChpc96 allows some changes and adaptations in the code.

(and the number) the benchmark results published are very different for the three benchmark organisations. TPC and OLAP Council results are published by system and software vendors while SPEC results are published not only by vendors but also by widely available magazines and consumer organisations. In other words, the users of the target systems can easily run most of the SPEC benchmarks while for TPC benchmarks, the users (i.e., system buyers) are limited in practice to the available results published by vendors. From a vendor point of view, besides the cost attached to actually run benchmarks (especially those benchmarks characterized simply by functional specifications), another major concern is related to the cost for tuning the system for running the benchmark.

Finally, a common aspect related to the performance measures provided by existing performance benchmarks is that all of them rely on direct measures only (i.e., measures that can be obtained directly from running the benchmark as specified in the benchmark documentation). In DBench a clearly innovative aspect is that we need modelling and other methods to estimate comprehensive dependability measures from specific results collected during the benchmark running. However, we must not forget that the relevance put by the performance benchmark community on the direct measures is mainly related to the need of defining tamper-proof benchmarks. All the indirect ways of obtaining results are avoided to make it easier to solve the problem of producing optimistic results. This means that as we need modelling to obtain some dependability measures, we also will need special care to avoid optimistic results in this additional layer.

6 Conclusion

This report surveys current state-of-the-art techniques and best practices for dependability assessment and performance benchmarking, with the aim of identifying techniques and problems relevant to dependability benchmarking. The survey has helped identify areas requiring more research as part of the development of a dependability benchmarking framework. While solutions to all problems cannot be expected from this type of survey, the awareness of existing solutions to similar problems can be beneficial.

The purpose of a dependability benchmark is to assess the dependability of the system under test. Today, there exist two major techniques for conducting such an assessment: model-based and measurement-based techniques. A purely model-based benchmark for COTS components is impossible, since the models require knowledge about the internal architecture of the system, which is not usually known for COTS components. A measurement-based benchmark, using fault injection, works well with COTS components. However, fault injection alone will not be able to provide all the desired measures, in particular comprehensive dependability measures, such as availability and reliability. Therefore, a combination of the techniques is necessary. Studies combining modelling and fault injection have been carried out, but it is likely that further research will be required in order to incorporate such methods into the dependability benchmarking framework.

As dependability benchmarking relies on experimental approaches, to a large extent, many aspects will be common to conducting fault-injection experiments.

Software implemented fault injection (SWIFI) appears as the most suitable method for injecting faults in a dependability benchmark. Compared to physical fault-injection, SWIFI is an easy and cheap technique. It has also been proved that SWIFI can be used to emulate both software and hardware faults. When injecting faults into a system, attributes such as coverage, failure modes and fault representativeness are important. Years of research in the area of fault-injection (including simulation techniques) have resulted in methods for taking them into consideration.

Robustness benchmarking is clearly related to dependability benchmarking, the difference being the type of fault model used. For robustness benchmarking, traditionally only faults entering the systems through its defined interface (e.g., the application programming interface for an operating system) is considered, while dependability benchmarks may consider faults affecting the system by other means, e.g., internal hardware faults. One shortcoming of current robustness benchmarks is that they fail to consider the representativeness of the faults injected, which will become a major research area for dependability benchmarks. Another shortcoming is that stressful conditions are not considered.

Performability, which incorporates both the concept of performance and dependability, also provides a link between the areas in this survey. Since it is normal for systems to contain multiple processing units that are used to increase performance, if one unit fails, it is possible to redistribute the load among the remaining units. This will often provide an increase in dependability, but at the same time the performance of the system will then be lower than in the original configuration. This may in turn cause the system to fail due to high load. Thus, it is important to take performability aspects into account when designing a dependability benchmark.

A common feature of all performance benchmarks is that all of them rely on a single metric or on a reduced set of metrics. It is quite clear that several performance metrics would express much better the different aspects related to system or component performance. However, the reality is that the goal of many performance benchmarks is to produce performance results for the customers (system buyers) and the customers are normally not experts on system performance. A benchmark that produces a single metric is easier to understand and probably that is the reason why all benchmarks feature a very small number of metrics.

Concerning DBench research, it is very hard to imagine a dependability benchmark based on a single metric. Nevertheless, the lesson learnt from the performance benchmark community is that all benchmark users must understand the benchmark results. The performance benchmarking organizations solved this problem by focusing on a small set of easy to understand measures, instead of providing a large set of measures that could only be understood by specialists. This means that in our case we should find ways to provide clear dependability benchmark measures, even for the classes of users that are not necessarily experts on computer dependability.

Achieving portability is difficult for all categories of benchmarks. Using standardized interfaces is one way of making benchmarks portable. An example of this is current robustness benchmarks targeting COTS components. Some transaction processing benchmarks define the workload as a set of specifications to be implemented. This is another way of increasing portability. Both these techniques, along with others, have to be considered in the DBench framework.

The importance of workload representativeness has been emphasized for all types of performance benchmarks, and this will become an issue for dependability benchmarks as well. In benchmarks for transactional systems some workloads have been established as consensus workloads, i.e., they are considered by many to be representative for such applications. Therefore, it should be possible to use such workloads in a dependability benchmark. No such consensus workloads exist for microprocessor performance benchmarks, but much useful work has been carried out on finding representative workloads for different application areas.

As evidenced by performance benchmarks experience, a benchmark requires that specific procedures for application and rules for disclosure be enforced, so that misuse can be prevented. As these benchmarks have maturated, this role has been transferred to large consortiums with representatives from the computer industry. It is expected that similar procedures and rules will be needed for dependability benchmarking, as well.

There are still many open research issues to be dealt with before a dependability benchmark can become a reality. However, the experiences from the dependability assessment and performance benchmarking disciplines must not be ignored. These experiences are used in the preliminary dependabilitybenchmarking framework, presented in the companion document [CF2 2001].

7 References

[Amoia *et al.* 1981] V. Amoia, G. D. Micheli and M. Santomauro, “Computer-Oriented Formulation of Transition-Rate Matrices via Kronecker Algebra”, *IEEE Trans. on Reliability,* R-30 (2), pp.123-32, June 1981.

[Arlat *et al.* 1989] J. Arlat, Y. Crouzet and J.-C. Laprie, “Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems”, in *Proc. 19th Int. Symp. on Fault-Tolerant Computing (FTCS-19),* (Chicago, IL, USA), pp.348-55, IEEE CS Press, 1989.

[Arlat *et al.* 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E Martins and D. Powell, “Fault Injection for Dependability Validation — A Methodology and Some Applications”, *IEEE Trans. on Software Engineering,* 16 (2), pp.166-82, February 1990.

[Arlat *et al.* 1993] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems”, *IEEE Trans. on Computers,* 42 (8), pp.913-23, August 1993.

[Arlat *et al.* 1999] J. Arlat, J. Boué and Y. Crouzet, “Validation-based Development of Dependable Systems”, *IEEE Micro,* 19 (4), pp.66-79, July-August 1999.

[Arlat *et al.* 2002] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, “Dependability of COTS Microkernel-based Systems”, *IEEE Transactions on Computers,* 51 (2) February 2002. (To appear - Also LAAS Report 00-466, Revision July 2001).

[Avizienis & Rennels 1972] A. Avizienis and D. Rennels, “Fault-Tolerance Experiments with the JPL STAR Computer”, in *Proc. 6th Annual IEEE Computer Society Conference (COMPCON'72),* (San Francisco, CA, USA), pp.321-4, IEEE CS Press, 1972.

[Balakrishnam & Trivedi 1995] M. Balakrishnam and K. S. Trivedi, “Component-wise Decomposition for an Efficient Reliability Computation of Systems with Repairable Components”, in *25th Int. Symp. on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.259-68, IEEE CS Press, 1995.

[Balbo *et al.* 1988] G. Balbo, S. C. Bruell and S. Ghanta, “Combining Queuing Networks and GSPNs for the Solution of Complex Models of System Behaviour”, *IEEE Trans. on Computers,* 37, pp.1251-68, 1988.

[Béounes *et al.* 1993] C. Béounes, M. Aguéra, J. Arlat, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell and P. Spiesser, “SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems”, in *Proc. 23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.668-673, IEEE CS Press, 1993.

[Blough & Torii 1997] D. M. Blough and T. Torii, “Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message Passing Parallel Computers”, in *Proc. 27th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-27),* (Seattle, WA, USA), pp.258-67, IEEE CS Press, 1997.

[Bondavalli *et al*. 1997] A. Bondavalli, I. Mura and M. Nelli, "Analytical Modeling and Evaluation of Phased Mission Systems in space Applications", *in 2nd IEEE High Assurance System Engineering Workshop (HASE),* (Bethesda, MD, USA), pp.85-91, 1997.

[Bondavalli *et al*. 1999] A. Bondavalli, I. Mura and K. S. Trivedi, "Dependability Modelling and Sensitivity Analysis of Scheduled Maintenance Systems", in *3rd European Dependable Computing Conference (EDCC-3),* (A. Pataricza, J. Hlavicka and E. Maehle, Eds.), (Prague, Czech Republic), pp.7-23, Springer, 1999.

[Bondavalli *et al*. 2000] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli and F. Sandrini, "DEEM: a Tool for the Dependability Modeling and evaluation of Multiple Phased Systems", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.231-6, IEEE CS Press, 2000.

[Bouissou 1993] M. Bouissou, "The FIGARO Dependability Evaluation Workbench in Use: Case Studies for Fault- Tolerant Computer Systems", in *23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.680-5, IEEE CS Press, 1993.

[Bouricius *et al*. 1969] W. G. Bouricius, W. C. Carter and P. R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems", in *Proc. 24th. National Conference,* pp.295-309, ACM Press, 1969.

[Bradley *et al*. 1995] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer and Michael Smith, "The Measured Performance of Personal Computer Operating Systems", in *Proc. 15th ACM Symp. on Operating System Principles*, volume 29, pp. 299--313. ACM SIGOPS Operating System Review, December 1995.

[Brown & Seltzer 1997] A. Brown and M. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: Case Study of the Performance of NetBSD on the Intel Architecture", in *Proc. Sigmetrics Conference*, pp. 214--224, Seattle, WA, June 1997.

[Buckley & Siewiorek 1995] M. F. Buckley and D. P. Siewiorek, "VAX/VMS Event Monitoring and Analysis", in *Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.414-23, IEEE CS Press, 1995.

[Buckley & Siewiorek 1996] M. F. Buckley and D. P. Siewiorek, "A Comparative Analysis of Event Typling Schemes", in *Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26),* (Sendai, Japan), pp.294-303, IEEE CS Press,1996.

[Busquets 1994] J. V. Busquets, K. Nilsen, "Proposal for a New Real-Time Benchmark". *Internal Report DISCA, Technical University of Valencia*, 1994.

[Butner & Iyer 1980] S. E. Butner and R. K. Iyer, "A Statistical Study of Reliability and System Load at SLAC", in *Proc. 10th Int. Symp. on Fault Tolerant Computing (FTCS-10),* (Kyoto, Japan), IEEE CS Press,1980.

[Carrasco & Figueras 1986] J. A. Carrasco and J. Figueras, "METFAC: Design and Implementation of a Software Tool for Modeling and Evaluation of Complex Fault-Tolerant Computing Systems", in *Proc. 16th Int Symp. on Fault-Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.424-9, IEEE CS Press, 1986.

[Carreira *et al*. 1998] J. Carreira, H. Madeira and J. G. Silva, “Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers”, *IEEE Trans. on Software Engineering,* 24 (2), pp.125-36, February 1998.

[Carreira *et al.* 1999] J. V. Carreira, D. Costa and J. G. Silva, “Fault Injection Spot-checks Computer System Dependability”, *IEEE Spectrum,* 36, pp.50-5, August 1999.

[Carrette 1996] G. J. Carrette, *“CRASHME: Random Input Testing”*, http://people.delphi.com/gjc/crashme.html, 1996.

[Castillo & Siewiorek 1981] X. Castillo and D. P. Siewiorek, “Workload, Performance, and Reliability of Digital Computing Systems”, in *Proc. 11th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-11),* (Portland, Maine, USA), pp.279-85, 1981.

[CF2 2001 H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johansson, R. Lindström, S. Blanc, K. Buchacker, J. Durães, P. Gil, T. Jarboui, J.-C Laprie, J. J. Serrano, J. G. Silva, N. Suri and M. Vieira, "*Conceptual Framework, Deliverable CF2, Preliminary Dependability Benchmark Framework*", DBench Project, IST 2000-25425.

[Chan 1998] C. Y. Chan, Y. E. Ioannidis, “Bitmap Index Design and Evaluation”, SIGMOD, pp. 355-66, 1998.

[Chaudhuri 1997] S. Chauduri and U. Dayal, “An overview of data warehousing and OLAP technology”, SIGMOD Record, 26(1), pp. 65-74, March 1997.

[Chillarege & Bowen 1989] R. Chillarege and N. S. Bowen, “Understanding Large System Failures — A Fault Injection Experiment”, in *Proc. 19th Int. Symp. on Fault-Tolerant Computing (FTCS-19),* (Chicago, IL, USA), pp.356-63, IEEE CS Press, 1989.

[Chillarege *et al.* 1995] R. Chillarege, S. Biyani and J. Rosenthal, “Measurement of Failure Rate in Widely Distributed Software”, in *Proc. 25th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.424-33, IEEE CS Press,1995.

[Choi & Iyer 1992] G. S. Choi and R. K. Iyer, “FOCUS: An Experimental Environment for Fault Sensitivity Analysis”, *IEEE Trans. on Computers,* 41 (12), pp.1515-26, December 1992.

[Ciardo & Miner 1996] G. Ciardo and A. S. Miner, “SMART: Simulation and Markovian Analyzer for Reliability and timing”, in *Proc. 2nd IEEE Int. Computer Performance and Dependability Symp. (IPDS'96),* (Urbana-Champain, IL, USA), p.60, IEEE CS Press, 1996.

[Ciardo & Miner 1999] G. Ciardo and A. Miner, “A Data Structure for the Efficient Kroneker Solution of GSPNs”, in *Proc. 8th Int. Workshop on Petri Nets and Performance Models,* (Zaragoza, Spain), pp.22-31, IEEE CS Press, 1999.

[Ciardo *et al*. 1989] G. Ciardo, J. K. Muppala and K. S. Trivedi, “SPNP: Stochastic Petri Net Package”, in *Proc. 3rd Int. Workshop on Petri Nets and Performance Models,* pp.142-51, IEEE CS Press, 1989.

[Cramp *et al*. 1992] R. Cramp, M. A. Vouk and W. Jones, “An Operational Availability of a Large Software-Based Telecommunications System”, in *Proc. 3rd Int. Symp. on Software Reliability Engineering,* (Research Triangle Park, NC, USA), pp.358-66, 1992.

[Cukier *et al.* 1998] M. Cukier, J. Arlat and D. Powell, “Frequentist and Bayesian Coverage Estimations with Stratified Fault-Injection”, in *Dependable Computing for Critical Applications (Proc. 6th IFIP Int. Working Conference on Dependable Computing for Critical Applications: DCCA-6, Grainau, Germany, March 1997)* (M. Dal Cin, C. Meadows and W. H. Sanders, Eds.), Dependable Computing and Fault-Tolerant Systems, 11, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.43-61, IEEE CS Press, 1998.

[Cukier *et al.* 1999] M. Cukier, R. Chandra, D. Henke, J. Pistole and W. H. Sanders, “Fault Injection based on a Partial View of the Global State of a Distributed System”, in *Proc. 18th Symp. on Reliable Distributed Systems (SRDS'99),* (Lausanne, Switzerland), pp.168-77, IEEE CS Press, 1999.

[Dawson *et al.* 1996] S. Dawson, F. Jahanian, T. Mitton and T.-L. Tung, “Testing of Fault-Tolerant and Real-Time Distributed Systems via protocol Fault Injection”, in *Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26),* (Sendai, Japan), pp.404-14, IEEE CS Press, 1996.

[DeLong *et al.* 1996] T. A. DeLong, B. W. Johnson and J. A. Profeta III, “A Fault Injection Technique for VHDL Behavioral-Level Models”, *IEEE Design and Test of Computers,* Winter, pp.24-33, 1996.

[Donohoe 1987] P. Donohoe. “A Survey of Real-Time Performance Benchmarks for the ADA Programming Language”. *Technical Report CMU/SEI-87-TR-28 ESD-TR-87-191*, December 1987.

[Eigenmann 2001] Rudolf Eigenmann (Editor), *Performance Evaluation and Benchmarking With Realistic Applications,* MIT Press, 2001.

[Essamé *et al.* 1997] D. Essamé, J. Arlat and D. Powell, “Available Fail-Safe Systems”, in *Proc. 7th Workshop on Future Trends of Distributed Computing Systems (FTDCS'97),* (Tunis, Tunisia), pp.176-82, IEEE CS Press, 1997.

[Fabre *et al.* 1999] J.-C. Fabre, F. Salles, M. Rodríguez Moreno and J. Arlat, “Assessment of COTS Microkernels by Fault Injection”, in *Dependable Computing for Critical Applications (Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7, San Jose, CA, USA, January 1999)* (C. B. Weinstock and J. Rushby, Eds.), Dependable Computing and Fault-Tolerant Systems, 12, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.25-44, IEEE CS Press, 1999.

[Folkesson *et al.* 1998] P. Folkesson, S. Svensson and J. Karlsson, “A Comparison of Simulation Based and Scan Chain Implemented Fault Injection”, in *Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28),* (Munich, Germany), pp.284-93, IEEE CS Press, 1998.

[Fota *et al.* 1999a] N. Fota, M. Kâaniche and K. Kanoun, “Dependability Evaluation of an Air Traffic Control Computing System”, *Performance Evaluation,* 35 (3-4), pp.553-73, 1999.

[Fota *et al.* 1999b] N. Fota, M. Kâaniche and K. Kanoun, “Incremental Approach for Building Stochastic Petri Nets for Dependability Modeling”, in *Statistical and Probabilistic Models in Reliability* (D. C. Ionescu and N. Limnios, Eds.), pp.321-35, Birkhäuser, 1999.

[Furht 1989] B. Furht, J. Parker, D. Grostick, H. Ohel, T. Kapish, T. Zuccarelly, O. Perdomo. “Performance of REAL/IX – A Fully Preemptive Real-Time Unix”, *ACM Operating Systems Review,* 23(4), pp. 45-52, October 1989.

[Gil *et al.* 1999] D. Gil, R. Martínez, J. V. Busquets, J. C. Baraza, P. J. Gil, “Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System”, in *Proc. 3rd European Dependable Computing Conf. (EDCC-3),* (Prague, Czech Republic), LNCS, 1667, pp.191-208, Springer, 1999.

[Goyal *et al.* 1986] A. Goyal, W. C. Carter, E. de Souza e Silva and S. S. Lavenberg, “The System Availability Estimator”, in *Proc. 16th IEEE Int Symp. on Fault-Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.84-9, 1986.

[Gray 1986] J. Gray, “Why Do Computers Stop and What Can Be Done About It?” in *Proc. 5th Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS-5),* (Los Angeles, CA, USA), pp.3-12, IEEE CS Press, 1986.

[Gray 1990] J. Gray, “A Census of Tandem System Availability Between 1985 and 1990”, IEEE Trans. on Reliability, R-39 (4), pp.409-18, 1990.

[Halang 2000] W. A. Halang, R. G. Matja_ Colnari_, M. Dru_ovec. “Measuring the Performance of Real-Time Systems”. *The Int. Journal of Time-Critical Computing Systems,* 18, pp. 59-68, 2000.

[Hansen & Siewiorek 1992] J. P. Hansen and D. P. Siewiorek, “Models of Time Coalescence in Event Logs”, in *Proc. 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22),* (Boston, MA, USA), pp.221-7, IEEE CS Press,1992.

[Harnedy 1998] S. Harnedy, *Total SNMP: Exploring the Simple Network Management Protocol,* Prentice Hall PTR, 1998.

[Hiller *et al.* 2001] M. Hiller, A. Jhumka and N. Suri, “An Approach for Analysing the Propagation of Data Errors in Software”, in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), IEEE CS Press, 2001.

[Howard 1971] R. A. Howard, *Dynamic Probabilistic Systems,* vol. II, Wiley & Sons, 1971.

[Iyer & Rossetti 1985] R. K. Iyer and D. J. Rossetti, “Effect of System Workload on Operating System Reliability: A Study on IBM 3081”, *IEEE Trans. on Software Engineering,* SE-11 (12), pp.1438-48, December 1985.

[Iyer & Tang 1996] R. K. Iyer and D. Tang, “Experimental Analysis of Computer System Dependability”, *Fault-Tolerant Computer System Design (D. K. Pradhan, Ed.),* pp.282-392 (Chapter 5), Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996.

[Iyer & Velardi 1985] R. K. Iyer and P. Velardi, “Hardware-Related Software Errors: Measurement and Analysis”, *IEEE Trans. on Software Engineering,* SE-11 (2), pp.223-31, 1985.

[Iyer *et al.* 1986] R. K. Iyer, L. T. Young and V. Sridhar, “Recognition of Error Symptoms in Large Systems”, in *Proc. 1986 IEEE/ACM Fall Joint Computer Conference,* (Dallas, TX, USA), pp.797-806, IEEE-ACM, 1986.

[Jenn *et al.* 1995] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson and J. Karlsson, “Fault Injection into VHDL Models: The MEFISTO Tool”, in *Predictably Dependable Computing Systems* (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), pp.329-46, Springer, Berlin, Germany, 1995.

[Kaâniche *et al.* 1990] M. Kaâniche, K. Kanoun and S. Metge, “Failure Analysis and Validation of a Telecommunication Equipment Software System”, *Annales des Telecommunications,* 45 (11-12), pp.657-70, 1990.

[Kaâniche *et al.* 1994] M. Kaâniche, K. Kanoun, M. Cukier and M. Bastos Martini, “Software Reliability Analysis of Three Successive Generations of a Switching System”, in *Proc. First European Conference on Dependable Computing (EDCC-1),* (D. H. K. Echtle, D. Powell, Ed.), (Berlin, Germany), Lecture Notes in Computer Science, 852, pp.473-90, Springer-Verlag, 1994.

[Kaâniche *et al.* 1998] M. Kaâniche, L. Romano, Z. Kalbarczyk, R. K. Iyer and R. Karcich, “A Hierarchical Approach for Dependability Analysis of a Commercial Cache-Bsaed RAID Storage Architecture”, in *Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28),* (Munich, Germany), pp.6-15, IEEE CS Press, 1998.

[Kalyanakrishnam *et al.* 1999] M. Kalyanakrishnam, Z. Kalbarczyk and R. K. Iyer, “Failure Data Analysis of LAN of Windows NT Based Computers”, in *Proc. 18th Int. Symp. on Reliable Distributed Systems (SRDS'99),* (Lausanne, Switzerland), pp.178-87, IEEE CS Press, 1999.

[Kanawati *et al.* 1995] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, “FERRARI: A Flexible Software-Based Fault and Error Injection System”, *IEEE Trans. on Computers,* 44 (2), pp.248-60, February 1995.

[Kanoun & Borrel 1996] K. Kanoun and M. Borrel, “Dependability of Fault-Tolerant Systems - Explicit Modeling of the Interactions between Hardware and Software Components”, in *Proc. IEEE International Computer Performance & Dependability Symp. (IPDS'96),* (Urbana-Champaign, IL, USA), pp.252-61, IEEE CS Press, 1996.

[Kanoun & Laprie 1996] K. Kanoun and J.-C. Laprie, “Trend Analysis”, in *Handbook of Software Reliability Engineering* (M. Lyu, Ed.), pp.401-37 (Chapter 10), McGraw Hill, 1996.

[Kanoun & Sabourin 1987] K. Kanoun and T. Sabourin, “Software Dependability of a Telephone Switching System”, in *Proc. 17th IEEE Int Symp. on Fault-Tolerant Computing (FTCS-17),* (Pittsburgh, PA, USA), pp.236-41, IEEE CS Press, 1987.

[Kanoun *et al.* 1991] K. Kanoun, J. Arlat, L. Burrill, Y. Crouzet, S. Graf, E. Martins, A. MacInnes, D. Powell, J.-L. Richier and J. Voiron, “Validation”, in *Delta-4: A Generic Architecture for Dependable Distributed Computing* (D. Powell, Ed.), pp.371-406, Springer-Verlag, Berlin, Germany, 1991.

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

[Kanoun *et al*. 1999] K. Kanoun, M. Borrel, T. Moreteveille and A. Peytavin, “Modeling the Dependability of CAUTRA, a Subset of the French Air Traffic Control System”, *IEEE Trans. on Computers,* 48 (5), pp.528-35, 1999.

[Kao & Iyer 1995] W.-L. Kao and R. K. Iyer, “DEFINE: A Distributed Fault Injection and Monitoring Environment”, in *Fault-Tolerant Parallel and Distributed Systems* (D. Pradhan and D. R. Avresky, Eds.), pp.252-9, IEEE CS Press, 1995.

[Kao *et al.* 1993] W.-L. Kao, R. K. Iyer and D. Tang, “FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults”, *IEEE Trans. on Software Engineering,* 19 (11), pp.1105-18, November 1993.

[Kar 1989] K. P. Kar, K. Porter. “Rhealstone – A Real-Time Benchmarking Proposal”. *Dr. Dobbs Journal*, 14(2), pp. 14-24, February 1989.

[Kar 1990] R. P. Kar. “Implementing the Rhealstone Real-Time Benchmarking”. *Dr. Dobb’s Journal*, 15(4), pp. 46-104, April 1990.

[Karlsson et al. 1998] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber and J. Reisinger, “Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture”, in Dependable Computing for Critical Applications (Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-5, Urbana-Champaign, IL, USA, September 1995) (R. K. Iyer, M. Morganti, W. K. Fuchs and V. Gligor, Eds.), Dependable Computing and Fault-Tolerant Systems, 10, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.267-87, IEEE CS Press, 1998.

[Kendall 1977] M. G. Kendall, *The Advanced Theory of Statistics*, Oxford University Press, 1977.

[Kenney & Vouk 1992] G. Q. Kenney and M. A. Vouk, “Measuring the Field Quality of Wide-Distribution Commercial Software”, in *Proc. 3rd IEEE Int. Symp. on Software Reliability Engineering (ISSRE'92),* (Raleigh, NC, USA), pp.351-7, IEEE CS Press, 1992.

[Kimball 1998] R. Kimbal, L. Reeves, M. Ross and W. Thornthwalte, *The Data Warehouse Lifecycle Toolkit,* Ed. J. Wiley & Sons, Inc, 1998

[Koopman & DeVale 1999] P. Koopman and J. DeVale, “Comparing the Robustness of POSIX Operating Systems”, in *Proc. 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29),* (Madison, WI, USA), pp.30-7, IEEE CS Press, 1999.

[Koopman *et al.* 1997] P. J. Koopman, J. Sung, C. Dingman, D. P. Siewiorek and T. Marz, “Comparing Operating Systems using Robustness Benchmarks”, in *Proc. 16th Int. Symp. on Reliable Distributed Systems (SRDS-16),* (Durham, NC, USA), pp.72-9, IEEE CS Press, 1997.

[Labovitz *et al.* 1999] C. Labovitz, A. Ahuja and F. Jahanian, “Experimental Study of Internet Stability and Backbone Failures”, in *Proc. 29th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-29),* (Madison, WI, USA), pp.278-85, IEEE CS Press, 1999.

[Laprie 1995] J.-C. Laprie, “Dependable Computing: Concepts, Limits, Calenges”, in *Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.42-53, IEEE CS Press, 1995.

[Lee et *al.* 1991] I. Lee, R. K. Iyer and D. Tang, “Error/Failure Analysis Using Event Logs from Fault Tolerant Systems”, in *Proc. 21st Int. Symp. on Fault Tolerant Computing (FTCS-21),* (Montreal, Canada), pp.10-7, IEEE CS Press,1991.

[Levendel 1990] Y. Levendel, “Reliability Analysis of Large Software Systems: Defects Data Modeling”, *IEEE Trans. on Software Engineering,* SE-16 (2), pp.141-52, February 1990.

[Levy 2001] M. Levy. “Analyzing Processor, Tool Chain, and RTOS Performance to Get the Best Solution for your Embedded Application”, *Embedded Systems Conference,* no. 468, San Francisco, 2001.

[Lyu 1995] M. R. Lyu (Ed.), *Handbook of Software Reliability Engineering,* McGraw-Hill, 1995.

[Madeira *et al.* 1994] H. Madeira, M. Rela, F. Moreira and J. G. Silva, “RIFLE: A General Purpose Pin-level Fault Injector”, in *Proc. 1st European Dependable Computing Conf. (EDCC-1),* (K. Echtle, D. Hammer and D. Powell, Eds.), (Berlin, Germany), Lecture Notes in Computer Science, 852, pp.199-216, Springer-Verlag, 1994.

[Madeira *et al.* 2000] H. Madeira, D. Costa and M. Vieira, “On the Emulation of Software Faults by Software Fault Injection”, in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.417-26, IEEE CS Press, 2000.

[Marsden & Fabre 2001] E. Marsden and J.-C. Fabre, “Failure Mode Analysis of CORBA Service Implementations”, in *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'2001),* (Heidelberg, Germany), 2001. To appear - Also LAAS Report, May 2001.

[Martínez *et al.* 1999] R. J. Martínez, P. J. Gil, G. Martín, C. Pérez and J. J. Serrano, “Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection”, in *Dependable Computing for Critical Applications (Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7, San Jose, CA, USA, January 1999)* (C. B. Weinstock and J. Rushby, Eds.), Dependable Computing and Fault-Tolerant Systems, 12, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.249-65, IEEE CS Press, San Jose, CA, USA, 1999.

[Martins *et al.* 1990] E. Martins, J. Arlat, Y. Crouzet, J.-C. Fabre and D. Powell, “Testing Multipeer Protocols in the Presence of Faults”, in *Protocol Test Systems* (J. de Meer, L. Mackert and W. Effelsberg, Eds.), pp.77-91, Elsevier (North-Holland), 1990.

[McConnel *et al.* 1979] S. R. McConnel, D. P. Siewiorek and M. M. Tsao, “The Measurement and Analysis of Transient Errors in Digital Computer Systems”, in *Proc. 9th Int. Symp. on Fault-Tolerant Computing (FTCS-9),* (Madison, Wisconsin,), pp.67-70, IEEE CS Press,1979.

[McVoy & Staelin 1996] L. McVoy and C. Staelin, “lmbench: Portable Tools for Performance Analysis”, in *Proc. USENIX 1996 Technical Conference,* pp. 279-294, San Diego, CA, January 1996.

[Meyer & Sanders 1993] J. F. Meyer and W. H. Sanders, “Specification and Construction of Performability Models”, in *Int. Workshop on Performability Modeling of Computer and Communication Systems,* (Mont Saint Michel, France), pp.1-32, 1993.

[Miller *et al.* 1995] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan and J. Steidl, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services,* University of Wisconsin, USA, Research Report, N°CS-TR-95-1268, April 1995.

[Moran *et al.* 1990] P. Moran, P. Gaffney, J. Melody, M. Condon and M. Hayden, "System Availability Monitoring", *IEEE Trans. on Reliability,* R-39 (4), pp.480-5, 1990.

[Mukherjee & Siewiorek 1997] A. Mukherjee and D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Trans. of Software Engineering,* 23 (6) 1997.

[Muppala *et al.* 1992] J. K. Muppala, A. Sathaye, R. Howe, C and K. S. Trivedi, "Dependability Modeling of a Heterogeneous VAX-cluster System Using Stochastic Reward Nets", *in Hardware and Software Fault Tolerance in Parallel Computing Systems* (D. R. Avresky, Ed.), pp.33-59, 1992.

[Murphy 2000] B. Murphy, "Windows 2000 Dependability", in *Workshop on Dependable Networks and Operating Systems, at the Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.D20-D8, 2000.

[Musa *et al.* 1987] J. Musa, A. Iannino and K. Okumoto, *Software Reliability: Measurement, Prediction, Application,* Computer Science Series, 621p., McGraw-Hill, New-York, 1987.

[MVISTA] http://mvista.master.com/texis/master/search/?q=Benchmarks&s=SS

[NIST] www.itl.nist.gov/div897

[Orfali *et al.* 1996] R. Orfali, D. Harkey and J. Edwards, *The Essential Client-Server Survival Guide (Second Edition),* John Wiley & Sons, Inc., 1996.

[Ousterhout 1990] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware*?"*, in *Proc. Summer USENIX Conference*, pp. 247-56, June 1990.

[Pagès & Gondran 1986] A. Pagès and M. Gondran, *System Reliability: Evaluation and Prediction in Engineering,* Springer-Verlag, New York, USA, 1986.

[Pan *et al.* 2001] J. Pan, P. J. Koopman, D. P. Siewiorek, Y. Huang, R. Gruber and M. L. Jiang, "Robustness Testing and Hardening of CORBA ORB Implementations", in *Proc. 2001 Int. Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), pp.141-50, IEEE CS Press, 2001.

[PHILPS] Philips's original benchmark "XA benchmark versus the architectures 68000, 80C196, and 80C51", Results of Philips' benchmark "XA benchmark vs. the MCS251", http://www.semiconductors.philips.com/cgi-bin/searchcat?cat=3999&code=9

[Ponder 1991] C. Ponder. "Performance Variation Across Benchmark Suites", *Computer Architectures News*, pp.30-6, June 1991.

[POSIX] www.itl.nist.gov/div897/ctg/posix_form.htm.

[Powell 1994] D. Powell, "Distributed Fault-Tolerance — Lessons from Delta-4", *IEEE Micro,* 14 (1), pp.36-47, February 1994.

[Powell *et al.* 1995] D. Powell, E. Martins, J. Arlat and Y. Crouzet, “Estimators for Fault Tolerance Coverage Evaluation”, *IEEE Trans. on Computers,* 44 (2), pp.261-74, February 1995.

[Rabah & Kanoun 1999] M. Rabah and K. Kanoun, “Dependability Evaluation of a Distributed Shared Memory Multiprocessor System”, in *Proc. 3rd European Dependable Computing Conference (EDCC-3),* (A. Pataricza, J. Hlavicka and E. Maehle, Eds.), (Prague, Czech Republic), pp.42-59, Springer, 1999.

[Rabbat 1988a] G. Rabbat, B. Furht, R. Kibler. “Three-dimensional computer performance”. *IEEE Computer,* 21(7), pp. 59-60, July 1988.

[Rabbat 1988b] G. Rabbat, B. Furht, R. Kibler. “Three-dimensional Computer and Measuring Their Performance”, *ACM Computer Architecture News*, 16(3), pp. 9-16, June 1988.

[Reibman & Veeraraghavan 1991] A. Reibman and M. Veeraraghavan, “Reliability Modeling: An Overview for System Designers”, *IEEE Computer,* April, pp.49-57, 1991.

[Rennels & Avizienis 1973] D. A. Rennels and A. Avizienis, “RMS: A Reliability Modeling System for Self-Repairing Computers”, in *Proc. 3rd Int. Symp. on Fault-Tolerant Computing (FTCS-3),* (Palo Alto, CA, USA), pp.131-5, IEEE CS Press, 1973.

[Rodríguez *et al.* 1999] M. Rodríguez, F. Salles, J.-C. Fabre and J. Arlat, “MAFALDA: Microkernel Assessment by Fault Injection and Design Aid”, in *Proc. 3rd European Dependable Computing Conf. (EDCC-3),* (E. M. J. Hlavicka, A. Pataricza, Ed.), (Prague, Czech Republic), LNCS, 1667, pp.143-60, Springer, 1999.

[Rodríguez *et al.* 2001] M. Rodríguez, J.-C. Fabre and J. Arlat, *Dependability Assessment of Real-Time Systems*, LAAS-CNRS, Research Report, N°01-189, May 2001.

[Rojas 1996] I. Rojas, “Compositional Construction of SWN Models”, *The Computer Journal*, 38 (7), pp.612-21, 1996.

[Roussopoulos 1998] N. Roussopoulos, “Materialized Views and Data Warehouses”, SIGMOD Record 27(1): 21-26 (1998).

[Sahner & Trivedi 1987] R. A. Sahner and K. S. Trivedi, “Reliability Modeling Using SHARPE”, IEEE Trans. on Reliability, R-36 (2), pp.186-93, June 1987.

[Salles *et al.* 1999] F. Salles, M. Rodríguez, J.-C. Fabre and J. Arlat, “Metakernels and Fault Containment Wrappers”, in *Proc. 29th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-29),* (Madison, WI, USA), pp.22-9, IEEE CS Press, 1999.

[Salzberg 1996] B. Salzberg, “Access Methods”, *ACM Computing Surveys,* vol. 28, no. 1, pp.117-20, 1996.

[Sanders *et al*. 1995] W. H. Sanders, W. D. Obal II, M. A. Qureshi and F. K. Widjanarko, “The UltraSAN Modeling Environment”, Performance Evaluation, 21 (special "Performance Evaluation Tools") 1995.

[Santonja *et al.* 1996] V. Santonja, M. Alonso, J. Molero, J. J. Serrano, P. Gil and R. Ors, “Dependability Models of RAID Using Stochastic Activity Networks”, in *Proc. 2nd European Dependable Computing Conference (EDCC-2),* (Taormina, Italy), pp.141-58, Springer, 1996.

[Shelton *et al.* 2000] C. Shelton, P. Koopman and K. D. Vale, "Robustness Testing of the Microsoft Win32 API", in *Proc. Int. Conference on Dependable Systems and Networks (DSN'2000),* (New York, NY, USA), IEEE CS Press, 2000.

[Sieh et al. 1997] V. Sieh, O. Tschäche and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions", in Proc. 27th Int. Symp. on Fault-Tolerant Computing (FTCS-27), (Seattle, WA, USA), pp.32-6, IEEE CS Press, 1997.

[SIEMENS] www.siemens.com.

[Siewiorek & Swarz 1992] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems - Design and Evaluation,* Digital Press, Bedford, MA, USA, 1992.

[Siewiorek *et al*. 1978] D. P. Siewiorek, V. Kini, H. Mashburn, S. R. McConnel and M. M. Tsao, "A Case Study of C.mmp, Cm*, and C.vmp: Part I—Experience with Fault Tolerance in Multiprocessor Systems", *Proceedings of the IEEE,* 66 (10), pp.1178-99, 1978.

[Siewiorek *et al.* 1993] D. P. Siewiorek, J. J. Hudak, B.-H. Suh and Z. Segall, "Development of a Benchmark to Measure System Robustness", in *Proc. 23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.88-97, IEEE CS Press, 1993.

[Smith *et al.* 1996] D. T. Smith, B. W. Johnson and J. A. Profeta III, "System Dependability Evaluation via a Fault List Generation Algorithm", *IEEE Trans. on Computers,* 45 (8), pp.418-24, August 1996.

[SPEC] www.spec.org.

[Sullivan & Chillarege 1992] M. Sullivan and R. Chillarege, "A Comparison of Software Defects in Database Management Systems and Operating Systems", in *Proc. 22nd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-22),* (Boston, MA, USA), pp.475-84, 1992.

[Tang *et al*. 1990] D. Tang, R. K. Iyer and S. Subramani, "Failure Analysis and Modeling of a VAXcluster System", in *Proc. 20th Int. Symp. on Fault Tolerant Computing (FTCS-20),* (Newcastle Upon Tyne, UK), pp.244-51, IEEE CS Press,1990.

[Thakur & Iyer 1996] A. Thakur and R. K. Iyer, "Analyze-NOW — An Environment for Collection & Analysis of Failures in a Network of Workstations", *IEEE Trans. on Reliability,* 45 (4), pp.561-70, 1996.

[Thévenod-Fosse *et al*. 1995] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "Software Statistical Testing", in *Predictably Dependable Computing Systems* (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), pp.253-72, Springer, Berlin, Germany, 1995.

[TPC] www.tpc.org.

[Trivedi *et al.* 1994] K. S. Trivedi, B. R. Haverkort, A. Rindos and V. Mainkar, "Methods and Tools for Reliability and Performability: Problems and Perspectives", in *Proc. 7th Int'l Conf. on Techniques and Tools for Computer Performance Evaluation* (G. Haring and G. Kotsis, Eds.), Lecture Notes in Computer Science, 794, pp.1-24, Springer-Verlag, Vienna, Austria, 1994.

[Tsai & Singh 2000] T. Tsai and N. Singh, "Reliability Testing of Applications on Windows NT", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.427-36, IEEE CS Press, 2000.

[Tsao & Siewiorek 1983] T. Tsao, M. and D. P. Siewiorek, "Trend Analysis on System Error Files", in *Proc. 13th Int. Symp. on Fault-Tolerant Computing (FTCS-13),* (Milano, Italy), pp.116-9, IEEE CS Press,1983.

[Turley 1999] J. Turley, "Microprocessors for Consumer Electronics, PDAs, and Communications", *Embedded System Conference,* no.424, September 1999.

[Vanada 1984] J. Vanada, *Vanada Benchmark,* Sampson Computer Consulting Inc., Pleasant Valley, Iowa, 1984.

[Voas & McGraw 1998] J. M. Voas and G. McGraw, *Software Fault Injection,* 353p., Wiley Computer Publishing, New York, 1998.

[Weiderman 1989] N. H. Weiderman, "Hardstone: Synthetic Benchmark Requirements for Hard Real-Time Applications", *Technical Report CMU/SEI-89-23*, Carnegie Mellon University, USA, 1989.

[Weiderman 1992] N. H. Weiderman, N. I. Kamenoff, "Hardstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems", *The Journal of Real-Time Systems,* 4, pp. 353-382, Kluwer Publishers, 1992.

[Wein & Sathaye 1990] A. S. Wein and A. Sathaye, "Validating Computer System Availability Models", *IEEE Trans. on Reliability,* 39 (4), pp.468-79, 1990.

[Weiss 1999] A. R. Weiss, R. Clucas, "The Standardisation of Embedded Benchmarking: The pitfalls and the Opportunities", *Embedded Systems Conference,* no. 411, Spring 1999.

[Wells 1993] G. Wells. "A Comparison of Four Microcomputer Operating Systems", *Journal of Real-Time Systems* 5, pp. 345-368, 1993.

[Wood 1995] A. Wood, "Predicting Client/Server Availability", *Computer* (April), pp.41-8, 1995.

[Xu et al. 2001] J. Xu, S. Chen, Z. Kalbarczyk and R. K. Iyer, "An Experimental Study of Security Vulnerabilities Caused by Errors", in Proc. 2001 Int. Conference on Dependable Systems and Networks (DSN-2001), (Göteborg, Sweden), IEEE CS Press, 2001.

[Yount & Siewiorek 1996] C. R. Yount and D. P. Siewiorek, "A Methodology for the Rapid Injection of Transient Hardware Errors", *IEEE Trans. on Computers,* 45 (8), pp.881-91, August 1996.