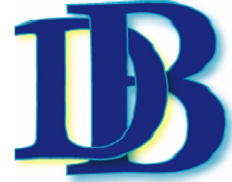


Dependability Benchmarking

IST-2000-25425

Description of the Selected Enabling Technologies

Report Version: Deliverable ETIE4

Report Preparation Date: June 2002 / Revision December 2002

Classification: Public Circulation

Contract Start Date: 1 January 2001

Duration: 36m

Project Co-ordinator: LAAS-CNRS (France)

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

Sponsor: Microsoft (UK)

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

The DBench Framework for Dependability Benchmarking[1]

J. Arlat, K. Kanoun, Y. Crouzet — LAAS-CNRS, Toulouse, France
H. Madeira — Univ. of Coimbra, Portugal
M. Dal Cin — FAU, Erlangen, Germany
P. Gil — UPV, Valencia, Spain
D. Costa — Critical Software, Coimbra, Portugal

Abstract

The goal of dependability benchmarking is to provide generic ways for characterizing the behavior of components and computer systems in the presence of faults, allowing for the quantification of dependability measures. Beyond existing evaluation techniques, dependability benchmarking must provide a reproducible and cost-effective way of performing such a characterization, especially for comparative purposes. This paper proposes a framework for defining dependability benchmarks for computer systems, with particular emphasis on COTS and COTS-based systems. The multiple dimensions of the problem are discussed and the framework is presented through a set of well-defined dimensions. Examples of concrete benchmark set-ups for selected operating systems, transactional systems and embedded systems are presented and discussed.

Keywords: Dependability Characterization, Dependability Assessment, Benchmarking, COTS components and COTS-based Systems.

1 Introduction

The goal of benchmarking the dependability of computer systems is to provide generic and reproducible ways for characterizing their behavior in the presence of faults. The key aspect that distinguishes benchmarking from existing evaluation and validation techniques is that a benchmark represents an agreement that is widely accepted both by the computer industry and by the user community. This technical agreement should state the system that is benchmarked, the measures, the way and conditions under which these measures are obtained, and the domain of validity. The objective of a benchmark is to provide practical ways to characterize the dependability of computers.

The success of well-established performance benchmarks in comparing performance of components and computer systems probably accounts for the generalized idea that the main goal of benchmarks is to compare systems on the basis of benchmark results. However, we would like to emphasize that dependability benchmarks can be used in a variety of cases by both the end-users and manufacturers to:

- *Characterize* the dependability of a component or a system.
- *Identify* weak parts of a system, requiring more attention and perhaps needing some improvements by tuning a component to enhance its dependability (e.g., using software wrappers), or by tuning the system architecture (e.g., adding fault tolerance) to ensure a suitable dependability level.
- *Compare* the dependability of alternative or competitive solutions according to one or several dependability attributes.

[1] This work was partially financed by the European Commission, in the framework of the DBench project IST-2000-25425

This paper proposes a framework for defining dependability benchmarks for computer systems, with emphasis on off-the-shelf (OTS), commercial (COTS) or not, and also OTS-based systems, via experimentation and modeling.

Although a variety of dependability evaluation methods and techniques aimed at different system levels and domains, ranging from analytical modeling to simulation and experimental approaches, including those based on fault injection have been reported (e.g., see [1-7]), a generalized approach that can be used to evaluate and compare different systems and applications still does not exist. It is also worth mentioning the pioneering work that addressed the benchmarking of software robustness [8] and investigated the development of some forms of benchmarks [9, 10]. Among these efforts, operating systems received much attention, e.g., see [11-13], including the development of tools (respectively, Ballista, Fuzz and MAFALDA-RT) for robustness testing.

On the other hand, performance benchmarking is now a well-established area that is led by organizations such as the TPC (Transaction Processing Performance Council) and SPEC (Standard Performance Evaluation Corporation), and supported by major companies in the computer industry [14]. Clearly, the preliminary efforts at developing dependability benchmarks, including focused robustness testing techniques, do not benefit yet from such a level of recognition. Much work is still needed despite the many relevant efforts that have tackled in a unified way several issues concerning performance and dependability assessment. Among these, we would like to point out the following studies that addressed jointly a couple of such issues: bringing together performance benchmarking and dependability assessment [15], field measurement and fault injection [16], field measurement and modeling [17], fault injection and modeling [18], and use of standard performance benchmarks as a workload for dependability evaluation [19]. To the best of our knowledge, little has been reported to cover these various facets in a comprehensive manner. Accordingly, this is the path that is specifically being explored in our work. This paper summarizes the main views and guidelines outlined by the DBench consortium (e.g., see [20, 21]). It elaborates on the preliminary insights first reported in [22].

The ultimate objective of our work is to comprehensively define dependability benchmarks for computer systems, and provide means for implementing them. In this paper, we identify the main dimensions that are decisive for defining dependability benchmarks and the way experimentation can be conducted in practice. Three examples of prototype benchmarks are considered to exemplify how the various dimensions of the proposed framework can be actually instantiated: the first one concerns operating systems that are the core components in computer systems; the other two focus on transactional and embedded systems and applications. It is expected that these prototypes, that are currently being implemented and applied, will aid defining exploitable dependability benchmarks, at least for the considered classes of systems.

The paper is structured into five sections. Section 2 presents the relevant dimensions of dependability benchmarking. Section 3 illustrates the application of this framework in the light of the main issues relevant for conducting dependability benchmarks. Section 4 describes and discusses the definition of prototype benchmarks for three distinct categories of targets, namely: operating systems, transactional systems and embedded systems. Finally, Section 5 concludes the paper.

2 Dependability Benchmarking Dimensions

What distinguishes a dependability benchmark with respect to most dependability validation efforts that have been targeting many computer systems, including fault-tolerant systems, is the fact that a benchmark is primarily meant to be used on an open basis and to provide a set of standardized metrics. Accordingly, the *a priori* knowledge of the target systems(s) that is necessary should rely on information that is undisputable (e.g., because it is widely publicized). This puts a great deal of constraints on the type of experiments and analyses that can be actually carried out, especially when COTS systems and components are accounted for. Also, the results obtained are to be made publicly available and the features of the system under benchmarking as well as the benchmark procedures should be unambiguously disclosed.

Such a level of standardization and openness is actually achieved by performance benchmarks. Although, the same objective is to be aimed at for dependability benchmarks, the main specificities induced by the added features that need to be explicitly accounted for (e.g., faultload, error signaling, failure modes, etc.) makes such an achievement a much more difficult and challenging task.

In this context, the definition of a meaningful framework for dependability benchmarking requires first of all a clear understanding of all impacting dimensions. Their investigation is essential to understand the problem space as well as to classify and specify all relevant aspects of dependability benchmarks.

Figure 1 outlines our classification of dimensions. The *categorization* dimensions allow us to organize the dependability benchmark space into well-identified categories. These dimensions describe and specify the target of the benchmark, as well as the benchmarking context. The *measure* dimensions stipulate the dependability benchmarking measure(s) to be assessed according to the choices made for the categorization dimensions. The *experimentation* dimensions include all aspects related to experimentation on the target system to get the base data needed to obtain the selected measure(s) on the benchmark target. In addition to specifying the workload and faultload applied and the related measurements, this dimension has to precisely identify the actual system under benchmarking on which experimentation is to be (or has been) conducted. For example, when the Benchmark Target (BT) is an Operating System (OS), the System Under Benchmarking (SUB) includes the hardware computer necessary for the OS to execute.

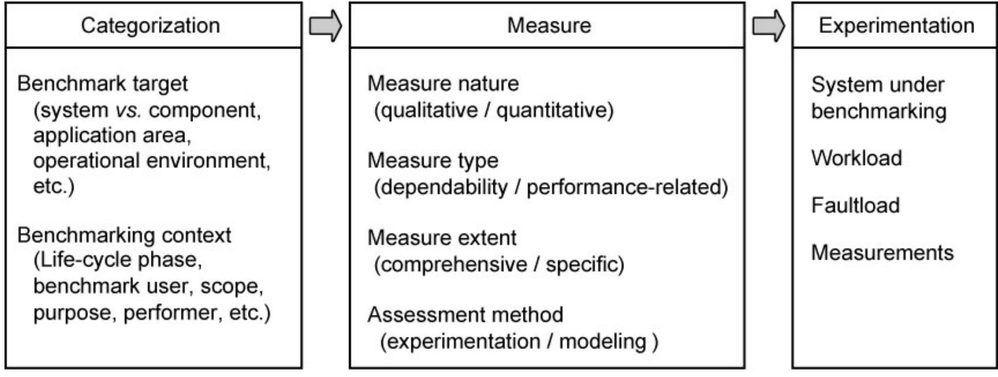


Figure 1 – Dependability benchmarking dimensions

In addition to these dimensions, a dependability benchmark must fulfill a set of properties to be considered valid and useful. For example, it must be reproducible (in statistical terms), must not cause undesirable

interferences or perturbations on the target system, must be portable, cost-effective, etc. These properties represent goals that must be achieved when defining actual dependability benchmarks.

We concentrate hereafter on the three groups of dimensions as depicted in Figure 1.

2.1 Categorization Dimensions

The categorization dimensions impact the basic selection of meaningful benchmark measures as well as the experimentation dimensions.

2.1.1 The Benchmark Target

It is essential to clearly identify the type of system that is the target of the benchmark with respect to its application area and its operating environment.

The *application area* is a key dimension for the definition of dependability benchmarks, as it impacts the operating environment and the selection of suitable benchmark measures. The division of the application spectrum into well-defined application areas is necessary to cope with the diversity of applications. The main difficulty is clearly the establishment of the appropriate granularity to divide the application spectrum. Obviously, different application areas will need different dependability benchmarks. For example, database applications have requirements that are different from those for control applications. The dependability benchmarking prototypes discussed in Section 4 illustrate this point.

The *operating environment* characterizes the environment in which the system is usually operated. It encompasses many important aspects ranging from the functional activity to the faults to which the target is being subjected. In particular, this set of faults is an essential facet as it clearly marks a distinctive feature for a dependability benchmark that goes beyond what it usually accounted for by a performance benchmark. This is even more difficult when including human-related interaction faults (such as operator mistakes or intrusions). Embedded control systems are usually more sensitive to physical faults that affect the underlying hardware platform while it is more likely that transactional systems are also impacted by operator errors.

2.1.2 Benchmarking Context

Performing a dependability benchmark and using the benchmark results can be done with multiple perspectives. The benchmarking context is, in fact, a composite dimension including the following:

Life Cycle Phase - A dependability benchmark can be performed in any phase of the system life cycle. The measures obtained for a specific phase are then useful for the current or subsequent phases.

Benchmark user - Person or entity actually using the benchmark results.

Benchmark purpose - The purpose of a benchmark may vary significantly along the system life-cycle. For example, during the very early phases, a dependability benchmark could be useful to reveal weak points and thus could help support the decision whether to purchase a particular computer platform that is to be integrated into a critical application. During development, results could be used to monitor the improvement actually achieved by fault removal activities (regression testing). With respect to operational life, benchmark results could be useful to evaluate the impact of physical faults or operator faults on system dependability. Results of a standardized benchmark are used ultimately to compare alternative systems. The primary results derived from a benchmark are two fold; they can be aimed at i) either characterizing system dependability capabilities

in a qualitative manner (e.g., on the basis of the dependability features supported or claimed, such as on-line error detection, fail-silent failure mode, etc.), ii) or assessing quantitatively these properties.

Benchmark performer - Person or entity actually performing the benchmark (e.g., manufacturer, integrator, third party, end-user). These entities have i) different visions of the target system, ii) distinct accessibility and observability levels for experimentation and iii) different expectations from the measures.

2.2 Measure Dimensions

Dependability benchmarks encompass measures assessed under particular conditions. Therefore an important task in the benchmark definition concerns the identification of meaningful measures. Such measures allow the characterization of the system in a *qualitative manner* (in terms of features related to the system dependability capabilities and properties) or *quantitatively*.

As the occurrence or activation of faults may lead to performance degradation without leading to system failure, dependability and performance are strongly related. Thus the evaluation of system performance under faulty conditions (referred to as *performance-related measures*) allows for extending the characterization of system behavior from a dependability perspective.

The benchmark target and the benchmarking context impact the type of benchmark measures. Typically, end-users are interested in measures defined with respect to the expected services, referred to as *comprehensive measures*, while manufacturers and integrators could be more interested in *specific measures*. Nevertheless, as was already pointed out, in both cases the benchmark measures should be easily understandable and openly available.

Comprehensive measures characterize the system at the service delivery level, taking into account all events impacting its behavior. Availability and safety are examples of such measures. Evaluation of comprehensive measure may require the use of both experimentation and modeling. It is worth mentioning that this constitutes a new aspect in the context of computer benchmarking, as performance benchmarks rely only on experimental measures.

A specific measure is usually associated to a specific system feature without necessarily taking into account all processes impacting its behavior (e.g., without taking into account failure rates and maintenance duration). Specific measures characterize for example error detection and recovery or fault diagnosis capabilities. For example, two specific measures can be associated to error detection**:** i) *coverage factor* (conditional probability of detecting an error when an error exists) and ii) *latency* (time required to detect an error). Such measures can be evaluated based on experimentation.

Indeed, a comprehensive measure usually combines the impact of several specific measures. The support of analytical models provides an elaborated means to combine the effects of various specific measures (e.g., the coverage factors and latencies associated to distinct error handling mechanisms) with the fault occurrence and maintenance rates that characterize the operating environment of the system. On another hand, simple statistics (e.g., mean values) allow for combining together the various specific measures to obtain comprehensive measures.

The actual set of benchmark measures is very much dependent upon the categorization dimensions. In particular, the failure modes are directly deduced from the service that is fulfilled by / expected from the system in the actual application domain in which it operates (e.g., the impact of the failure of a control

system heavily depends on the nature of the process it controls). This issue will illustrated in the case of the prototype benchmark targeting a control function from the automotive application domain (Section 4.4). More generally, availability and safety measures relate to distinct failure modes. Also, transactional and control applications have usually different constraints with respect to real-time requirements and thus timing failures.

2.3 Experimentation Dimensions

These include all aspects related to experimentation on the target system to obtain the selected measures, i.e., the workload, the faultload and measurements to be performed on the benchmark target. It is also important to clearly identify and specify the set-up that is often necessary to host and run the benchmark target, and perform the series of experiments defined by the benchmark. This is what is being designated as "System Under Benchmarking" in Figure 1. The explicit identification of the type of system under benchmarking is essential as the three other dimensions (workload, faultload and measurements) should be described on the basis of the system under benchmarking.

2.3.1 The System Under Benchmarking

In practice, for the benchmark target (BT) to be assessed, it is often necessary to include it into a wider system, which we call the System Under Benchmarking (SUB). This is especially the case when the BT is a software component (e.g., an operating system or a database management system), as the target component needs the SUB to work properly. The SUB is the equivalent of what is usually termed as System under Test (SUT) in the case of performance benchmarks.

The SUB provides a support (especially, hardware platform and software resources) thanks to which the benchmark target can execute. It might also be the case that for practical reasons, the workload and faultload are applied on the SUB and measurements are made via the SUB.

Accordingly, it is important that the SUB be explicitly documented (e.g., in the form of a disclosure document) so that benchmark measures can be properly interpreted and the benchmarks can be actually reproduced. Moreover, the functional interface(s) between the SUB and the BT should be explicitly identified. This is very much important as for being considered as valid the faultloads should impact the BT via such interfaces. Indeed, for the benchmark being credible, especially from the point of view of the provider of the BT, the BT should not be modified directly by the faultload[2].

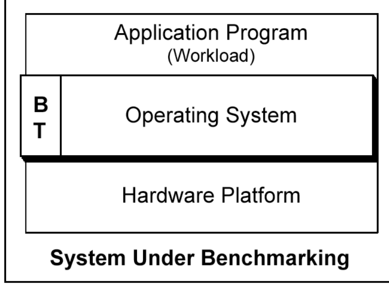


Figure 2 – Example of System Under Benchmarking (SUB) for an OS Benchmark Target (BT)

[2] In particular, in the case of software-intensive COTS-based BTs, mutations of the software part of the BTs are not considered as adequate for two main reasons: i) the mutations would make each BT different from its nominal version, ii) lack of knowledge on the structure and code of COTS software components, it is unlikely that one can match these mutations for the considered BTs.

As an example, Figure 2 illustrates the related notions of BT and SUB in the case when the BT is an operating system. The SUB shown encompasses the hardware platform (as well as the drivers) and the application program layer that is interfaced with the OS via the API (and that constitutes the workload). In such a case, a typical faultload would encompass corrupted system calls (that would simulate the impact of erroneous applications) and underlying hardware/drivers faults.

2.3.2 Workload

The workload represents a typical operational profile for the considered application area. Widely accepted performance benchmarks can provide workloads for many application areas and clearly show that it is possible to agree on an abstract workload that reasonably represents a given application area.

As is often the practice in performance benchmarks, and as was already identified in previous experimental dependability studies (e.g., see [23]), it is often practical to distinguish two forms of workloads: background and foreground. The provision of a *background* worlkload is a practical means to adjust the workload to fit a specific activity profile. A *foreground* workload superimposed to the previous one can be tailored specifically to analyze the impact of faults on the service supported by this particular workload. It is worth pointing out that the foreground workload should provide enhanced observability means with respect to what is usually offered by performance benchmarks (timing measurement) to be able to allow for the validity of the computations performed in presence of faults to be assessed.

As is the case for several performance benchmarks and as was used also in several experimental dependability evaluation studies (e.g., see [13]), the workload can take the form of a modular workload, made up of a set of workload processes, each tailored to sensitize a specific entity of the BT.

In addition, in dependability benchmarking, the concept of workload must be expanded to take into account the impact of preventive and corrective maintenance actions that are always conducted as part of routine operations.

2.3.3 Faultload

The faultload consists of a set of faults and exceptional conditions (including significantly stressful workload) that are intended to emulate the real threats the system would experience. The resources that constitute the benchmark target specify the set of *internal* faults it can be subjected to. Considering that few information is usually available concerning the actual structure (physical and logical) of a COTS-based benchmark target, it is thus likely that internal faults should be represented in the faultload by high-level classes of faults. In this context, equivalent sets of faults for different benchmark targets must be understood on a statistical basis rather than being literally identical. *External faults* clearly depend on the operating environment.

The definition of the faultload is a pragmatic process based on knowledge, observation, and reasoning. Inputs from many different application areas are needed to accomplish this task with success. Some examples are information from failure data reported in the field (e.g., see [24] for software faults), characteristics of the operating environment (for operator faults), or even information from experimental and simulation studies.

The definition of representative faultloads is probably the most difficult part of defining a dependability benchmark. As a first attempt, we advocate a “divide-and-conquer” approach by considering selective

faultloads that address only one class of faults at a time (e.g., hardware, software, operator). At the same time, the fact that practical dependability benchmarks will have to address a specific application area or a specific benchmark target component also simplifies the task of defining the faultload, as this is another way to reduce the fault space that a given faultload should emulate. This leads to the definition of separate *basic* dependability benchmarks each corresponding to a class of faults.

Besides the problem of combining adequately several classes of faults, the representativeness issue is directly related to the practical means that are available to apply the faultload. Indeed, not all faults can be confidently emulated by any fault injection technique! Extensive work has been reported concerning fault representativeness (i.e., the plausibility of the supported fault model with respect to real faults) and the equivalence between fault injection techniques (e.g., see [25], [2], [26, 27]), but often featuring contrasted conclusions. Accordingly, in order to gain further insights on this issue, we have conducted a specific set of analyses and experiments. First of all, it is important to point out that what actually matters is the impact and consequences of an injected fault (i.e., the error propagated) rather than the faults themselves. Second, we have introduced a generic multilevel framework that encompasses the definition of several distances (e.g., *injection* distance: between the levels of injected faults and real faults, *observation* distance: between the level where faults are injected and the level their consequences are observed). This helped identify the important issues involved when conducting such specific experiments [28].

For what concerns hardware faults, it was shown that i) multiple-bit bit-flips are highly likely in processor registers, ii) the software implemented fault injection (SWIFI) technique can emulate bit-flip faults in the programmable registers at zero injection distance (i.e., faults are injected exactly at the same level of the real faults that are supposed to be emulated), and iii) SWIFI can emulate faults in other processor units at a non-zero injection distance. However, in this last case the fact that the injection distance is non-zero imposes several limitations concerning both fault injection accuracy and intrusiveness. For software faults, it was shown that selective mutations performed by SWIFI can emulate software faults [29] and more specifically for what concerns OSs, that it was seldom possible to simulate the erroneous behaviors induced by faulty drivers when injecting at the level of the API [30], and thus more focused approaches were needed [31]. For operator faults, it has been possible to devise a faultload featuring virtually a zero injection distance [32].

2.3.4 Measurements

Measurements performed on the benchmark target allow for the observation of its reaction to the applied workload and faultload. The measures of interest are then obtained from processing these measurements.

Basic measurements include the identification of reported and non-reported failure modes. Reported failure modes are based on events that are explicitly signaled by the benchmark target (e.g., built-in error detection mechanisms and warnings). The exposure of the other failure modes requires some specific means for monitoring the benchmark target: e.g., monitoring of its (non) liveness, in the case of hangs. In practice, for such outcomes to be diagnosed it might even be necessary that the behavior of the benchmark target (results computed and/or control flow) in presence of faults be compared with its nominal behavior (notion of *oracle*).

Another important dimension is related to the assessment of the behavior in the time domain. Although timing related-measurements form the basis for the insights one can get from performance benchmarks

(e.g., number of transactions processed per unit of time), what really matters here is less the absolute time needed to run the workload (or associated rate) than the identification of the impact of the faultload on this execution time (e.g., the ratio between the number of transactions executed in the presence of faults and without faults). Accordingly, timing measurements should also be related to a baseline performance (oracle). In addition to workload execution times, dependability benchmarks should be able to support the measurement of restoration times after the occurrence of a faulty situation before the delivery of nominal service is resumed.

3 Benchmark Conduct

Given the large spectrum of dimensions that need to be taken into account, it is not reasonable to expect to be able to provide dependability benchmarks integrating all dimension choices consistently. As is actually the case in the domain of performance benchmarking, we are rather advocating the development of focused benchmarks. In particular, we are investigating distinct benchmark prototypes for a wide variety of BTs: operating systems, transactional systems, and embedded systems.

Moreover, we believe the shape of dependability benchmarks to be rather a detailed specification including a list of procedures and rules to guide all the processes of *producing benchmark measures*, from experimentation and modeling as well. In particular, a dependability benchmark should include procedures and rules defining the way the specification should be implemented for a given type of BT as well as procedures and rules for conducting experiments and to ensure uniform conditions for measurement.

We are working towards these objectives and we expect that the case studies reported in Section 4 will help us eliciting such specifications, at least for the categories of systems that we are considering. These efforts are aimed at experimenting three sets of benchmark prototypes.

3.1 Benchmarking Steps

Assuming that such specifications do exist at least for some classes of systems, benchmarking can be achieved in several steps forming a *benchmarking scenario*.

As shown on Figure 3, benchmark definition starts by an *analysis step* that is necessary to settle the dimension space (categorization, measure and experimentation). Characterization and assessment of the target system dependability is based on an *experimentation step* according to the selection made among the whole dimension space. If comprehensive measures of the target system are of interest, a *modeling step* is required in addition to experimentation to combine the experimental measures together with the actual processes governing the faultload and the workload. Examples of modeling techniques for evaluating comprehensive measures are: block diagrams, faults trees, Markov chains or stochastic Petri nets.

The various steps and related scenarios combining analysis, modeling and experimentation were described in detail in [22]. In the sequel, we focus our attention on experimentation issues.

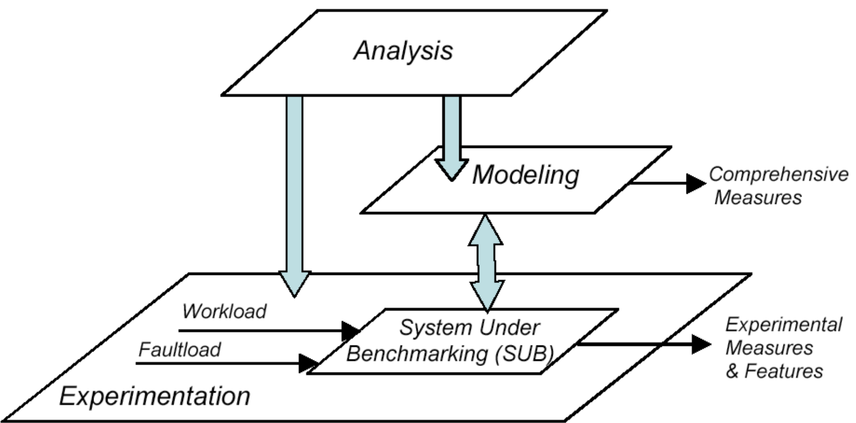


Figure 3 – Dependability benchmarking steps

3.2 Strategies for Performing the Experimentation

In addition to the SUB, some additional resources and instrumentation might be needed in order to perform the dependability benchmark. This is what we are referring to as the Benchmark Management System (BMS). The BMS is part of the configuration needed for performing the dependability benchmark, but it role is limited to facilitate the interaction with the SUB/BT with respect to all experimentation dimensions (workload, faultload and measurements). The BMS is in charge of managing and automating the conduct of the experiments carried out as part of the dependability benchmark. The tasks assigned to the BMS must be clearly defined in the benchmark specification (but are very dependent on the actual benchmark).

In order to illustrate the issues involved, let us discuss the two main strategies that can be identified for conducting the experimentation step:

- Independent strategy: All experiments/runs are carried out independently and the SUB is reset by the BMS after each run in order to purge all residual errors. Each experiment features the application of the workload and of a basic element of the faultload: a single fault is applied (e.g., see [23]).
- Cumulative strategy: A full combination of workload and faultload is applied to the OS for a fixed time interval or for a prescribed number of elementary faults. SUB reboot/restart are only initiated by the BT and are explicitly part of the behaviour that is benchmarked (e.g., see [9]).

We briefly identify hereafter the main differences between these two strategies[3].

The *independent* strategy defines a well-controlled assessment framework. In particular, it allows for a flexible *a posteriori* analysis of the results obtained for each experiment. This way, data grouping for benchmark results can be adjusted by post-processing the outcomes obtained for each experiment according to various facets (e.g., measures can be derived that discriminate the outcomes obtained between the invalid parameter and the erroneous parameters of the considered combined faultload). More generally it allows for additional insights to be derived from the experiments being conducted. Accordingly, this strategy seems more prone to be useful from an integrator viewpoint. The use of deterministic parameters or pseudo-randomly selected parameters (e.g., interval between workload activation and fault injection, etc.) for controlling the variables that determine the experiments (interval between workload activation and fault

[3] These differences can be related to the differences between independent Bernouilli trials (e.g., see [23]) and experiments based on Monte Carlo simulations, that rely on a specific distribution of the faultload (e.g., see [33]).

injection, etc.) is recommended. Such parameters should be part of the disclosures that are to be provided to allow for the benchmark conditions to be confidently reproduced.

The *cumulative* strategy is basically aimed at obtaining a specific experimental measure that encompasses various facets of the behavior of the BT in presence of faults into a *single figure*. It is worth pointing out that this strategy actually pays-off if the BT exhibits sufficient fault tolerance features (i.e., when it is likely that it will sustain unaffected execution in presence of faults); indeed, if this is not the case then this strategy is not very much different from the first one. Such a strategy provides a practical means to obtain a measurement of the impact of the faultload on the time taken by the BT to restore its state (and of the SUB as well, if applicable). As an example of a pragmatic measure in this category one can consider the number of successive faults being applied before the BT has crashed. Here as well, the synchronization of the faultload with the workload and the ordering of the faults being applied are major characteristics of the benchmark, and as such, should be explicitly revealed and by all means maintained unchanged for reproducibility. However, it is worth noting that the results obtained this way are very much dependent of the way the workload and faultload are applied and combined together and especially how they fit the application area of interest for the end-user. Indeed, in this case, the selective analysis of the impact of each of the faults applied is much more difficult and in many cases impossible. Accordingly, such a strategy is much less attractive from an integrator point view.

From a practical point of view, it is worth noting that the first strategy has been widely used in the case of the experimental evaluation of fault-tolerant systems and can be extended to include the "restoration times" that separate two consecutive runs in the measurements to be considered as part of the benchmark. On the other hand, the second strategy is more prone to avoid the cases of "no reply" or "non-significant experiments" that are often encountered in fault injection experiments.

At this stage, the benchmarking activities conducted so far are essentially based on the independent strategy. Nevertheless, we plan to investigate the potential of the cumulative strategy during the experiments and tests we are currently carrying out to implement the benchmark prototypes defined in Section 4.

3.3 Procedures and Rules

Procedures and rules are needed to control the way a dependability benchmark is applied and used. These are of course dependent of each specific benchmark, nevertheless the following items give some guidelines that are relevant for most cases:

- Standardized procedures for "translating" the workload and faultload defined in the benchmark specification for a specific BT into the actual workload and faultload that is applied via the SUB.
- Scaling rules to adapt the same benchmark to systems of varying sizes. These scaling rules would define the way the system load can be changed. At first sight, the scaling rules are related to baseline performance measures and should mainly affect the workload, but one task of the experimental research planned for the prototypes is to investigate the need of scaling up or down other components of the dependability benchmark to make it possible to use the same benchmark in systems of quite different sizes.

- Rules related to the collection of the measurements. For example, these rules may include allowances for specific instrumentation and degree of interference, as well as common references and precision for timing measures, signaling of "non-significant" experiments, etc.
- Rules for the production of the results (experimental results and/or comprehensive measures) from the elementary measurements, such as calculation formulas, ways to deal with uncertainties, errors and confidence intervals for possible statistical measures, etc.
- System configuration disclosures required to fairly interpret the measures provided by a dependability benchmark. This encompasses disclosures on the SUB and on the BMS.

4 Definition of benchmark prototypes

In this section, we illustrate how the general concepts presented in the previous sections can be applied to the definition of preliminary benchmarks. To study a representative set of benchmarks, thus allowing for the investigation of a substantially distinct range of benchmark dimensions, we are considering three categories of targets (BTs): i) *general purpose operating systems*, ii) *OLTP systems* and iii) *embedded systems*.

Two distinct sets of architectures are addressed for OLTP systems: a classical OLTP system and a web server system. In both cases, the target of the benchmark is the OLTP system. However, these benchmarks will be used mainly to characterize key components of an OLTP system, such as the Data Base Management System (DBMS) and the web server. Concerning embedded systems, two classes of application domains are being considered (space and automotive). However, for sake of conciseness, we restrict the presentation here to the automotive application, more precisely a diesel engine electronic control unit.

Although these BTs are (purposely) quite distinct, the benchmark prototypes we are considering share some common aspects among the characterization dimensions:

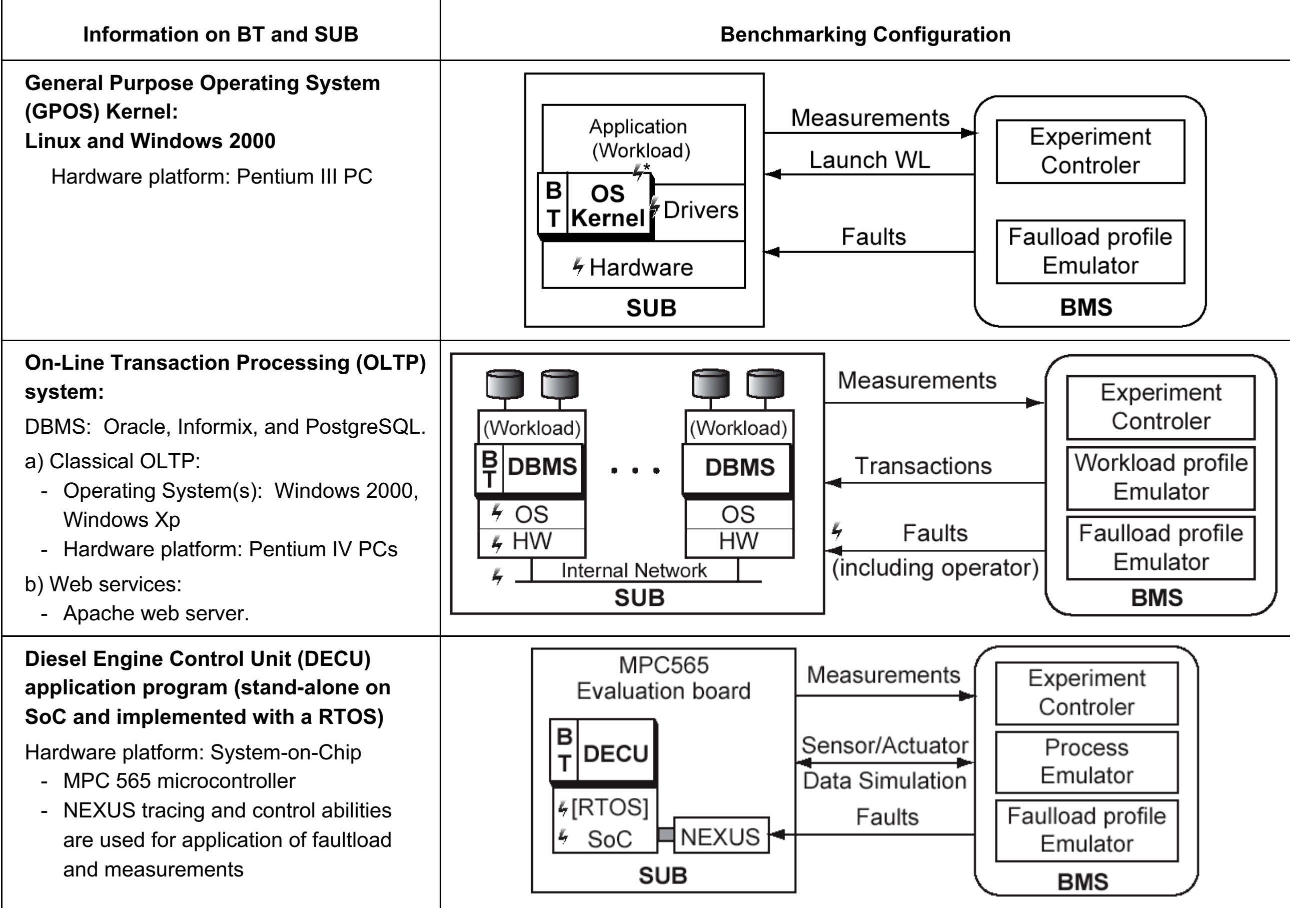
- Benchmark Target: In all cases the benchmark target is always either an *Off-the-Shelf component*, either commercial (COTS) or Open System Software or a *system including at least one such component*.
- Life cycle phase: It is assumed that benchmark is being performed during the *integration phase* of a system including the COTS BT or when the system is available for *operational phase*.
- Benchmark user: The primary users are the *integrators of the system including the BT* or the *end-users* of the BT, as it is assumed that the benchmark results are to be standardized so that they can be made publicly available. However, some results may be of interest to the developer(s) of the BT component, for improving its dependability, should the benchmark reveal some deficiencies.
- Benchmark purpose: For all target systems, the following possible purposes are identified: i) assess some dependability features, ii) assess dependability (and performance) related measures, and iii) identify potential weaknesses.
- Benchmark performer: We consider that the benchmark performer is someone (or an entity) who has no in depth knowledge about the BT and who is aiming at i) improving significantly her/his knowledge about its dependability features, and ii) publicizing information on the BT dependability in a standardized way.

In the sequel, for each benchmark prototype, we describe and discuss the main experimentation dimensions.

4.1 The considered BTs, SUBs and associated benchmarking set-ups

Table 1 presents the main features of the benchmark targets, systems under benchmarking, and the respective set-ups considered for each of the prototype benchmarks. The first column provides some details on the BT and on the SUB. The second one identifies how the BT and supporting resources are included in the SUB. It also illustrates the interconnection between the SUB and the BMS. The benchmarking configurations also give some information about the workload and about the location where the faultload is applied. Both of these issues will be addressed further in the subsequent sections.

Table 1 – Benchmark targets and benchmarking configurations



Information on BT and SUB	Benchmarking Configuration
General Purpose Operating System (GPOS) Kernel: **Linux and Windows 2000** Hardware platform: Pentium III PC	
On-Line Transaction Processing (OLTP) system: DBMS: Oracle, Informix, and PostgreSQL. a) Classical OLTP: - Operating System(s): Windows 2000, Windows Xp - Hardware platform: Pentium IV PCs b) Web services: - Apache web server.	
Diesel Engine Control Unit (DECU) application program (stand-alone on SoC and implemented with a RTOS) Hardware platform: System-on-Chip - MPC 565 microcontroller - NEXUS tracing and control abilities are used for application of faultload and measurements	

BT = Benchmark Target — **SUB** = System Under Benchmarking — **BMS** = Benchmark Management System

Note: 7 This symbol indicates locations where the faultload is applied.

For **General Purpose Operating System (GPOS)** benchmarks, the BT is the kernel of the OS; i.e., we consider that the drivers are part of the SUB and thus we are explicitly investigating the impact of faultload affecting these drivers.

For the **OLTP** systems, although the main BT is normally the DBMS (and we are using several DBMS such as Oracle8i, Oracle9i, Informix, and PostgreSQL), experiments are also currently being conducted using two variants of OSs (Windows and Linux). The second benchmark corresponding to web services [34]. This second set-up features a database server, four application servers and 16 serial terminals (end-users); an additional database server has been added to serve as a reference (oracle). It is worth noting that the fact that

these experiments make use of Windows and Linux OSs, will allow for cross analysis with the results obtained in the case of the GPOS benchmarks.

For the **Diesel Engine Control Unit (DECU)** benchmarks, two implementations of the control unit are being studied: one directly interfaced with the hardware layer (stand alone) and one running on top of a RTOS. In both cases, the underlying platform relies on the NEXUS[4] standard that offers enhanced observability and controllability features: in particular, it is possible to access memory "on-the-fly".

4.2 Workload

Table 2 summarizes the workloads being considered for the benchmark prototypes.

Table 2 – Workloads for the benchmark prototypes

GPOS	OLTP	DECU
• Modular Workload • Background and Foreground Workload	• TPC-C based Workload (classical OLTP and web services) • SPECweb9 (web services only)	• Simulation of operation of a diesel engine electronic control unit

For the **GPOS** benchmarks, the basic type of workload concerns synthetic workload targeting specific OS services/functions (e.g., scheduling, memory management, synchronization, etc.). The main interest of such an approach is that it allows for a series of modular benchmarks to be developed (each focusing on a specific OS function). The results obtained for each workload can then be analyzed, either independently or on a weighted basis, according to the expected activation of the system in which the OS is to be integrated.

For the **OLTP** benchmarks, we have adopted the workload of the well-established TPC Benchmark™ C (TPC-C). It represents a mix of read-only and update intensive transactions that simulates the activities of most common OLTP environments, including transactions resulting from human operators working on interactive sessions. This workload represents a typical database installation. The business represented by TPC-C is a wholesale supplier having a number of warehouses and their associated sale 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. In addition to possible changes in the TPC-C workload (to fulfill specific dependability benchmarking needs, e.g., for better observability), we also consider to perform experiments using the SPECweb99 as workload in the case of web server applications.

The workload selected for the automotive application (**DECU**) benchmarks is focused on the considered automated function: a simplified version of a diesel engine electronic control unit. Basically, it aims at simulating all possible regulation conditions in the control unit. It features a sequence of inputs (concerning the throttle, the rail prescribed pressure and the intake pressure) that simulates an acceleration followed by a deceleration of the engine.

4.3 Faultload

Table 3 summarizes the main features of the faultloads for the considered benchmark prototypes.

---

[4] Also known as IEEE-ISTO 5001™-1999 (See http://www.ieee-isto.org/Nexus5001/standard2.html).

Table 3 – Faultload for the benchmark prototypes

GPOS	OLTP	DECU
• Software faults (application and/or driver programs) SWIFI • Hardware faults (bit-flips in memory) and ill-timed exceptions	• Operator faults (scripts) • Software faults (educated mutations) • Hardware faults (UMLinux)	• Transient faults affecting the memory elements of the hardware platform • NEXUS-based SWIFI

For **GPOS**, concerning software faults, we have concentrated on API-level faultload. Such a faultload is mainly geared towards the assessment of the OS being benchmarked with respect to errors induced by faulty applications. We are investigating means for applying a faultload emulating OS driver faults. Furthermore, as this is now usual, we also include in the faultload considered in the prototype benchmarks under investigation the case of bit-flips in the kernel memory space, which are more related to hardware induced errors.

The SWIFI technique is used to corrupt the kernel calls that are applied to the OS by the considered workload. Indeed, bit-flip corruption of the parameters of the kernel calls is prone to lead to both invalid (parameter is out of range of the domain of validity) and erroneous (values are different from nominal values) parameters.

The considered faults are transient in nature (the corruption are applied only once when the targeted call is executed). Indeed, such faults tend to provoke more subtle erroneous behaviors. We plan also to consider random selection among the kernel calls, the parameters and the bits for applying bit-flips. Both single and multiple bit-flips have been included in the faultload.

During these experiments, we intend to carefully monitor the proportion of “no signaling” outcomes, as this might be considered as some form of “non significant test” and thus impair the efficiency of the benchmark.

The simulation of hardware faults is being carried out using SWIFI both on the code and data segments. This makes it possible to consistently study the impact of hardware faults both on the BT (OS kernel) as well as on other parts of the SUB as well. Another form of faultload will consist in provoking ill-timed hardware exceptions.

For **OLTP**, we give special attention to operator and software faults. Hardware faults are also considered, as they are still reported as a significant cause of failure in transactional systems, especially for faults affecting devices such as disks, network interface devices, power sources, etc.

Operator faults are unanimously considered as one of the major causes of failures in transactional systems. More precisely, in database systems most operator faults are database administrator mistakes. Operator faults in a DBMS can be easily injected by simply reproducing common database administrator mistakes. That is, we are using exactly the same means used in the field by the real database administrator (i.e., we really reproduce operator faults). In order to make the procedure fully automatic, faults are injected by a set of scripts that perform the wrong operation at a given moment (the fault trigger). As usually happens in traditional fault injection, the fault trigger can be defined in such a way that operator faults can be uniformly distributed over time or can be synchronized with a specific event or command of the workload. In a distributed environment where the SUB is composed by several machines the faults should be distributed uniformly through all the machines. An example of a preliminary dependability benchmark using operator faults developed within DBench can be found in [32].

For the injection of *software faults* we use the Generic Software Fault Injection Technique (G-SWFIT) [29] that consists of modifying the ready-to-run binary code of software modules by introducing specific changes that correspond to the code that would have been generated by the compiler if the software fault were in the high-level source code. A library of mutations previously defined guides the injection of code changes: the target application code is scanned for specific low-level instruction patterns and selective mutations are performed on those patterns to emulate related high-level faults defined (notion of "educated" mutations). A crucial aspect of the proposed technique (concerning its use for benchmark faultloads) is in the fact that it works at the machine-code level and it does not required the availability of the source code of the target program. This feature makes it possible to apply the proposed technique to virtually any program, which is particularly relevant for dependability benchmarking of COTS components or COTS based systems.

Experiments concerning *hardware faults* are carried out using UMLinux, which provides an environment to set up the SUB on virtual machines. Emphasis is put on vulnerable items, such as storage and interconnection components. We are thus investigating the impact of various kinds of hard-disk faults (single hard-disk completely inaccessible or only certain blocks on the hard-disk faulty). Depending on when and how the data is accessed, the injected fault affects the server either at once, at some later point or even not at all. We are also injecting faults into the machine network interfaces as well as into the interconnection network itself. Faults injected into a machine network interface include inability to send or receive network packets or sending/receiving corrupted network packets. Faults injected into the interconnection network include broken cables and unavailable routes.

For **DECU**, the faultload consists in transient hardware faults, especially in the memory components of the SoC. The NEXUS embedded processor debug interface is used to support a low-intrusive SWIFI technique. It builds upon the "on-the-fly" memory access function that was primarily meant for debugging hard real-time systems (without stopping nor affecting the system). This feature allows reading and writing the memory while the processor is running, without any significant overhead.

4.4 Measurements and Measures

Table 4 summarizes the main dependability measures considered for the benchmark prototypes.

Table 4 – Dependability measures for the benchmark prototypes

GPOS	OLTP	DECU
• Classical OS-level reported and non-reported failure modes • Restoration time (after failure) • OS-specific timing measurements • Error propagation channels within the OS	• Transaction throughput • Availability • Integrity • DBMS-specific dependability measures • Network-level measures (UMLinux)	• Processing time for the workload tasks • Reaction time • Failure modes and criticity analysis with respect to process variables

For **GPOS**, the dependability measures of interest encompass:

- Classical OS-level reported and non-reported failure modes. Of course both types of failure modes are characterized by the observation of events/values and related timing data,
- Error propagation channels between application processes via the OS.

The first category encompasses outcomes that have been classically used to characterize OS robustness such as, i) reported failures: returned error code, exception raised, ii) non (explicitly) reported failures: OS hang

or OS crash. Although they characterize the behavior of the OS in presence of faults (independently of the service being delivered at the application level), these outcomes are usually diagnosed within the SUB (i.e., the application layer here) by means of the observation capabilities of the BMS (see Table 1).

Also we are considering higher-level failure modes by explicitly monitoring some application-level services and comparing the behaviors observed with and without faultload (*oracle*). It is worth pointing out that the ultimate form of such an application-level assessment is experienced in the case of the benchmarks targeting specific application domains (as is the case for the DECU benchmark prototype).

Timing measurements (duration of execution of workload) are supported by (OS-related) performance benchmarks. On the contrary, discrepancies of the results and/or integrity checks with respect to a reference run (value and/or integrity failures) are far more difficult to diagnose in the case when performance benchmarks are only available as executable code packages (source code is not always made available). Accordingly, we plan to use specific reference features (e.g., synthetic/generic type of service) in association with or in addition to the background traffic applied to activate the OS kernel being benchmarked.

One special class of timing measurements is very specific of a dependability benchmark; it corresponds to time taken by the OS to be able to provide its service after a failure has occurred (the so-called *reboot* and *restart* times). Indeed, this time might be significantly impacted by: i) the type of failure experienced (two specific cases are OS crash and OS hang), and ii) the extent of the corruption of the internal state of the OS.

OS specific timing features that govern the transitions between its internal states could also be of interest for a system integrator. Examples are the “context switch time” and the “system call overhead”. It is worth noting that such measurements require a very detailed analysis and knowledge of the architecture of the OS kernel. Nevertheless, as is considered by performance benchmarks, “empty” kernel calls can be used to allow for the overall time spent for a kernel call to be readily measured.

Another interesting insight that can be gained from the experiments is to assess the risk of propagation (via the OS) of errors between application processes that have a priori no functional and explicit communications, but that share the resources provided by the OS. Such an analysis is especially interesting when application processes with different levels of integrity share common resources (in particular, the data structures of the OS). What is feared is that the effects of errors in a low-integrity process impact a higher integrity process through the OS. It is important to note that such impacts encompass not only the propagation of erroneous data, but also denial of service that may result for example in the high-integrity process to hang.

For **OLTP**, the basic measure is the transaction throughput in the presence of the faultload. It is simply adapted from the TPC-C benchmark, by measuring the number of transactions executed *in the presence of the faults specified in the faultload*. This measures the impact of faults in the performance and favors systems with higher capability of tolerating faults, fast recovery time, etc

The availability measure is estimated both from the point of view of the SUB and of an end-user:

- SUB availability: The system is available when it is able to respond to at least one end-user within the minimum response time set for each type of transaction by the TPC-C benchmark.
- End-user availability: The system is available for one end-user if it responds to a submitted transaction within the minimum response time set for that type of transaction by TPC-C.

An estimate of the impact of the faultload on data integrity is obtained by measuring the number of data errors detected by means of consistency checks using both business rules (defined in TPC-C workload specification) and the database metadata, so as to achieve a comprehensive test.

Another interesting measure concerns the number of transactions aborted when the database engine is subjected to the faultload.

In addition to these global measures, more specific measures can be obtained, provided the independent strategy (see Section 3.2) is being used to conduct the experiments. Two examples, reported in [35], are: the study of the variation of the measures with respect to each fault categories or according to each injection run. Other specific measures include coverage measures such as error detection and recovery efficiencies.

For the **DECU** benchmarks, the first set of measures considered are based on timing measurements made with and without faultload: execution time for each task of the workload, reaction time (delay between any input variable change to the corresponding change of an output variable).

Because in this case a specific application program is available, a detailed analysis of the application can lead to a refined characterization of failure modes from either the application program or the controlled process, or both. As an example, we show hereafter the type of detailed classification that can be carried out. This illustrates the issue

From the application viewpoint, the anticipated failure modes have been decomposed into the following categories: i) no new data is being sent or the command line remains stuck at the last valid data (application program hang and communication link is OK) or, ii) wrong data is being sent (in which case we distinguish between two cases: close to or very apart from nominal data), iii) the sending of the (correct) data is delayed. These application-level failures can affect the engine in several ways according to the output variables to the actuators that are affected. Three criticality levels have been considered for this analysis: non-optimal operation, Noise and/or vibration, unpredictable (with risk of permanent damage). For sake of conciseness, we denote this last level as "catastrophic". Table 5 shows a classification that relates the application-level failure modes, the output variables and the criticality levels.

Table 5 – Relations between application failure modes, output variables and criticality

Output variable → ↓ Failure mode	Rail regulator valve	Swirl valve	Waste gate valve	Fuel injection (timing)	Fuel injection (duration)
No (new) data	catastrophic	non optimal op.	non optimal op.	catastrophic	catastrophic
Close to nominal	noise / vibration	noise / vibration	noise / vibration	noise / vibration	non optimal op.
Distinct from nominal	catastrophic	non optimal op.	non optimal op.	catastrophic	catastrophic
Delayed	noise / vibration	noise / vibration	noise / vibration	noise / vibration	non optimal op.

5 Conclusion

The work reported in the paper presented a comprehensive framework aimed at supporting the development of dependability benchmarks for computer systems, with particular emphasis on COTS components and COTS-based systems. Our goal is to determine generic ways for defining dependability benchmarks. Given the multi-dimensional nature of the problem, we started by the identification of all the dimensions of the problem. Three groups of dimensions have been identified: categorization, measure and experimentation dimensions.

In this paper we have focused on the experimentation dimension by identifying the three main experimentation sub-dimensions: workload, faultload and measurements, as well as the set of procedures, rules, and resources needed to conduct the experiments carried out in a dependability benchmark

To illustrate how this framework can be used in real situations, we have proposed three examples of dependability benchmarks concerning operating systems, transactional and embedded systems. As it is explicitly shown in the paper, the fact that the proposed framework could be readily used to specify prototype dependability benchmarks for targets covering such a wide spectrum of distinct categories of computer systems and components systems can be viewed as already a significant step forward.

These prototype benchmarks are still under investigation as part of the DBench project. More important than the results that will be obtained for the various systems and components being considered, is the actual experience that we will gain from the implementation and application of these prototype benchmarks. We are confident that this necessary step will allow us to refine the proposed conceptual framework for dependability benchmarking.

Preliminary results are progressively obtained (some are already becoming available, e.g., see [35]). We expect to obtain quite a significant amount of data from these experiments. Our intent is to carry out an extensive cross-exploitation of the various results obtained. To support this effort, we will take advantage of the services and facilities offered by data warehousing and web-enabled On-Line Analytical Processing (OLAP) technologies to i) conveniently store and share the results obtained and ii) efficiently perform multidimensional analyses (e.g., see [36]).

References

[1] M. Malhotra and K. S. Trivedi, "Dependability Modeling Using Petri Nets," *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 428-440, September 1995.

[2] 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.), pp. 267-287, Los Vaqueros, CA, USA: IEEE CS Press, 1998.

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

[4] 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.), pp. 249-265, San Jose, CA, USA: IEEE CS Press, 1999.

[5] M. Dal Cin, G. Huszerl and K. Kosmidis, "Quantitative Evaluation of Dependability-Critical Systems Based on Guarded Statechart Models," in *Proc. 4th Int. High-Assurance Systems Engineering Symp. (HASE-99),* Washington, DC, USA, 1999, pp. 37-45, (IEEE CS Press).

[6] J. Arlat, J. Boué and Y. Crouzet, "Validation-based Development of Dependable Systems," *IEEE Micro*, vol. 19, no. 4, pp. 66-79, July-August 1999.

[7] 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, 2001, pp. 161-170, (IEEE CS Press).

[8] A. Mukherjee and D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking," *IEEE Transactions of Software Engineering*, vol. 23, no. 6, pp., 1997.

[9] T. K. Tsai, R. K. Iyer and D. Jewitt, “An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems,” in *Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26),* Sendai, Japan, 1996, pp. 314-323, (IEEE CS Press).

[10] A. Brown and D. A. Patterson, “Towards Availability Benchmarks: A Cases Study of Software RAID Systems,” in *Proc. 2000 USENIX Annual Technical Conference,* San Diego, CA, USA, 2000, (USENIX Association).

[11] P. Koopman and J. DeVale, “Comparing the Robustness of POSIX Operating Systems,” in *Proc. 29th Int. Symp. on Fault-Tolerant Comp. (FTCS-29),* Madison, WI, USA, 1999, pp. 30-37, (IEEE CS Press).

[12] J. E. Forrester and B. P. Miller, “An Empirical Study of the Robustness of Windows NT Applications Using Random Testing,” in *Proc. 4th USENIX Windows System Symposium,* Seattle, WA, USA, 2000.

[13] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, “Dependability of COTS Microkernel-Based Systems,” *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 138-163, February 2002.

[14] J. Gray (Ed.) *The Benchmark Handbook for Database and Transaction Processing Systems,* San Francisco, CA, USA: Morgan Kaufmann Publishers, 1993.

[15] D. Brock, *A Recommendation for High-Availability Options in TPC Benchmarks,* http://www.tpc.org/information/other/articles/ha.asp, 1999.

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

[17] M. Kalyanakrishnam, Z. Kalbarczyk and R. K. Iyer, “Failure Data Analysis of LAN of Windows NT Based Computers,” in *Proc. 18th Int. Symposium on Reliable Distributed Systems (SRDS'99),* Lausanne, Switzerland, 1999, pp. 178-187, (IEEE CS Press).

[18] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems,” *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913-923, August 1993.

[19] D. Costa, T. Rilho and H. Madeira, “Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection,” in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* New York, NY, USA, 2000, pp. 251-260, (IEEE CS Press).

[20] M. Dal Cin, K. Buchacker, L. Lemus Zuniga, R. Lindström, A. Johansson, H. Madeira, V. Sieh and N. Suri, “Workload and Faultload Selection,” DBench Project IST 2000-25425 Deliverable no. ETEI3, Available at http://www.laas.fr/dbench/deliverables.html, 2002.

[21] H. Madeira *et al.*, “Dependability Benchmark Definition: DBench Prototypes,” DBench Project IST 2000-25425 Deliverable no. BDEV1, Available at http://www.laas.fr/dbench/deliverables.html, 2002.

[22] K. Kanoun, H. Madeira and J. Arlat, “A Framework for Dependability Benchmarking,” in *Supplemental Volume of the 2002 Int. Conf. on Dependable Systems and Networks (DSN-2002) - Workshop on Dependability Benchmarking,* Washington, DC, USA, 2002, pp. F.7-F.8.

[23] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, “Fault Injection for Dependability Validation — A Methodology and Some Applications,” *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166-182, February 1990.

[24] K. Kanoun, M. Kaâniche and J.-C. Laprie, “Qualitative and Quantitative Reliability Assessment,” *IEEE Software*, vol. 14, no. 2, pp. 77-86, mars 1997.

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

[26] 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, 1998, pp. 284-293, (IEEE CS Press).

[27] 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, 2000, pp. 417-426, (IEEE CS Press).

[28] J. Arlat and Y. Crouzet, “Faultload Representativeness for Dependability Benchmarking,” in *Supplement Int. Conference on Dependable Systems and Networks (DSN-2002),* Washington, DC, USA, 2002, pp. F.29-F.30.

[29] J. Durães and H. Madeira, “Emulation of Software Faults by Selective Mutations at Machine-code Level,” in *Proc. 13th Int. Symp. on Software Reliability Engineering (ISSRE-2002),* Annapolis, MD, USA, 2002, pp. 329-340, (IEEE CS Press).

[30] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun and T. Marteau, "Analysis of the Effects of Real and Injected Software Faults — Linux as a Case Study," in *Proc. 2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC-2002),* Tsukuba, Japan, 2002, pp. 51-58, (IEEE CS Press).
[31] J. Durães and H. Madeira, "Characterization of Operating Systems Behavior in the Presence of Faulty Drivers Through Software Fault Emulation," in *Proc. 2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC-2002),* Tsukuba, Japan, 2002, pp. 201-209, (IEEE CS Press).
[32] M. Vieira and H. Madeira, "Definition of Faultloads Based on Operator Faults for DMBS Recovery Benchmarking," in *Proc. 2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC-2002),* Tsukuba, Japan, 2002, pp. 265-272, (IEEE CS Press).
[33] 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, 1997, pp. 32-36, (IEEE CS Press).
[34] K. Buchacker and V. Sieh, "Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects," in *Proc. High-Assurance System Engineering Symp. (HASE-2001),* Boca Raton, FL, USA, 2001, pp. 95-105.
[35] M. Vieira and H. Madeira, "Benchmarking the Dependability of Different OLTP Systems", Technical Report DEI-006-2002, ISSN 0873-9293, Departamento de Engenharia Informática – Faculdade de Ciências e Tecnologia da Universidade de Coimbra, Portugal, 2002 (Submitted for publication).
[36] H. Madeira, J. Costa and M. Vieira, "The OLAP and Data Warehousing Approaches for Analysis and Sharing of Results from Dependability Evaluation Experiments," Technical Report, DEI-007-2002, ISSN 0873-9293, Departamento de Engenharia Informática – Faculdade de Ciências e Tecnologia da Universidade de Coimbra, Portugal, 2002 (Submitted for publication).