Chapter 1

Dependability Benchmarking Concepts

Abstract

The goal of dependability benchmarking is to provide generic ways for characterizing the behaviour 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 chapter presents the DBench framework for defining dependability benchmarks for computer systems, with particular emphasis on off-the-shelf (OTS), commercial (COTS) or not, and also OTS-based systems. The concepts and the multiple dimensions of the problem are discussed and the key aspects of the specification of dependability benchmarks are presented. A benchmark validation approach is proposed, consisting mainly on the verification of a set of well-defined properties that key benchmark components should meet in order to be meaningful under economically acceptable conditions. A tool for On-Line Analytical Processing (OLAP) of experiment results allowing analysing, sharing, and cross-exploiting results from dependability benchmarking experiments is then presented. The chapter ends with a short overview of the dependability benchmarks specified and implemented in DBench.

1.1 Introduction

The goal of benchmarking the dependability of computer systems is to provide generic and reproducible ways for characterizing their behaviour 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/or 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 their domain of validity. The objective of such 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, such as to:

- *Characterize* the dependability of a component or a system.
- *Compare* alternative or competitive solutions according to one or several dependability attributes.

The dependability benchmark results could be used to 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.

DBench has developed a framework for defining dependability benchmarks for computer systems, with emphasis on OTS, commercial or not, and also OTS-based systems, via experimentation and modelling. The ultimate objective of our work is to provide a framework and guidelines for defining dependability benchmarks for computer systems, and provide means for implementing them.

From a practical point of view, a dependability benchmark is a specification of a procedure to assess measures related to the behaviour of a computer system or computer component in the presence of faults. Obviously, the benchmark specification may include source code samples or even tools to facilitate the benchmark implementation. However, what is relevant is that one must be able to implement the dependability benchmark from that specification (i.e., perform all the steps required to obtain the measures for a given system or component under benchmarking). In other words, the specification should be unambiguous and clear enough to allow:

- Implementation of the specification in order to benchmark the dependability of a given target system or component.
- Full understanding and interpretation of the benchmark results.

In DBench we have identified the main dimensions that are decisive for defining dependability benchmarks and the way experimentation can be conducted in practice. These dimensions describe: i) the target system and the benchmarking context, ii) the measures to be evaluated, as well as iii) the experimental conditions.

The *DBench framework* defines not only the above dimensions but also the different guidelines that should be followed in order to develop useful benchmarks. These guidelines should include procedures and rules for i) implementing the specifications, ii) performing the experiments to ensure uniform conditions for measurement and iii) exploiting results.

In addition, to be *meaningful* under *economically acceptable conditions*, a dependability benchmark should satisfy a set of properties. For example, a benchmark must be repeatable (in statistical terms), representative, portable, cost effective, etc. These properties represent goals that must be achieved when defining dependability benchmarks. The relevance of the benchmark properties is quite clear, as they take into account all the relevant problems that must be solved to define and validate actual dependability benchmarks. These properties should be taken into consideration from the earliest phases of the benchmark definition as they have a deep impact on the experimental dimensions and, consequently, on the benchmark specification. Also, these properties should be checked after specifying a benchmark, which can be accomplish (in our proposal) through the implementation of that benchmark in a set of representative systems for a given application domain.

To exemplify how the above issues can actually be handled in different application domains, five examples of benchmarks and their associated prototypes (i.e., actual implementations of the benchmarks) have been developed in DBench. They concern general-purpose operating systems, embedded and transactional systems. It is expected that these benchmarks and the results obtained will help understanding the various concepts presented in this chapter, at least for the considered classes of systems.

In order to make the analysis and sharing of dependability benchmark results possible, we propose an approach, based on multidimensional analysis and data warehousing, and On-Line Analytical Processing (OLAP) technology, to solve the problem of analysing, sharing, and cross-exploiting results from dependability benchmarking experiments. OLAP allows the analysis of raw data of single experiments, analysis and comparison of benchmark results obtained in different systems, and sharing of results among project partners.

This chapter is composed of five sections. Section 1.2 presents some related work. Section 1.3 presents the dimensions and the experimental benchmarking rules that should be clearly defined in the specifications of a dependability benchmark. Section 1.4 describes the most important properties expected from such type of benchmarks and provides some practical considerations concerning their validation. Section 1.5 presents the OLAP tool. Finally, Section 1.6 briefly presents the five benchmarks developed within DBench. These benchmarks are detailed in Chapters 2 to 6.

1.2 Related Work

Although a variety of dependability evaluation methods and techniques aimed at different system levels and domains, ranging from analytical modelling to simulation and experimental approaches, including those based on fault injection have been reported (e.g., see [Malhotra and Trivedi 1995; Karlsson et al. 1998; Arlat et al. 1999; Carreira et al. 1999; Dal Cin et al. 1999; Martínez et al. 1999; Hiller et al. 2001]), a generalized approach that can be used to evaluate and compare different systems and components still does not exist. It is also worth mentioning the pioneering work that addressed the benchmarking of software robustness [Mukherjee and Siewiorek 1997] and investigated the development of some forms of benchmarks [Tsai et al. 1996; Brown and Patterson 2000]. Among these efforts, operating systems received much attention, e.g., see [Koopman and DeVale 1999; Forrester and Miller 2000; Arlat et al. 2002], 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 [Gray 1993]. Clearly, the preliminary efforts at developing dependability benchmarks, including focused robustness testing techniques, do not benefit yet from such a level of recognition. 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 [Brock 1999], field measurement and fault injection [Iyer and Tang 1996], field measurement and modelling [Coccoli et al. 2002], [Kalyanakrishnam et al. 1999], fault injection and modelling [Arlat et al. 1993], and use of standard performance benchmarks as a workload for dependability evaluation [Costa et al. 2000].

To the best of our knowledge, little has been reported on dependability benchmark, as it has been addressed in DBench (i.e., as a standardized approach to evaluate and compare different computer systems and components). However, the work carried out in the DBench project in the last three years (and published in many papers in major international conferences and journals) has contributed to spark the research on dependability benchmarking in other teams, both at universities and computer industry sites.

The work carried out in the context of the Special Interest Group on Dependability Benchmarking (SIGDeB), created by the IFIP WG 10.4, has been particularly relevant, as it merges contributions from industry and academia. A concrete proposal issued by the SIGDeB was in the form of a set of standardized availability classes to benchmark database and transactional servers [Wilson et al. 2002]. This work is quite complementary with the work carried out in DBench, as the SIGDeB has addressed mainly the system provider point of view, while DBench has focused mainly the system integrator and end-users points of view.

Recent work at Sun Microsystems defined a high-level framework [Zhu et al. 2003a] dedicated specifically to availability benchmarking. Within this framework, two specific benchmarks have been developed. One of them [Zhu et al. 2003] addresses specific aspects of a system's robustness on handling maintenance events such as the replacement of a failed hardware component or the installation of a software patch. The other benchmark is related to system recovery [Mauro et al. 2004].

At IBM, the Autonomic Computing initiative (see http://www.ibm.com/autonomic) is also developing benchmarks to quantify a system's level of autonomic capability, addressing four main spaces of IBM's self-management: self-configuration, self-healing, self-optimisation, and self-protection [Lightstone et al. 2003].

In the academia side, the work developed at Berkeley University has lead to the recent proposal of a dependability benchmark to assess human-assisted recovery processes [Brown et al. 2004].

1.3 Benchmark Specifications

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 system(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 signalling, failure modes, etc.) make 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 specify and understand all relevant aspects of dependability benchmarks.

We will first present the main dimensions and then address the benchmark conduct concerns.

1.3.1 Dependability Benchmarking Dimensions

The various dimensions are grouped into three classes referred to respectively as categorization, measure and experimentation dimensions. These classes of dimensions are introduced hereafter and further detailed in the next section.

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. Measures could be either qualitative or quantitative, related to dependability or performance and dependability, comprehensive or specific and assessed from experimentation only or modelling and experimentation.

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 class dimensions 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 1.1 outlines our classification of dimensions.

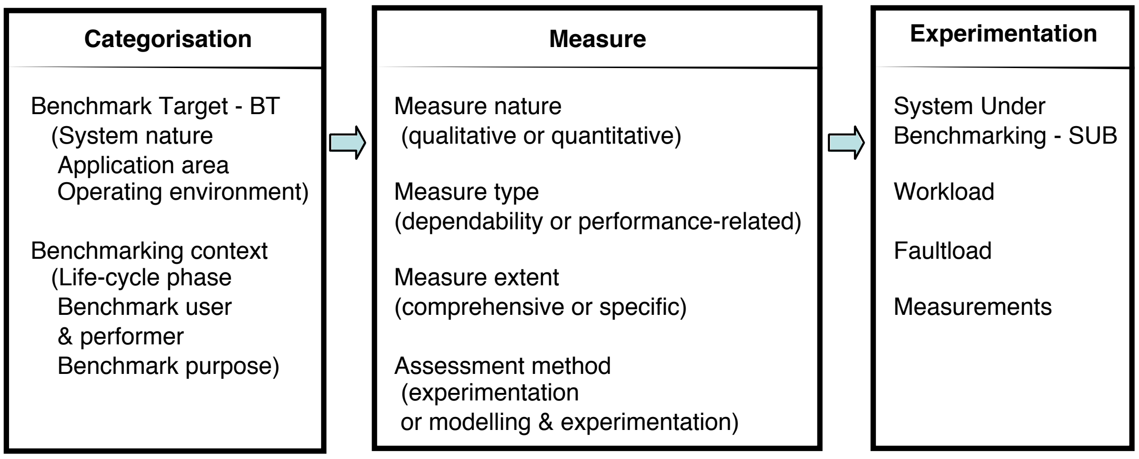

Figure 1.1:Dependability benchmarking dimensions

1.3.1.1 Categorization Dimensions

The categorization dimensions are intended to describe clearly the benchmarked system and its benchmarking context.

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 *operating environment* characterizes the environment in which the system usually works. It encompasses many important aspects ranging from the functional activity of the benchmark target to the faults affecting its behaviour. 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.

Benchmarking Context

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

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

Benchmark user and *benchmark performer* - The benchmark user is the person or entity actually using the benchmark results. The benchmark performer is the person or entity actually carrying out the benchmark experiments. The user and the performer may be the same person or entity but they also may be different entities. A benchmark user or performer can be a system manufacturer, a system integrator or an end-user. In addition, the benchmark performer could be a third party. These entities usually have i) different visions of the target system, ii) distinct controllability and observability levels for experimentation and iii) different expectations from the measures.

Benchmark purpose - The main purpose of a dependability benchmark is to compare systems and/or components according to a given set of dependability attributes. However, a benchmark can also be used for other purposes. For example, during the early design phases, a dependability benchmark could be useful to reveal weak points in a prototype and thus could help support the decision whether to purchase a particular computer platform or to use a given software COTS component. 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, and help the system manager to tune the system for the best compromise between performance and dependability (e.g., in a large database, this can be used to tune the recovery mechanisms).

However, as mentioned before, the results of a 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.

1.3.1.2 Measure Dimensions

Dependability benchmarks encompass measures assessed under particular conditions. Therefore an important task in the benchmark specification concerns the definition 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* (e.g., system availability, safety or system response time in the presence of faults).

As the occurrence or activation of faults may lead to performance degradation without leading necessarily 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 behaviour 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 behaviour. Evaluation of comprehensive measures may require the use of both experimentation and modelling. The system restart time in the presence of faults or the number of transactions per minutes in the presence of faults are examples of comprehensive measures (respectively dependability and performance–related measures) that can be obtained directly from experimentation on the benchmarked system. Availability and safety are examples of comprehensive dependability measures than can be obtained from modelling, and modelling may be based on information obtained from benchmark experimentation. It is worth mentioning that modelling 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 behaviour. Specific measures characterize for example error detection and recovery or fault diagnosis capabilities with respect to a given class of faults. 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.

The actual set of benchmark measures is very much dependent upon the categorisation dimensions. In particular, the failure modes are directly deduced from the service that is fulfilled by the system in the 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). 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.

Benchmarking measures should be specified in a non-ambiguous way. They should also be representative of the application domain and take into account the benchmarking context.

1.3.1.3 Experimentation Dimensions

These dimensions include all aspects related to experimentation on the benchmark 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". The explicit identification of the type of system under benchmarking is essential as the three other dimensions (workload, faultload and measurements) should be described with respect of the system under benchmarking.

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[1].

As an example, Figure 1.2 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 application programming interface (API). 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.

[1] 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.

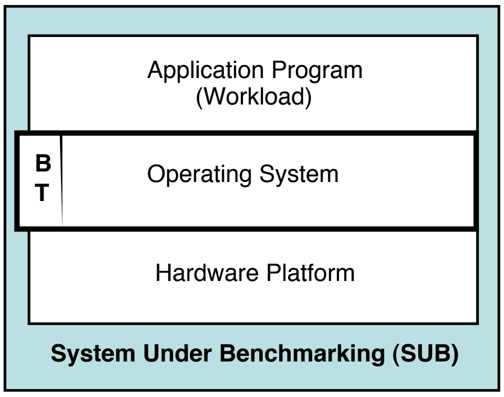

Figure 1.2 Example of System Under Benchmarking (SUB) for an OS Benchmark Target (BT)

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.

From a practical point of view, we recommend to take advantage of already developed performance benchmark workloads. Usually, such workloads constitute representative workloads for the application domain they are concerned with. In addition, generally, they are defined by working groups and consortiums. As a consequence, in general, they result from an agreement between these group and consortium members, which is a guarantee of representativeness.

However, specific synthetic workloads can be developed for specific application areas. Such workloads should be as representative of the application domain as possible in order for the benchmark to be useful.

As is often the practice in performance benchmarks, and as was already identified in previous experimental dependability studies (e.g., see [Arlat et al. 1990]), 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 analyse 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 [Arlat et al. 2002]), the workload can take the form of a modular workload, made up of a set of workload processes, each tailored to sensitise a specific entity of the BT.

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. External faults clearly depend on the operating environment.

In the DBench framework, only faults that are *external* to the BT could be included in the faultload, in order not to modify the BT. It is worth to mention, that faults affecting the parts of the SUB that are outside the BT are considered external faults with respect to the BT. Such faults can be part of the applied faultload.

A particular attention has been devoted to fault representativeness in DBench. A full deliverable [Gil et al. 2002] was entirely dedicated to this very important topic. The results obtained have guided the faultloads we used in the benchmarks developed in the project.

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 [Kanoun et al. 1997] and [Durães and Madeira 2003b] 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. 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. To be pragmatic, we use dependability benchmarks corresponding to a specific 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 [Daran and Thévenod-Fosse 1996], [Karlsson et al. 1998], [Folkesson et al. 1998; Madeira et al. 2000]), 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 [Arlat and Crouzet 2002].

For what concerns **hardware faults**, it was shown that i) multiple-bit bit-flips are highly likely in processor registers, ii) flipping bits in memory using a 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 [Durães and Madeira 2002] and more specifically for what concerns OSs, that it was seldom possible to simulate the erroneous behaviours induced by faulty drivers when injecting at the level of the API [Jarboui et al. 2002], and thus more focused approaches were needed [Durães and Madeira 2002c].

For **operator faults**, which are particularly relevant for complex systems that require intensive management such as big servers, it has been possible to devise a faultload featuring virtually a zero injection distance [Vieira and Madeira 2002b]. In fact, operator faults are emulated using exactly the same interface used by operators to perform normal administrative tasks, and operator faults consist just in sequences of commands that perform erroneous administrative operations.

Measurements

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

Basic measurements include the identification of SUB outcomes as resulting from the workload and faultload application. In practice, for such outcomes to be diagnosed it might even be necessary that the behaviour of the benchmark target (results computed and/or control flow) in presence of faults be compared with its nominal behaviour (notion of *oracle*).

Another important dimension is related to the assessment of the behaviour 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. 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.

From a practical point of view, as far as COTS are concerned, our recommendation is to make exclusively use of information that is available from outside the BT.

1.3.2 Benchmark Conduct

Benchmarking a real life system may require several steps forming a *benchmarking scenario*. Experimentation requires a benchmark management system in addition to procedures and rules that are needed to control the way a dependability benchmark is applied and used. These three complementary aspects are addressed hereafter.

1.3.2.1 Benchmarking Scenarios

The analysis of a real-life system allows identification of the dependability measures that are interesting to be evaluated for this specific system. Some of them may be directly evaluated through an experimental benchmarking approach, while other comprehensive measures may require in addition a modelling approach.

Figure 1.3 shows the three benchmarking steps: analysis, modelling and experimentation. The benchmarking scenarios differ by the kind of links between these steps. Various scenarios have been described in detail in [Madeira et al. 2001]. These scenarios range from the most simple one in which only experimentation is involved to the most complex one in which modelling and experimentations are tightly linked.

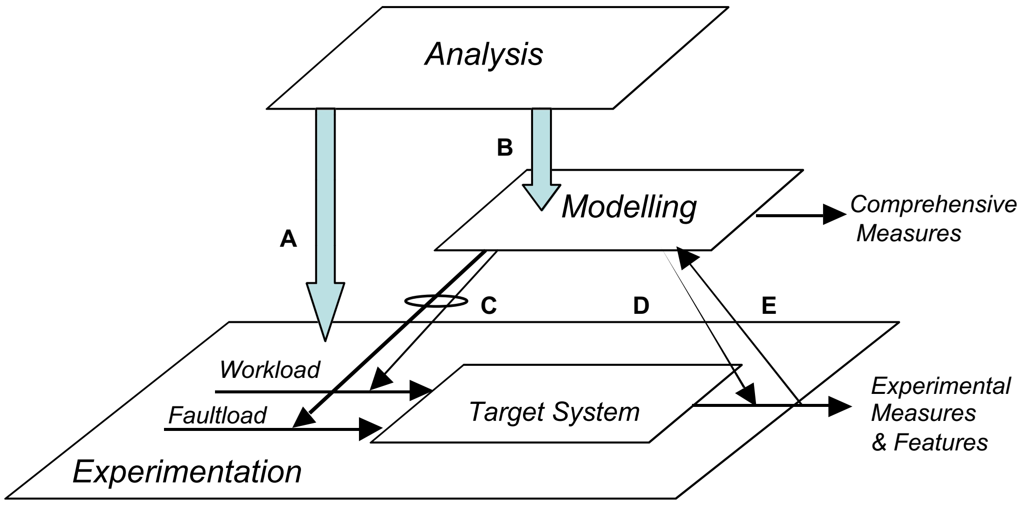

Figure 1.3:Dependability benchmarking steps

Modelling is very specific to the system and application considered and usually models cannot be defined in a general and standard approach (except for very specific systems). On the other hand, even though the experimental step is also dependent on the system to be benchmarked, by selecting small generic systems (such as operating systems or database management systems), it is possible to define standard procedures for benchmarking such systems.

In order to illustrate how modelling can be used to complement experimental benchmarking, an example of a very simple model is given in Chapter 6, for On Line Transaction Processing systems. The model evaluates steady availability and the total cost of failures.

1.3.2.2 Benchmark Management System

In addition to the SUB, some supplementary resources and instrumentation are usually 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 its 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).

1.3.2.3 Procedures and Rules

The procedures and rules needed to control the way a dependability benchmark is applied and used 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 are applied via the SUB.
- Scaling rules to adapt the same benchmark to systems of varying sizes. These scaling rules 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. However, it may also affect, to a less degree, the associated faultload size. One has to investigate the need of scaling up or down the faultload 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, signalling 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.
- Rules to avoid "gaming" to produce optimistic or biased results.

1.4 Benchmark Validation

As stated in the introduction, to be meaningful under economically acceptable conditions, a dependability benchmark should satisfy a set of properties. These properties should be taken into account from the benchmark specification phase, mainly when defining the measure and experimental dimensions. Also, during the benchmark implementation in a form of a prototype, one should address explicitly these properties. Finally, one has to verify experimentally that the key properties are fulfilled. Verification of these properties constitutes a large part of the benchmark validation.

Given the agreement nature of all benchmarks, it is obvious that gaining confidence on a real dependability benchmark is a fundamental step towards the acceptance by the computer industry or by the user community. The fundamental question is whether one can rely on the results provided by a given dependability benchmark or not. This question is particularly acute, as the ultimate goal of a dependability benchmark is to allow the comparison of different systems/components, which means that without proper validation of the benchmark one may reach the wrong conclusions.

A useful benchmark must be representative, portable, repeatable, scalable, non-intrusive, and simple to use. This way, the validation of a dependability benchmark consists on the verification that the main benchmark components (workload, faultload, and measures) fulfil the set of key properties, such as repeatability, portability, representativeness, scalability, non-intrusiveness and simplicity of use mentioned before. The following sub-sections present some guidelines on how to validate each property.

1.4.1 Representativeness

A benchmark is defined for an abstraction of a given class of systems and of a given application domain. Representativeness concerns all benchmarking dimensions. In particular, the measures, the workload and the faultload should be as representative as possible.

Measures are strongly related to the benchmark context and should be meaningful for this context.

The workload representativeness reflects how well the workload of the benchmark corresponds to the actual workload of the real system. The workload representativeness must be considered in the context of the benchmark application area, and it is essential to assure that workload profile includes a realistic set the activities found in real systems for the benchmark application area. Different areas have different workloads and may therefore need different benchmarks.

The experience from performance benchmarking has been quite helpful in achieving workload representativeness, and workloads available from performance benchmarks have been used in DBench whenever possible. Nevertheless, some issues still need to be researched concerning the definition of representative workloads for dependability benchmarks. Indeed, the erroneous behaviour of the system results from the combined effects of the workload and the faultload, and this interaction should be addressed carefully.

Fault representativeness is a measure of how well injected faults correspond to real faults, i.e., faults affecting systems in real use. It is important to know how representative the faults are for the target system being benchmarked, so that it is possible to estimate how relevant the result is. This property must, of course, be considered in the context of the application area and the operating environment. Field data is the best way to validate whether the set of faults considered in the faultload represent real faults from the field or not. The problem is that field data on the underlying causes of computer failures is not generally available. However, depending on the class of faults, previous works available in the dependability literature have been used to achieve confidence on the representativeness of a given faultload.

Faultload representativeness has largely been addressed in DBench and an entire deliverable [Gil et al. 2002] was devoted to this research problem.

1.4.2 Repeatability and Reproducibility

The benchmarking of a given system can be performed either based on an existing benchmark implementation (an existing prototype) or only based on existing specifications. Repeatability concerns the first situation (i.e., the benchmark prototype) while reproducibility is more related to the second situation (i.e., the benchmark specifications).

Repeatability is the property, which guarantees *statistically equivalent results* when the benchmark is run more than once in the *same environment* (i.e., using the same SUB, the same execution profile — workload and faultload — and the same prototype. This property is central to benchmarking and is one of the foundations upon which the entire concept of benchmarking relies. Without repeatability no one would be able to trust the results obtained from benchmarking experiments. This would mandate that everybody used their own evaluation method, contradictory to the underlying philosophy and aim of benchmarking. All credible benchmarks must therefore be repeatable.

Reproducibility is the property, which guarantees that *another party* obtains statistically equivalent results when the benchmark is implemented from the *same specifications* and is used to benchmark the same SUB. Reproducibility is strongly related to the amount of details given in the specifications. Indeed, at the specification level, a trade-off is usually required: the specifications should be at the same time i) *general enough* to be applied to the class of systems addressed by the benchmark and ii) *specific enough* to be directly applicable without distorting the original specifications during the benchmark implementation.

Generally, the details for the benchmark experimental dimensions are only settled at the prototype level. The benchmark repeatability is in fact quite easy to verify by running the benchmark several times in the same system and measuring the statistical variation in the results. The reproducibility is more difficult to verify, as it requires the benchmark implementation (from the specification) and running by different teams.

1.4.3 Portability

Portability refers to the applicability of a benchmark specification to various target systems within a particular application area. A portable benchmark can be implemented and run on various target systems within the application area. Portability enables the use of benchmarks for comparing computer systems or components.

Portability is very dependent on the benchmark specification, particularly the way some key benchmark components (such as the faultload and workload) are specified. For example, if the faultload is defined in a general way the benchmark will be easily portable. However, portability and reproducibility have opposite impact on the specification level of details. This reinforces the fact that the properties should be taken into account from the specification phase.

The portability property can be verified by implementing a given benchmark in a representative number of systems for a given application area, which has been partially done in the DBench prototypes.

1.4.4 Non-intrusiveness

If the implementation of the benchmark (particularly to what concerns the workload and faultload) introduces changes on the system under benchmarking (either at the structure level or at the behaviour level) it means that the benchmark is intrusive. The benchmark must require minimum changes in the system under benchmarking.

In order to satisfy non- intrusiveness at least at the benchmark target, a very important rule has been established in the DBench framework: this rule avoids fault injection in the target system it self. However, faults can be injected in the parts of SUB outside the target system. In this way, the non-intrusiveness property with respect to the BT is normally easy to verify through the benchmark implementation.

1.4.5 Scalability

A benchmark must be able to evaluate systems of different sizes. Usually, benchmark scaling is achieved by defining a set of scaling rules in the benchmark specification. Scaling rules mostly affect the workload, but other components such as the faultload may also have to be scaled. For example, in a large transactional system the number of disks could be very high, and this will affect the faultload based on hardware faults, as more disks may fail. Note that, the size of the faultload may also affect the time needed to conduct the benchmarking process, which means that there are always some limitations to the scale up of a given benchmark for extremely large systems.

The obvious way to show that a given benchmark verifies the scalability property is to implement that benchmark for systems of quite different sizes. Another important aspect is to assess if the benchmark allows the comparison among systems of different sizes. In fact, the benchmark may be able to evaluate systems of different sizes, but that does not mean that the comparison of results from systems of different sizes is always meaningful.

1.4.6 Benchmarking Time and Cost

The benchmarking time is the time required to obtain the result from the benchmark. It consists of three parts: i) set-up and preparations, ii) running the actual benchmark program (i.e., execution time) and iii) data analysis. Of course it is desirable to have as short a benchmarking time as possible.

Ideally one would construct benchmarks that both evaluate the system thoroughly and accurately, while having short benchmarking times. Normally, these properties are contradictory which calls for a trade-off.

A key goal of dependability benchmarking is to provide an efficient and cost effective approach to characterise dependability of computer systems and components. Of course, dependability benchmarking is only attractive as long as its perceived value is higher than the associated costs. Thus the level of automation in all the phases of benchmarking is very important in order to reduce the benchmarking time as much as possible.

The time needed to implement and run a benchmark and the degree of automation can be experimentally assessed very easily. Ideally, the benchmark execution should take only a few hours per system. However, due to the complexity of some systems (e.g., transactional systems), a few days are acceptable. As a guideline in DBench, we have stipulated that the maximum time acceptable to benchmark a given system is one week (beyond this limit, we assume this property is not valid anymore).

1.5 On-Line Analytical Processing (OLAP) of Experiment Results

As the key goal of dependability benchmarking is the comparative assessment of dependability features across different systems or components, the proposal of a general approach (and tools) to analyse and cross-exploit results obtained from different dependability evaluation experiments is a decisive issue. This way, two important questions may be raised concerning the analysis and cross-exploitation of results of dependability benchmarking:

- How to analyse the usually large amount of raw data produced in dependability benchmarking experiments, especially when the analysis is complex and have to take into account many aspects of the experimental set-up (e.g., SUB, configurations, workload, faultload, measures, etc)?
- How to compare results from different experiments or results of similar experiments across different systems if the tools, data formats, and the set-up details are different and, often, incompatible?

We propose an approach, based on multidimensional analysis and data warehousing, and On-Line Analytical Processing (OLAP) technology, to solve the problem of analysing, sharing, and cross-exploiting results from dependability benchmarking experiments [Madeira et al. 2003]. The central idea is to collect the raw data produced in the experiments and store it in a multidimensional data structure (data warehouse). The data analysis is done through the use of commercially available OLAP tools such as the ones traditionally used in business decision support analysis [Kimball 98] (e.g., Discoverer® from Oracle). That is, instead of following the usual trend of adding data analysis features to experimental set-up tools, we propose a clear separation between the experimental set-up (target specific) and the result analysis set-up (general in our approach). Existing tools and experimental set-ups are used as they are and just export the data obtained in the experiments to a data warehouse, where all the analysis and cross-exploitation of results can be done in an efficient and general way. The key advantages of the proposed approach are the following:

- It is a **general** approach:
  - As the experimental evaluation of dependability is a multidimensional problem, our proposal of using data warehousing and OLAP technologies for result analysis is applicable to all the experimental dependability evaluation scenarios.

  - Any existing tools and experimental set-up can be used; the only thing we need to know is the data format of the raw results produced during experiments, in order to read them into the data warehouse.
  - The results analysis set-up is based on standard and well-proved OLAP technologies, which are the general-purpose tools for the analysis of multidimensional datasets.
- It is easy to **compare** and cross-exploit raw results from different experiments, as all the raw data is stored in a common data warehouse (the multidimensional structure needed to store the data from each different experiment forces that only meaningful comparisons can be done).
- It is easy to **share** raw results world wide as the data stored in a data warehouse can be explored by web-enabled versions of OLAP tools. This way, it is possible to make available to the entire dependability community the raw data of dependability evaluation experiments (or field data on impact of faults collected from system logs) together with an OLAP tool to explore that data.

OLAP was developed for the analysis and sharing of dependability benchmark results in several scenarios: analysis of raw data of single experiments, analysis and comparison of benchmark results obtained in different systems, and sharing of results among project partners[2]. We see this possibility of making the raw results available to the dependability community, together with a tool to analyse them, as an important step to increase the exchange of results among researchers and practitioners. This way, people not only can read the traditional paper presenting the final results, but also will have access to the raw data and can use that data for other purposes or to compare with their own results.

1.6 Benchmarks developed within DBench

Chapters 2 to 6 illustrate how the general concepts presented in the previous sections can be applied to the definition of dependability 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) *embedded systems* and iii) *OLTP systems* (On-Line Transaction Processing systems).

Concerning embedded systems, two classes of application domains are being considered (space and automotive). Also, two complementary benchmarks are defined for OLTP systems..

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

2 Some examples and the corresponding raw results that can be analysed through a web based OLAP tool can be found at http://gbd.dei.uc.pt/?view_highlight=12.

- Benchmark Target: In all cases the benchmark target is always either an *Off-the-Shelf component*, either commercial (COTS) or Open Software System or a *system including at least one such component*.
- Life cycle phase: It is assumed that the 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 reveals 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) compare alternative systems.
- 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.

For each benchmark prototype, we briefly describe the measures and the main experimentation dimensions that will be detailed in Chapters 2 to 6.

1.6.1 General purpose Operating Systems

The benchmark developed in Chapter 2 for general purpose OSs addresses mainly the robustness of the OS (and more precisely its kernel) with respect to faulty applications. The measures evaluated are: i) the distribution of the OS outcomes following activation of faulty system calls, ii) reaction time of the OS for faulty system calls and iii) system restart time after activation of faulty system calls. These basic measures are complemented by additional measures intended to refine them.

The workload we have considered is a realistic workload, TPC-C client. The workload and the faultload are implemented separately. A subset of system calls used by the workload is selected a priori (according to OS function criticality). When a system call belonging to this subset is invoked it is intercepted and substituted by the same system call with a corrupted parameter value. Three different parameter corruption techniques are used and their effects are compared.

The prototype developed is used to compare the dependability of three operating systems: Windows NT4, Windows 2000 and Windows XP.

1.6.2 Real Time Kernels in Onboard Space Systems

In space real-time systems, correctness of operation depends not only on the right results being generated but also on the results being produced within certain time constraints. With the increase use of COTS Real-Time Kernels (RTK) in embedded systems the need for assuring a dependability level of in such kernels also arose. Among several dependability attributes, the determinism of the response time of RTK services, even in presence of faults, is of paramount importance for hard real-time systems. This is particularly true for onboard space systems that are more exposed to external disturbances such as radiation.

DBench-RTK, presented in Chapter 3, is a benchmark for assessing the predictability of response time of a Real-Time Kernel (RTK) service calls. This benchmark aims to allow integrators/developers to compare different RTKs with respect to their ability to provide their services within the expected time frame. The benchmark provides metrics for characterizing this capability. The main measure provided is the *Predictability* that scores a RTK vis-à-vis its system calls response time fitting within its specification.

1.6.3 Engine Control Applications in Automotive Systems

The core of modern vehicle engines is managed by the control algorithms running inside Electronic Control Units (ECUs). Due to the high scales of integration used in these electronic components, engine control systems are subject to a number of transient faults that may impact their hardware and lead their software to the production of unsafe outputs for the vehicle engine. The dependability benchmark developed in Chapter 4 addresses the robustness of the control applications running inside the ECUs with respect to transient hardware faults.

The proposed workload is inspired by the standards currently used in Europe for the emission certification of light duty vehicles. On the other hand, the faultload is defined in terms of hardware faults that affect the cells of the memory allocating the engine software control. The high scale of integration used in most modern engine ECUs induces many controllability and observability problems that have a deep impact over the definition of a suitable benchmark procedure. In order to overcome these problems, the benchmark exploits the tracing and on-the-fly memory access features existing in the debugging interfaces of current automotive embedded microprocessors.

The benchmark prototype has been specialized to the case of diesel engine control units. This prototype shows the feasibility of the approach and the various steps in which the benchmark procedure can be divided in practice. The prototype is also used as a support for the experiments conducted for validation purposes.

1.6.4 On Line Transaction Processing Systems

Large transactional systems are usually at the very centre of the IT infrastructure of companies. Even short downtimes of such systems are very expensive. To be able to evaluate the dependability of transactional systems is, therefore, of great importance. We have developed two complementary benchmarks for OLTP systems, respectively DBench-OLTP and TPC-C-Depend. Both benchmarks are extensions of TPC-C and closely follow the form and structure of the latter and use TPC-C workload. Both benchmarks are used to characterize the Data Base Management System (DBMS) and to compare DBMSs.

The measures of **DBench-OLTP,** presented in Chapter 5, include the TPC-C measures in the presence of faults (i.e., the number of transactions executed per minute and the price per transactions in the presence of faults), system availability during the benchmarking (for both the server and the clients point of view), and the number of data integrity errors detected during the benchmark runs. The measures are derived directly from experimentation.

The faultload includes the three fault classes considered in DBench: hardware faults, operator faults and software faults.

The DBench-OLTP prototypes implemented along the DBench project have been used to benchmark many OLTP systems and configurations, including large database management systems (DBMS) such as Oracle 9i, small DBMS as PostgreSQL, running on top of Windows (several versions) and Linux operating systems, and including several database/server configurations.

The two final measures provided by **TPC-C-Depend**, presented in Chapter 6, are the stationary system availability and the total cost of failures. The measures area evaluated by combining measures obtained from experimentation on the target system (e.g., the percentages of the various *failure modes*) and information from outside the benchmark experimentation (e.g., the failure rate, the repair rate and the cost of each failure mode).

The faultload used in Chapter 6 includes exclusively hardware faults, but operator faults have also been considered for validation purpose.

The prototype developed has been used to illustrate the benchmark on Oracle and PostgreSQL.

Main difference between the two OLTP benchmarks

Having two different benchmarks for OLTP systems may raise one important question from the potential benchmark users: which benchmark to use under given circumstances?

The answer to that question lies in the difference between the two benchmarks developed. In addition to the difference in the measures and the way to obtain them presented in the previous paragraph, the main difference between the two benchmarks is mostly related to the approach followed in i) the faultload definition and, as a consequence, to ii) the benchmark potential users. The following paragraphs detail the differences between the two benchmarks and can be used as a guideline for the benchmark user on how to select the appropriate benchmark for a given OLTP system.

Faultload Definition

In DBench-OLTP, the faultload specification is part of the benchmark specification. The faultload specification results from research concerning which faults are possible and representative in different system under benchmarks (SUBs). This includes the database management system (DBMS), the OS and hardware. When applying DBench-OLTP to a new SUB, the faultload has to be ported, i.e., the injection tools and emulation techniques may have to be adapted to the new DBMS/OS/Hardware. However, from a conceptual viewpoint, the faultload is the same. This is especially useful to compare the results across different SUBs.

Users that choose the DBench-OLTP benchmark accept that the faultload defined in the benchmark specification is representative of real scenarios. In this case, as the faultload used is conceptually the same, the results obtained by different users are directly comparable.

In TPC-C-Depend benchmark the faultload is considered dependent of each particular configuration used in the SUB, in particular hardware faults. Fault rates and costs associated to each fault are taken into account. These figures are not part of the benchmark specification and must be provided by the end-user of the benchmark. A formal language to describe the hardware is specifically recommended to emulate the hardware and its faults. Because the hardware can be emulated, several configurations may be evaluated without actually incurring in the costs of buying each one. This is especially useful for developers or IT administrators that are tied to a particular DBMS and wish to select the “best” hardware that should be used with the DBMS.

Users that chose the TPC-C-Depend benchmark must be able to provide fault rates for the systems being considered. In this case, results obtained by different users are not directly comparable because different users may consider different fault rates for the same system.

Potential Benchmark Users

From what precedes, it could be seen that:

- DBench-OLTP benchmark potential users are:

- End-users (i.e., system/IT administrators) when choosing among different (similar) DBMS.
- End-users (i.e., system/IT administrators) with a well defined (already decided) SUB, experimenting different optimisation settings to decide the best trade-off between performance and stability.
- DBMS manufacturers when assessing the quality of the DBMS before releasing it to the market. This is especially useful during development phases where compromises must be made when deciding to improve stability or performance or time-to-market, etc.

- TPC-C-Depend benchmark potential users are:

- DMBS developers intending to recommend a specific hardware for their its DBMS.
- Vendors of Information Systems Package (a complete solution including software and hardware).
- Legacy DBMS that must be used in new hardware.