DBench

Dependability Benchmarking

IST-2000-25425

Workload and Faultload Selection

Report Version: Deliverable ETIE3

Report Preparation Date: June 2002

Classification: Public Circulation

Contract Start Date: 1 January 2001

Duration: 36m

Project Co-ordinator: LAAS-CNRS (France)

Partners: Chalmers University of Technology (Sweden),
Critical Software (Portugal),
Universidade de Coimbra (Portugal),
Universität, Erlangen-Nürnberg (Germany),
LAAS-CNRS (France),
Universidad Politechnica de Valencia (Spain).

Sponsor: Microsoft (UK)

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

Workload and Faultload Selection*

V 4.0

Contents

*Contact : M. Dal Cin, Friedrich Alexander Universität Erlangen- Nürnberg
dalcin@informatik.uni-erlangen.de

Authored by: Mario Dal Cin[+], Karama Kanoun*, Kerstin Buchacker[+], Lenin Lemus Zuinga[++], Robert Lindstrom**, Andreas Johanson**, Henrique Madeira***, Volkmar Sieh[+], and Neeraj Suri**

*LAAS **Chalmers ***FTCTUC [+]FAU [++]UPVLC

Abstract

The selection of appropriate work- and faultloads is an important step when conducting dependability benchmarking experiments. In this deliverable we first present basic definitions pertinent to the selection of work- and faultloads. Then we address important challenges encountered when selecting appropriate work- and faultloads, e.g., representativeness, portability or implementability of the selected work- and faultload, etc..

In the second part of this deliverable examples of workloads and faultloads for the focussed areas are presented, viz. for transactional and embedded systems, in order to show the types of work- and faultloads that can be used in different benchmarking experiments. We also include short descriptions of the considered applications and of the experiment platforms employed to make the examples more comprehensible. Our intention is to show that the process of selecting a work- and faultload is very sensitive to the experimental set-up and the goals of benchmarking.

1 Introduction

The selection of a ‘good’ work- and faultload is an important step (phase) when conducting a dependability benchmarking experiment. Here we will consider this load selection step only. Task T3.1 (deliverable BDEV1) deals with the issues of conducting dependability benchmarking experiments.

A workload represents a typical execution profile for the considered application area, whereas a faultload represents a typical set of faults affecting the operation of the system. We consider work- and faultload as part of the stressload, which comprises, roughly speaking, all stressful conditions applied to the system under benchmarking. Of course, the selected stressload should reflect the actual conditions encountered in the considered application area.

On the one hand, widely accepted performance benchmarks can provide the basis for selecting application specific workloads. However, for dependability benchmarking this workload must be modified and extended. For example, it could take into account the impact of preventative and corrective maintenance actions evoked by the considered faultload. On the other hand, the selection of a faultload is a non-standard, rather pragmatic and less well understood task. As we will see, this gives rise to some serious challenges. In any case, it must be possible to tune both the workload and the faultload to closely match the “real” world situation.

For selecting the work- and faultload it is important that the scope of a benchmark is well defined. Certain benchmarks may focus on the services provided by the target system. Then, for example, the services provided by the OS for the application have to be identified as well as the service provided by the application to the end-user. Or, the services delivered by a Data Base Management System (DBMS) are to be defined considering the transactional paradigm. These services provide the basis for defining appropriate work- and faultloads (and appropriate measures). Different benchmark scopes require different approaches. There are several ways of accomplishing this including, for example, having a set of smaller benchmarks targeting different application areas, conducting several benchmarking experiments targeting different system components or addressing different measures. For each approach the appropriate work- and faultload has to be specified.

The Deliverable is structured as follows. In Section 2 we present basic definitions pertinent to the selection of work- and faultloads. Section 3 addresses some important challenges encountered when selecting appropriate work- and faultloads, and Section 4 addresses the topic of work- and faultload synchronisation. In Section 5 some examples of workloads and faultloads for the focussed areas are presented, i.e., transactional and embedded systems. We present these examples in order to ilustrate the types of work- and faultloads that can be used in different benchmarking experiments. We also include short descriptions of the considered applications and of the experiment platforms in order to make the examples more comprehensible.

2 Definitions

First we present several definitions pertinent to the selection of work- and fault loads for dependability benchmarking experiments (DBE). In general, a dependability benchmark specifies several DBE's, and, hence, several work- and faultloads.

2.1 Dependability Benchmarks

A dependability benchmark is the specification of a procedure to assess measures related to the behaviour of a computer system or computer component in the presence of faults. The main elements of a dependability benchmark are:
- Measures
- Workloads
- Faultloads
- Procedures and rules

In this way, a dependability benchmark consists of the specifications of the benchmark elements. This specification could be just a document. What is relevant is that one must be able to implement the benchmark (i.e., perform all the steps required to obtain the measures for a given component under benchmarking) from that specification. Obviously, the benchmark specification can include source code samples. In addition to the specifications, one can include into the set of benchmark elements also certain tools as something required to implement the benchmark.

Dependability benchmarks compare the dependability of alternative or competitive systems according to one or several dependability attributes. To this end, the target system is described in a generic way as a set of components (including key components for dependability) that perform specific functions in the presence of a set of faults. Accordingly, the work- and faultload are also described on a quite abstract level. To compare alternative solutions, the benchmark results can be used either to characterize system dependability capabilities in a qualitative manner or to assess these capabilities quantitatively. Of course, DBE's use mainly fault injection to asses these capabilities.

The scope of so called internal fault injection experiments is to characterize the dependability of a system or a system component in order to identify weak parts. In this case it is necessary to know the system (and its faults) in more detail to make possible the evaluation of specific measures. Accordingly, internal fault injection experiments (FIE's) require the selection of very specific faultloads which, for example, stress specific fault tolerance capabilities of certain system components.

DBE's aim at finding the best existing solution and the 'best practice' by comparison; internal FIE's aim at improving an existing solution. Thus, DBE's produce results obtained in a standard manner for public distribution that comply with the benchmark specification, while internal FIE's produce results for internal use and are used mainly for system validation and tuning.

A drawback of benchmarks is that they may not yield enough information for improvements; a drawback of internal experiments is that certain weak points may not get detected when looking at only one possible solution. Therefore, internal experiments and benchmarks can complement each other. Comparing systems pinpoints to differences in their dependability and these differences may pinpoint to weak points in the systems architectures and organisations. (Note, that internal fault injection experiments my be used for comparison across systems as well, but in a more focused way than external benchmarks.)

A benchmark must be as representative as possible for a given domain but, as an abstraction of that domain, it will always be an imperfect representation of reality. However, the aim is to find a useful representation that captures the essential elements of the given domain and provides practical ways to characterise the computer features that help the vendors/integrators to improve their products and help the users in their purchase decisions. An important step towards this goal is, of course, the selection of appropriate work- and faultloads.

2.2 Benchmarking Phases

A dependability benchmark and the conduction of DBE's can be divided into several steps. In [CF2] and deliverable ETI1 we have identified three basic steps and defined scenarios for conducting benchmarks. These basic steps can be divided into phases. Any meaningful, documented sequence of phases defines a benchmarking process. Benchmarking processes can differ in the number and type of the employed phases. There are fundamental phases which can provide the basis for extensions and refinements, for example, selection of the benchmarking object and determination of the benchmarking objectives, analysis of required and available resources and costs analysis, determination of the measures, preparation of the benchmarking experiments, experimentation, and presentation of the results. The selection of work- and faultloads is a sub-phase of the preparation phase.

2.3 System Under Benchmarking and Dependability Benchmark Target

We distinguish between the system under benchmarking and the dependability benchmark target.

System under benchmarking is the system which is benchmarked. That is, on the system under benchmarking the measures of a benchmarking experiment are applied.

Dependability benchmark target is the benchmarking object. It is the system or system component which is intended to be characterized by the benchmark.

In the following we will use the abbreviations SUB and DBT for "System under Benchmarking" and "Dependability Benchmark Target", respectively.

The distinction between SUB and DBT is important since the DBT must not be the target for fault injections in a DBE, whereas it may very well be the target of fault injections for internal FIE's.

Thus, the DBT may be the SUB as such or a single component of the SUB. If the SUB is the DBT , only external faults (e.g. input or operator faults) can be considered. Moreover, a dependability benchmark may target several components of the SUB or the same component focusing on different dependability aspects. Hence, a benchmark may define several benchmarking experiments with differing stressloads.

Finally, the *dependability benchmarking configuration* (DBC) is the complete set of system components and tools required to perform the dependability bechmarking experiments.

2.4 Stressload

As we see it, stressload is the set of "stresses" that can be introduced into the system. It is the entire load explicitly or implicitly applied to the SUB in a DBE. It includes the workload, the background load and the faultload. Their definitions are given below.

Thus, a stressload comprises several components. Here, we provide generic definitions of these components. These definitions have to be refined in the context of the application areas. We see three different origins of stresses for a SUB: stresses of certain selected capabilities of the SUB, like stressing an scheduler or other "high level" capabilities of the OS. The second category are application-induced stresses that arise when a certain application uses the SUB capabilities to a high degree or induces errors in the behaviour of the SUB. Which capabilities are stressed is then of course highly application dependent. Hence, in this case it is important to define stressload with respect to the application itself. The third category is end-user interactions, this could be the requests to an application given by an end-user, like users making database requests. For an embedded system the "user", such as a human, might be replaced by the surrounding environment that might affect the sensors in the system giving rise to a high frequency of interrupts or faulty behavior.

As mentioned, the three components making up a stressload are: workload, background load and faultload.

2.4.1 Workload

The workload is the computational load for the SUB. It can comprise, for example, a real application, a client workload produced by external clients who communicate with the SUB, or corrective and preventive maintenance actions. Thus, the definition of a workload depends on which system component is benchmarked, that is, on the DBT. In the case of a transactional server as DBT, the workload is the "request stream" arriving on network connection. For benchmarking operating system kernels the workload is the composition of the libraries and the application. On the other hand, benchmarking an entire operating system the library may be part of the DBT. (If the hardware is the DBT, also the operating system may be considered part of the workload. Benchmarking hardware, however, is not our goal.)

Depending on the application and the benchmark scope, the workload can stress single system components as memory management or input/output devices, or it can stress the target system as a whole. Thus, it is highly application dependent which components and capabilities of the system are stressed in a DBE. For example for a switching system allowing processing of say up to 4000 subscribers, a realistic workload could be to simulate an increasing number of subscribers.

We distinguish between synthetic, realistic and real workloads.

Synthetic workloads can be random queries in a database table or a random sequence of system calls. A synthetic workload may be needed in case the application is known but not yet coded. Synthetic client workloads certainly make sense since they are easy to generate and to modify. Moreover, they can be adapted to a statistical mix of applications.

Realistic workloads are artificial but still reflect real situations. For example, a realistic workload may be a (benchmark) kernel, i.e. an extracted, computationally intensive part of a real application, or a deterministic or stochastic workload model. For example, for benchmarking a Transactional System, one may stochastically simulate the behaviour of clients accessing a model database.

Real workloads are applications or suits of applications that the end-user is running, for example, a suit of numerical applications. Results of dependability benchmarking using real workloads are, in general, more dependable and accurate.

2.4.2 Background Load

The background load is used to restrict the system resources available to the workload. For example, the background load could be a concurrent process running on the same system. The background load may be considered as part of the workload. We prefer, however, to distinguish the workload from the background load since the workload is, in general, an application determined by the user, whereas the background load (e.g., network traffic) may not always be under his control and its presence may not directly depend on the DBT.

2.4.3 Faultload

.

Faultload is the set of faults, their intended location, insertion time and distribution in time and space to be inserted into the SUB. For benchmarking a system component, faults of the faultload are injected into system parts external to this component, the DBT, or induced by inputs.

- Some examples are: a) injection of faults in a device driver to evaluate the way the operating system (OS) behaves in the presence of a “mad” driver; b) injection of faults in the OS to evaluate a fault-tolerance layer at the application level; c) injection of faults in an application process (emulating a buggy application) to evaluate the probability of error propagation to other processes. [ETI3].

This is not to say that the DBT must be faultfree, it may contain, for example, design faults a priori or may get corrupted by injected faults.

Also the faultload can be split into different parts: software faults (e.g., design faults in programs), operator faults, and hardware faults, which may be external to the SUB (external perturbations), or internal. Network faults are examples of external faults; incorrect functions of internal system components are internal faults.

Figures 2.4.1 – 2.4.5 show some examples for dependability benchmark targets, stressloads and the rest of the SUB. Notice that the faultload is divided into several parts, e.g., operator faults, and faults injected into the operating system and into the hardware. For certain DBE’s not all parts of the faultload may be relevant.

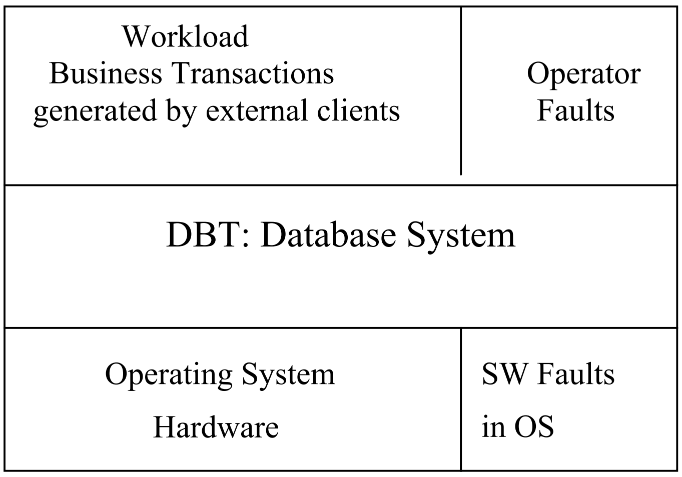

Fig.2.4.1 SUB: Transactional System

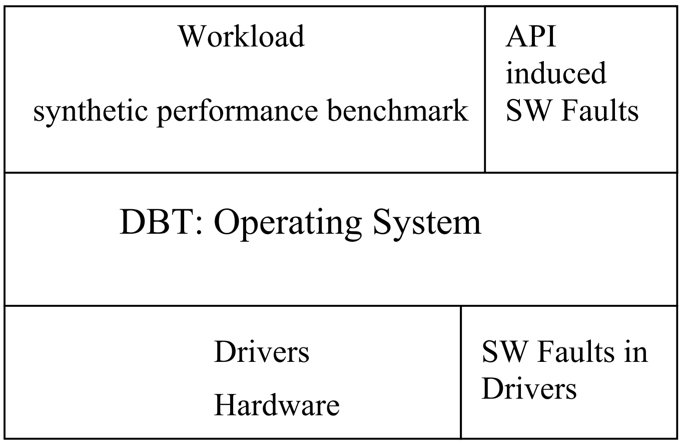

Fig.2.4.2 SUB: Operating System

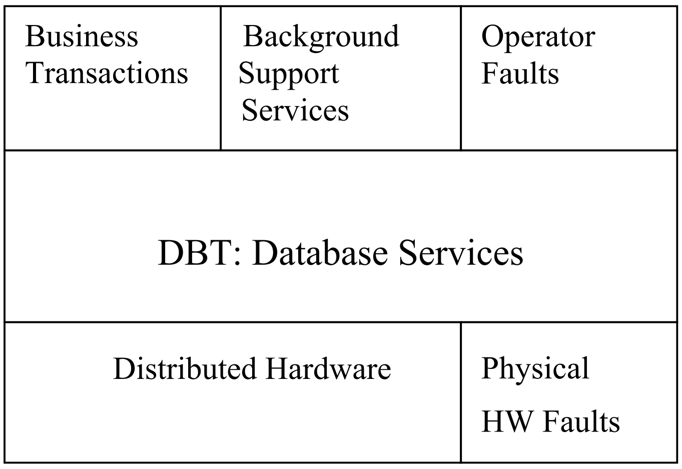

Figure 2.4.3 *SUB: Distributed Transactional System*

In case of embedded systems, the DBT can be the application or the application and the Real Time Operating System (RTOS). These options are shown in Figures 2.4.4 and 2.4.5 .

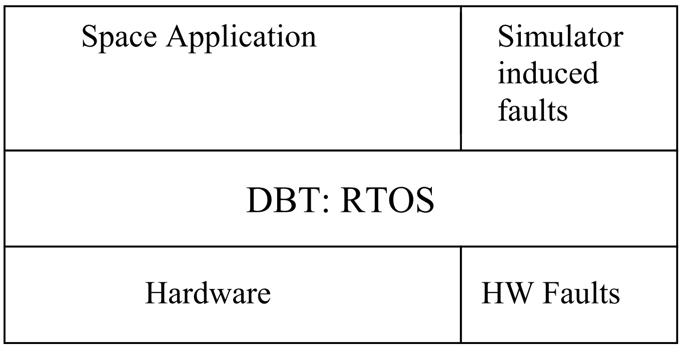

Figure 2.4.4 Embedded System .

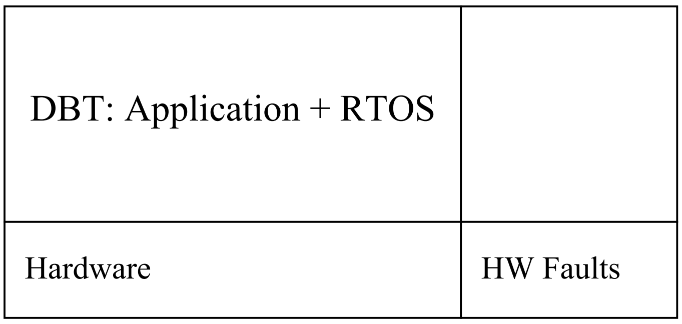

Fig. 2.4.5 Embedded System.
In this case the Application and the RTOS (or the runtime library) are the DBT

3 Challenges in Selecting Workloads and Faultloads

The basic questions of work- and faultload selection are: "What is considered an appropriate workload and what is a suitable faultload; what should be part of a good work- and faultload; how accurately can we represent the real world?" To answer these questions one is faced with some serious challenges.

Generally speaking, the approaches to select appropriate workloads can be either object-oriented, computation-oriented, or interaction-oriented.; the approaches to select suitable faultloads can be either function-oriented, structure-oriented, or statistics-oriented.

3.1 Workload

Widely accepted performance benchmarks can provide the basis for selecting application specific workloads also for dependability benchmarking.

However, the uncritical use of existing unchanged performance benchmarks as workloads for dependability benchmarking is not always meaningful, because we do not only want to assess the performance but we also want to know whether the operations were performed correctly – even in the presence of faults, if the system is to be fault-tolerant. Existing performance benchmarks do not usually save or output results of operations in any way. The results only exist in volatile memory or registers until overwritten with the results of the next operation. However, for dependability benchmarking, it is essential to be able to compare the results of the benchmark to a set of results that are known to be correct. Otherwise the effects of faults on the system cannot be judged. Moreover, the checks must be performed outside of the SUB – or at least outside the fault injection target, e.g., by an external client - such that they are not invalidated by the faultload. Fortunately, some of the existing widely used performance benchmarks can be extended and modified in an appropriate way and preventive and corrective actions can be taken into account, whenever required**.**

Although usage of real-world workloads is preferred, it can be useful to design realistic or pure synthetic ones as well. However, if one intends to benchmark COTS (software and hardware) as SUB's, one only has access to their public interface when defining a realistic or synthetic workload. These interfaces are, for example, the system call interface for Operating Systems, input values from the environment for Embedded Systems, and SQL commands for Transactional Systems.

3.2 Faultload

The selected faultload for a specific dependability benchmarking experiment should be widely acceptable by the computing community. Also, it must be possible to implement it by appropriate fault injection techniques. Last but not least, it must be portable.

3.2.1 Faultload based on operator faults

Concerning operator faults, although some faults are highly system dependent, the analysis made in [ETI3] shows, for example, that most of the operator faults can be found in several DBMS. Therefore, we consider here DBMS only.

Operator faults in database systems are database administrator mistakes. End-user errors are not relevant, as the end-user actions do not affect directly the dependability of the system. From the functional point-of-view the DBMS administration can be defined as a set of core functionalities, namely: memory and processes, security, storage, database objects, and recovery mechanisms. A possible solution to define a portable faultload is to focus on the high abstraction level that corresponds to the core functionalities of DBMS administration. This functional abstraction level corresponds to the set of administration functionalities common to most of the DBMS.

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

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

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

It is important to note that some types of faults do not affect the system in a visible way concerning recovery. An example of these faults is the security class faults. When a database administrator introduces inadvertently a security fault the system continues to work normally until another person maliciously takes advantage from that to break into the system (this is a second event).

3.2.2 Faultloads based on software faults

As observed in [ETI3], Orthogonal Defect Classification (ODC) is based on software faults found in real programs and classifies software defects in a set of non-overlapping classes (the classification is based on the way faults have been corrected) [Chillarege 95]. ODC classes of faults can be used as staring point for selecting a faultload, but as each class includes a very large number of possible faults a knowledgeable selection of the faults is necessary.

Certain software faults can be simulated by corrupting system call parameters. That is to say, the fault is injected in the system call and then the system call is executed with this corrupted data. The parameter values can be corrupted by i) issuing exhaustive bit-flips (e.g., 32 per parameter) or, ii) replacing them with invalid values.

For benchmarking, software faults can also be generated, for instance by educated mutations [ETI3]. The main problem is to find accurate emulations of specific high-level languages programming errors that are usually responsible for common software faults. For example, the key aspects of the G-SWIFT technique, introduced in [ETI3], are a library of low-level instructions patterns and mutations that relate to specific high-level faults in specific language constructs and structures, and a pre-processing step of the target application to generate a (large) number of mutants. The execution of each mutant represents the injection of a fault.

Inserting software faults rises the problem of intrusiveness as they may change the behavior (timing) of the software. Hence, care has to be taken when interpreting the results of a DBE based on software faults.

For internal FIE's, inserting software faults (design faults) into existing software could be interesting for developers, who might want to test parts of the software in the presence of other faulty parts. Then, faults are injected in one software module to evaluate the behavior of other modules.

3.2.3 Faultload based on hardware faults

In task T2.2, deliverable [ETI2], we have identified a large number of physical faults and their representations at higher, logical levels. These representations allow us to inject 'faults' into the SUB by avoiding the necessity to generate the actual, physical faults. The basic idea is to find a common layer for injecting representations of hardware faults. The RT layer (register transfer level) is an appropriate one because it is possible to represent most types of fault at lower levels (HW level) on the RT level. So, taking into account that injecting faults into RTL emulates a high percentage of hardware faults, this technique is a good fault injection method. However, as has been seen in task T2.2, there are some hardware faults that can not be emulated using this technique.

Hardware fault representations (in short, hardware faults) are specified by the following characteristics: *type, location*, and *temporal characteristics*. For example, when considering hardware faults, several injection locations can be chosen by a fault injection tool. (Note that these locations should be external to the DBT in order to benchmark the original, unmodified component.) Locations for faults that can be injected are, for example, CPU registers, code memory, data memory, or input signals. When injecting faults into the code memory, the fault can corrupt either the application code or the OS code.

With regard to temporal characteristics of faults, hardware faults may be permanent or transient faults and can have a certain duration or appear intermittently. Permanent faults, such as a stuck memory bit or a bad block on a hard disk, remain in the system from their activation time until the end of the measurement run. Transient faults are like permanent faults, except for the fact that they disappear after a given duration. Transient faults, such as bit flips, are generated once and will disappear as soon as the bit is set to a new value. Intermittent faults are transient faults which are generated again and again at a given interval.

Last but not least, hardware faults are also characterised by their rates of occurrence. It is a well known fact, that occurrence rates are very hard to come by.

When benchmarking computing systems such as networks or clusters or networks of clusters also wrong hardware configurations (in short, configuration faults) could be of interest. These configuration faults could be ‘injected’ as actual faulty configurations. However, we prefer to propose a dependability benchmark configuration that allows the emulation/simulation of the systems, and, hence, allows to simulate the system configuration faults.

3.2.4 Considered versus injected faults

Ideally, when benchmarking a system all existing faults should be considered in the first place. However, what faults to incorporate in the faultload depends, for example, on whether an benchmarking experiment or an internal fault injection experiment is to be performed. For DBE’s the faultload should be related to overall dependability features and to the services of the SUB. For internal FIE’s one may inject only faults which are said to be tolerated to test the system for its fault-tolerance capabilities and for the performance loss caused by faults. Contrary, in some internal experiments no faults which are said to be tolerated are injected to improve the speed of the experiment campaign. Internal faults are mainly determined by the implementation of the target system.

Thus, we have to distinguish between the set of recommended faults that should be considered in a benchmarking experiment and the actual faultload for the experiment. Among the recommended faults only a part may be “implementable”; other will not be injected because of time constraints, limited number of experiments, dangerousness (risk of data loss or hardware damage) etc. However, it is important that the benchmark also reveals whether or not the DBT behaves as expected also in the presence of certain faults that cannot be injected directly. In this case, it may be necessary that a simulation platform for the SUB is available.

3.3 Representativeness

The selected work- and faultload should be representative in order to closely match the “real” world situation. Also concentrating on representative fault classes only can help to reduce the complexity of the benchmarking process.

Representative workloads for DBE’s, for example representative client workloads for Operating or Transactional Systems, already exist. One can choose among virtually hundreds of performance related workloads from the literature. For Embedded Control Systems general representativeness is more difficult to achieve. On the other hand, to achieve workload representativeness for special embedded areas may be straightforward.

More important, the selection of a representative faultload is decisive for the success of a dependability benchmark. Hence, knowledge of the representativeness of faults is a pre-requisite for selecting a suitable faultload.

We admit that the issues of fault representativeness, (to what extent the errors induced are similar to those provoked by real faults?) and fault equivalence (to what extent distinct faults do lead to similar consequences, i.e., to similar errors and failures?) give rise to many problems [Arlat 2002]. Deliverable T2.2, [ETI2], focuses on this issues where in separate chapters the representativeness of hardware faults and software faults for different systems as well as the representativeness of operator faults in Transactional Systems is addressed.

Representativeness of software faults and representativeness of hardware faults have different flavors. Hardware faults are, in general, due to wear, production or external causes for which accepted fault models exist. Mobile and networked computers are particularly prone to external causes of failures. What makes these faults worth to be included in a representative faultload are their rates of activation in the considered operational environment. Therefore, we add here a few remarks on the role of fault rates when selecting representative hardware faults. For a more detailed discussion we refer to [ETI2].

As mentioned, the rates of hardware faults (and, perhaps, not only of hardware faults) are related to fault representativeness. Faults with high rates are, usually, more representative than faults with low rates. However, it depends on the scope of the benchmark. If the scope of the benchmark is to identify weaknesses, even faults with small rates may be important. If the scope is to evaluate availability, faults with high rates are important.

Fault rates may be simply not available from the open literature. If, however, some data is available, further questions arise. If only few data are available, what about their representativeness? Can we use modeling (formal methods) and extrapolation to increase their representativeness? Is sufficient estimated data available? When the available data is insufficient it is, certainly, not enough to say: "it may happen so we inject the fault".

We feel that the concept of a "*representative high level hardware fault*" is particularly relevant for dependability benchmaking. A high level hardware fault may, for example, be called representative, if its investigation gives answers to the same question using different hardware systems. Hence, what is representative depends on the benchmarking goal. Let us look at a simple example. Assume we ask which web server crashes more often. I.e., the web server is our DBT. We wish to compare a SUB (of which the web server is a part) supported by hardware H1 to the same SUB supported by hardware H2. H1 has ECC-memory and not-redundant disks; H2 has unprotected memory and RAID.

Now, a high level hardware fault could be *"web server crash due to storage faults"*. Hopefully, the physical fault rates λp_{mem} *(e.g., of bit-flips)* and λp_{disk} are the same for both hardware systems. (We assume here that both systems are based on the same technology.) However, the failure rate of the memory, λ_{mem}, *is 0* for H1 and the failure rate of the disk system, λ_{disk}, *is 0* for H2. Nevertheless, the high level hardware fault *"web server crash due to storage faults"* may have identical rates for H1 and H2, if λ_{disk} for H1 is equal to λ_{mem} for H2. In this case, both systems are equally good with respect to a benchmark (taking the view of the application), but not with respect to an internal FIE (taking the view of the hardware). Thus, the "representative hardware fault" for the a benchmark is the set of (at least) two physical faults.

When selecting a faultload one has to pay attention that faults should be related to their consequences, if they give rise to erroneous behavior. Clearly, faults for which it remains unclear what their consequences have been (e.g., where and when they become active), usually, are not part of a 'good' faultload. (This is true, in particular, for internal FIE's.) For example, consider kernel hangs. The challenge is how do we observe this? If the measuring software runs on the machine experiencing the kernel hang, it cannot measure /observe anything. If the measuring software runs on another machine, it can only watch the output of the machine under test/benchmark (e.g., does the machine still respond to SQL-queries). If there is no response, it cannot know whether the kernel hanged or crashed, the application hanged or crashed, the network is down etc. However, considering kernel behaviour may depend on the measure employed. If the benchmark measure is defined at the service level, it may not matter whether the kernel hanged or crashed. Furthermore, it is often not feasible to distinguish between exceptions that occur because of normal things happening (there are a lot of these) and those that occur because a fault was injected.

Thus, an important aspect is, where and how the (effects of the) selected faults can be observed. This is entirely dependent on the selected workload and is tied closely to the question of what and how to measure.

3.4 Portability

Another challenge faced when selecting an suitable work- and faultloads is portability. The question of portability of the work- and faultload arises since dependability benchmarking is used for comparing different systems. Hence, an important feature of any successful and fair dependability benchmark is to design it to be portable across as many target systems as possible. This is a difficult task in the case of performance benchmarks (hence, for workloads) and, much more so, in the case of dependability benchmarks.

For example, consider the portability of hardware faults. Some hardware faults may not have an effect on some types of hardware (e.g. bit flips injected into RAM will have no effect on ECC capable RAM). Some hardware may not be available (e.g., certain CPU-registers). Some hardware may cause problems, even if the application does not use it. For example, a defect network card may generate garbage on the PCI-bus, which in turn may disrupt disk/IO (disk controllers connected to PCI-bus), so even an application not using the network at all will be affected by a defective network card.

A solution here may be to inject, as mentioned, ‘high level faults’, which may come from a number of ‘lower level’ causes. This is again an issue of fault repesentativeness. In this effort , faults are represented by higher abstractions and then implemented for each particular instance.

Software (design) faults of hardware dependent OS parts are, clearly, not portable; other software faults might be portable in certain cases. Operator faults may not be portable between applications of different vendors (different configuration files, different user interfaces). A solution here may be again to inject ‘high level faults’, which may come from a number of ‘lower level’ causes. Such high level faults are then representations of fault classes. This again is an issue of fault representation.

3.5 Implementability

Finally, as already mentioned, for performing benchmarking experiments it is necessary that the selected faultload can be implemented. That is to say, it is necessary that the faults can be injected at the selected location and time and with their specified occurrence rates. Hence, the important question is to figure out whether a limited set of techniques can be identified that are sufficient to generate the relevant faultload according to the dimensions considered for the benchmark [Arlat 2002].

The task of designing portable workloads and portable and implementable faultloads is facilitated, if there is a simple and easy to use description of work- and faultloads. Ideally, the description should be such that a benchmark generator can automatically transform the description into a portable (instrumented) benchmark program.

Summarizing, the faultload for a DBE should ideally be:
- representative
- complete
- acceptable
- implementable
- portable
- repeatable
- adaptable.

On the other hand, it is sufficient that the fault load for a FIE is representative and implementable.
Thus, dependability benchmarking is a much more complex task than fault injection.

4 Synchronisation of Workload and Faultload

As soon as the appropriate work- and faultload is selected one has to decide how they are synchronized. Synchronisation helps in making the benchmarking process more efficient by avoiding to consider faults that have no effects on the SUB running the selected workload. So, if a new workload or a new faultload is selected synchronisation has also to be re-defined.

Synchronization is also necessary between the application under test and the background load. For instance, it is not useful to send SQL queries before the database server is up and running. Moreover, synchronization may be helpful to implement corrective operator actions/mistakes, as operator actions do not usually occur randomly but are triggered by certain observable states of the system. For example, when somebody notices, that a database server is no longer responding, an operator could logon to the server and try to find out what is wrong or simply restart the server.

Notice, that synchronization issues are dependent of the type of benchmark. For certain benchmark the synchronization can be loose; for other benchmarks one may implement a tight synchronization, depending on the needs and the details dependability benchmark configuration.

Synchronisation related questions are:

- How can faults be injected at "critical" locations/times to improve the "fault hit" ratio? (e.g. flipping memory bits by random will not make sense outside reachable regions).
- Do we need a synchronization language?
- Can we use such a mechanism to improve repeatability?.
- Can we use information available before starting experiments? (Maybe we know that an application does not use the disk. Then there is no need to inject disk faults.) This

can be important, for example if the application behaves differently in different states, this might be one way to "synchronize" when to inject different types of faults.
- Can we use formal methods to gain such information?
- Is it possible to calculate such critical locations/times based on some model or on field data and knowledge about the operation of the application.

As mentioned, depending on the aim of the benchmarking experiment, work- and faultloads might be tightly or loosely synchronized. Notice that the synchronization of workload and faultload is not relevant for probabilistic fault injection, but when speeding up the experiment. On the other hand, the synchronization of workload and faultload is useful and necessary for focused test cases.

Adaptable work- and faultloads

If we have some a priory knowledge on the target system or target application area, we might selectively use only some parts of the work- and faultload. For example, if we know that we always have experienced operators we might neglect operator faults. However, operator faults also depend on the application. Or, if we know that the software is tested well we might neglect software faults. If we know that an application does not use the disk, than there is no need to inject disk faults. Thus, to improve the benchmarking experiments the faultload should be adaptable. The specific knowledge for adaptations can be the basis of a knowledge base for dependability benchmarking experiments as part of the Dependability Benchmarking Management System (BMS). The knowledge base can make the information available before starting experiments.

Such knowledge can also be important, for example, if the application behaves differently in different execution states. This might be important to synchronize work and faultloads when to injecting different types of faults. One, probably, can use formal methods (model checking) to gain such information. The data base could also provide information relevant to calculate critical locations and injection times based on field data and knowledge about the operation of the application.

5 Examples of Work- and Faultloads

We now present several examples of work- and faultloads in order to show the work- and faultloads that can be used in different benchmarking experiments.

We present these examples in the context of specific DBE's in order to make them comprehensive. Therefore, we also include short descriptions of the applications, the experiment platforms and the fault injection methods. The discussion on the benchmarking goals, the benchmarking process, and the selection of suitable measures are topics addressed in deliverable [BDVEV1].

5.1 Distributed Transactional System Running Linux

In this experiment, we consider investigating a distributed transactional system running on top of the Linux operating system. Networked systems, like the system considered here, are, in general, very large and complex. They are, therefore, prone to failures, more than most other classes of computing systems. Nevertheless, we heavily rely on the availability of their services.

5.1.1 The System under Benchmarking

Our system under benchmarking is (an emulation of) the Virtual Server [Zhan 2000], see Figure 5.1.1 and Figure 5.1.2. The Virtual Server is a scalable server built on a cluster of real servers. The front-end of the real servers is a load balancer which schedules requests to the servers. The Virtual Server employs redundancy for fault-tolerance. For instance, the load balancer consists of two 'Directors', Figure 5.1.2.

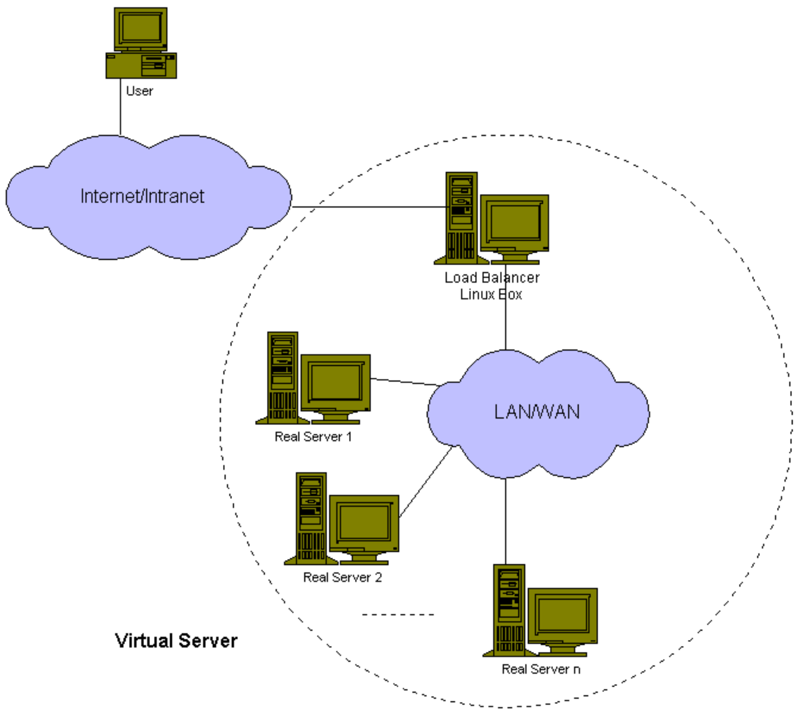

Figure 5.1.1 The emulated hardware configuration

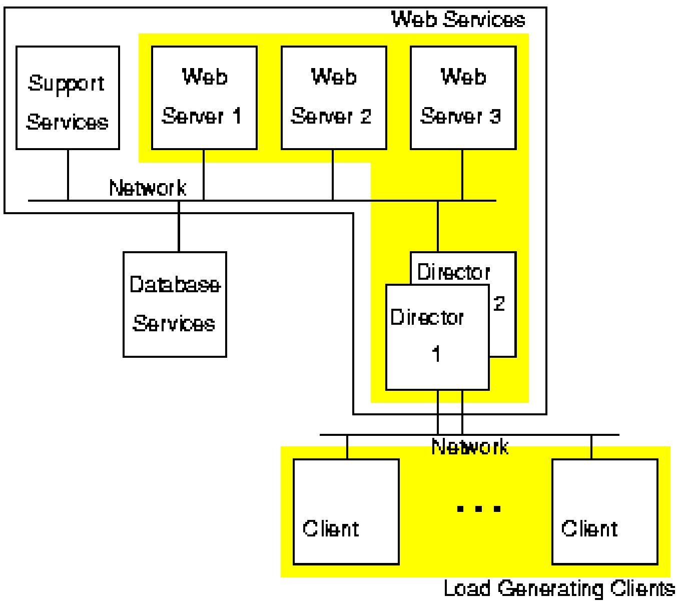

Figure 5.1.2 SUB

The services of web-based transactional systems consist of three basic groups:

- web services,
- database services,
- supporting services

The services provided by the web servers and the services provided by the load balancing Directors constitute the DBT. We use the well known open source product Apache as web server and either the open source product MySQL or the commercial Oracle as a database server. The support services (e.g., DNS) are part of the background load.

For our benchmarking experiments we need a realistic workload. The workload is generated by workload generating clients.

In addition to the above dependability benchmark configuration we will also considering a simplified set-up without database services.

5.1.2 Short Description of the Dependability Benchmarking Configuration

The benchmark will be conducted using the UMLinux framework [Buchacker 2001, Sieh 2002, Höxer 2002]. UMLinux is a framework capable of evaluating the dependability behaviour of networked machines running the Linux operating system in the presence of faults. (The Linux operating system is widely employed in networked server environments, for example as web- or mailserver.)

The core of the framework is a Linux simulator, which runs on top of a real world Linux machine as a single process and simulates a single machine running Linux. A second process paired with each simulated machine is used for fault injection. The framework is supported by a graphical user interface for experiment control. The framework uses software fault injection to inject faults into a simulated system of Linux machines. The simulation environment is made available to the experimenter by porting the Linux operating system to a new "hardware" the Linux operating system! Due to the *binary compatibility* of the simulated and the real system, any program that runs on the real system will also run on the simulated machine.

5.1.3 Work- and Background Load

The clients emulate end-users sitting in front of their workstations and making database requests via their web-browsers. Typical workload parameters are: transaction types, packet sizes, packet rates, or source/destination of packets.

The supporting services (the background load) are such as are necessary in a networked system (e.g. domain name service).

The system workload is based on the example systems described in TPC-C Version 5.0 [TPC01] from the Transaction Processing Performance Council. The workload described in TPC-C is used as the (external) workload for the web-based transactional system.

For the simplified system with web services only we consider using SPECweb99 [SPEC00] as workload. The workload will be generated by the clients which are part of the dependability benchmark configuration.

5.1.4 Faultload

We are mainly interested in fault coverage and fault propagation. The faultload will consist of high-level hardware faults injected into the hardware of the web- and database server machines as well as into the interconnection network. In order to simplify experimentation, the load generating clients and machines providing supporting services will, in this experiment, be exempt from fault injection. Thus, we are mainly interested in high-level hardware faults and hardware configuration faults. Later we will also consider operator and input faults (client faults).

The considered faults include hardware faults in computing core and peripheral devices of a single machine as well as faults external to machines, such as faults in external networking hardware, see Figures 5.1.3 and 5.1.4. For example, the faultload includes bit-flips in a machine's random access memory or central processing unit, defect blocks on storage devices (hard-disk, cd-rom, floppy etc.) and network send and receive failures.

Which specific consequences induced by the injected faultload are considered in each benchmarking experiment depends on the scope of the experiment. For example, if the scope is to view the server's behaviour from the client's point of view, the considered consequences (erroneous behaviour) can be classified into the following failure modes:

- faulty response (faulty record data)
- delayed response
- lost packets
- server error response
- server crash or hang

The last item is a server crash or hang from the clients point of view, i.e. the client is unable to evoke a response from the server until the end of the test run.

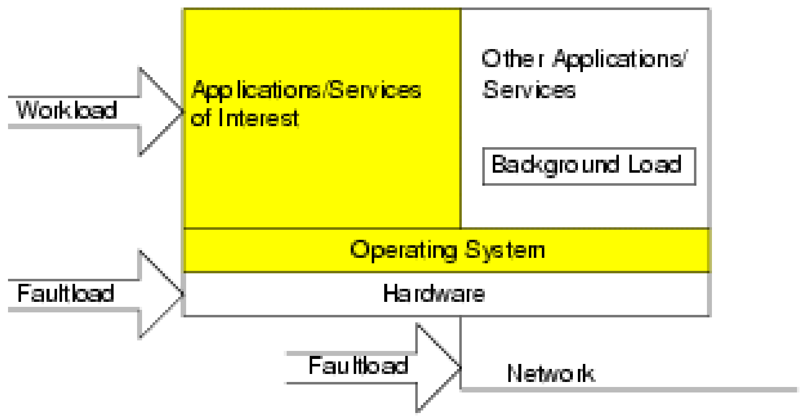

Figure 5.1.3 Workload, faultload and background load

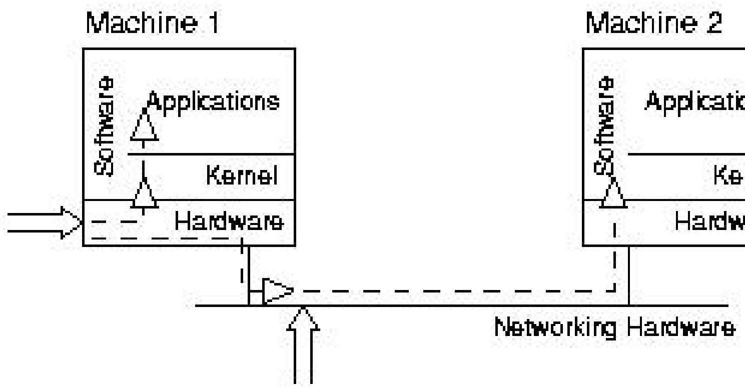

Figure 5.2.4 Error propagation. In each experiment, exactly one fault is injected in a single machine or the network hardware at a certain time.

The experiments will be conducted using the UMLinux framework [Buchacker 2001, Sieh 2002]. All the machines (hardware) are virtualised. The virtual machines run the Linux operating system and are binary compatible to the real world machines. All faults will be injected into the virtual machines.

Peripheral hardware access, such as hard-disk, floppy or cd-rom drive access is implemented using system calls of the Linux operating system. Thus the arguments of these system calls are checked to see if the faulty device is being accessed. If it is, the return value from a *read* or the data passed to a *write* is modified according to the fault model, for example to implement several defect blocks on a hard-disk. An inaccessible hard-disk is implemented by modifying the return value of the read system call to return an error. Of course the framework also allows to "inject" other types of faults, such as system configuration faults, site or environmental faults and interaction and operational faults. Since the framework is easily extensible, any well defined, representative fault can be injected.

5.2 Embedded System for Space Application

In this experiment we focus on the area of embedded systems; more precisely, we focus on the area of embedded control systems.

5.2.1 Brief Presentation of the Dependability Benchmarking Configuration

The benchmarking configuration consists of the hardware (an ordinary PC), the OS (Linux) and an application which is a model of satellite control and navigation system. This application was provided by Saab-Ericsson Space, who is an advisor for the project. The system under benchmarking is the operating system together with the application. The main reason for this choice is that the platform is easy to use; also the focus at this point is not to get representative results per se, but to define the process by which one can perform a benchmark.

For an end-user, the usefulness of a (computer) system is judged by the usefulness of the services provided. In our case, it is the services provided by an application running on top of an operating system. Therefore, the relevant level for measuring dependable behaviour is the service level. Example of indicators of such services could be the capability to handle interrupts within a certain time window or providing control signals to the process with a certain periodicity.

The experiment includes a stressload, consisting of a workload, a faultload and a background load. The application area very much controls the choice of workload for a benchmark and of course it affects the type of faultload/background load chosen. These concepts will be discussed in the following sections.

5.2.2 Workload Selection

In order to get a realistic evaluation of the target system a realistic application has been chosen that can run on top of the OS. The application chosen is a control and navigation system for a satellite. It was chosen because it is a realistic example of an embedded control system. Note that the full system cannot be used since the environment that it is designed for (space) cannot fully be simulated for use in a benchmark. Therefore, a simpler environment simulator is used.

The ODIN satellite has two different purposes, both astronomical research and atmospheric research. This means that the satellite must adapt to different objectives, looking out into space for astronomy and down at earth for atmospheric observations. The different objectives also give rise to different requirements on the accuracy of the attitude control system, causing the satellite to operate in different modes with different requirements. Also, a wide range of sensors and actuators are placed on the satellite [Berge 1997], differing in purpose and accuracy.

The workload is ported to C, from the original ADA control and navigation application, using POSIX-compliant mechanisms for providing concurrency and control of tasks. Since the Ccompiler is widespread to nearly all hardware and software platforms the issue of portability is made easier. Also, many OS's have POSIX-support (at least partially) which makes this application as portable.

The system consists of five main tasks of execution, implemented as POSIX pthreads [POSIX]. Two tasks are periodic tasks that handle the control algorithms, one is a background task and there is one task handling the on-board bus communication (simulated in software) and one task is the environment simulator, which drives the experiment and sends data to the control tasks via the bus interface. The system is illustrated in Figure 5.2.1. When conducting the experiments care must be taken so that the effects of the measurements are reduced, i.e., the amount of interference is reduced.

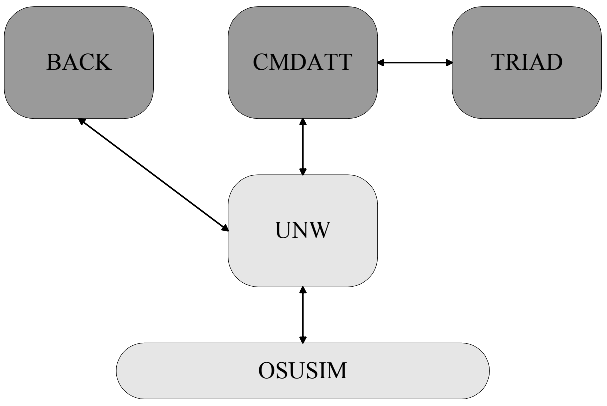

Figure 5.2.1: An overview of the software system used as workload in the benchmark. CMDATT and TRIAD are responsible for the control algorithms and execute at 16 Hz and 2 Hz respectively. UNW simulates the onboard network used for communication. BACK is a background process running on low priority.

The application we use is a representative of some, but not all, control applications. Therefore, in an external benchmark where results are made public, more applications than just one are needed. This approach is a similar approach as the one taken in common performance benchmarks like SPEC [SPEC] and EEMBC [EEMBC].

5.2.3 Faultload Selection

Basing a benchmark on the services provided by the workload is one way of trying to ensure that the results are fair. In order to make the benchmark effective it is important to find a faultload/background load that perturbs the parts of the OS that actually have an impact on the services provided. We have used a three step method for selecting a faultload/background load. This method only deals with external faults. As a consequence of this decision no internal hardware faults are part of the faultload. This is due to the fact that these faults are generally not dealt with at the OS level, and therefore they are not a key factor for the selection of OS's. Also, operator faults are not considered since there is generally no operator for embedded systems. The classes of faults being considered are thus external software and hardware faults. By using the three step process described below, relevant perturbations to the system can be identified.

An outline of the process is as follows:

1. Divide the services provided by the OS into suitable categories. An example of this division can be seen in Figure 5.2.2 .
2. Characterize how the workload makes use of the services provided by the OS. This can be accomplished by mapping the interactions between the OS and the workload to the different service categories identified in step 1.
3. Identify the relevant stresses for the components of the OS that are most heavily used by the workload, these are the most likely places to affect the services provided by the workload (which are the basis for our measurements). When a possible stress has been identified it is important to consider the representativeness of this stress to increase accuracy.

Steps 1 + 2 are closely intertwined and might benefit from being considered together, to avoid unnecessary details in the categorization of OS services. One benefit is that a common model for all target systems can be achieved using this method. This makes the stresses portable at a high level.

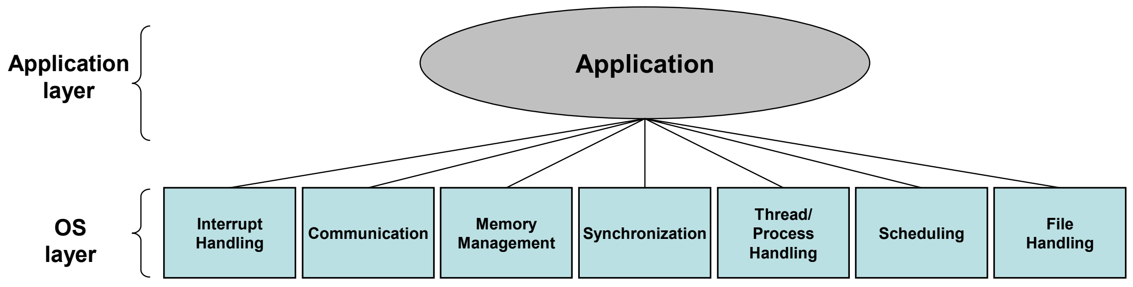

Figure 5.2.2: An example of how the OS can be divided into different components. Note that this componentization can be specific for a given application. This makes it easier to identify the relevant components for benchmarking purposes.

Figure 5.2.2 can be used as an example of how a faultload can be selected for a given application. For the control application used in this benchmark the main components used in the OS are Memory Management, Synchronization, Thread/Process Handling and Scheduling.

For instance the synchronization can give rise to the following faultload:

- High request rate to one semaphore
- Many semaphores used simultaneously, both by peer threads and other processes
- Releasing semaphores that are free
- Testing deadlock situations

This is just an example of how faults/stresses can be chosen. Of course the issue of representativeness must be considered as well. How and when each stress is applied is very important. All of the stresses in this example are possible to introduce using a user process/thread. As a start, stresses will be applied in a stream without restarting the system (unless it has crashed/hanged).

To increase the efficiency of the experiments some synchronization is needed between the workload and the faultload/background load. As a basic level of synchronization, the stresses must be applied when the workload is up and running at normal conditions and not before. Also, a stress must not be applied before the effect of the previous stress has been recorded.

5.3 Embedded System for Automotive Application

The application under study is a "diesel engine electronic control unit" (ECU). The figure 5.3.1 shows the inputs and outputs of the ECU. Nowadays, all functions in a modern diesel engine are controlled by the ECU communicating with an elaborate set of sensors measuring everything from engine speed to engine coolant and oil temperatures and even engine position.

5.3.1 The System Under Benchmarking

The SUB is the ECU. Its outputs need to be amplified. For that purpose a power amplifier is used. The ECU is implemented as a SoC. The structure of the SoC is as shown in figure 5.3.2.

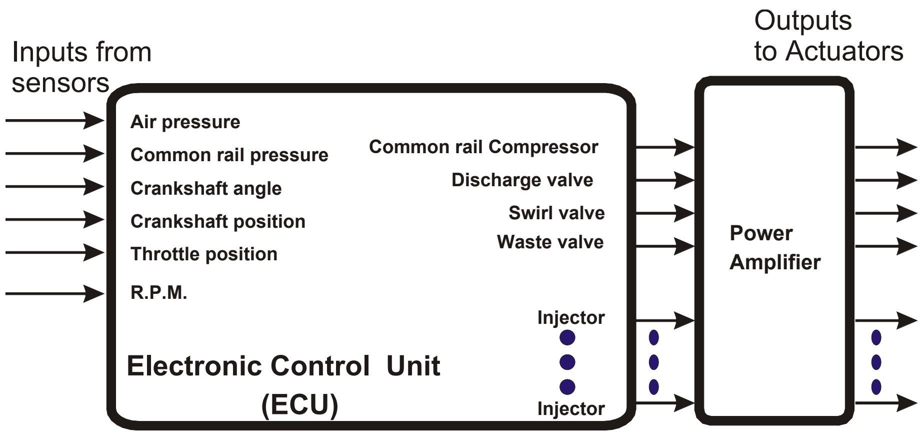

Figure 5.3.1. ECU inputs and outputs

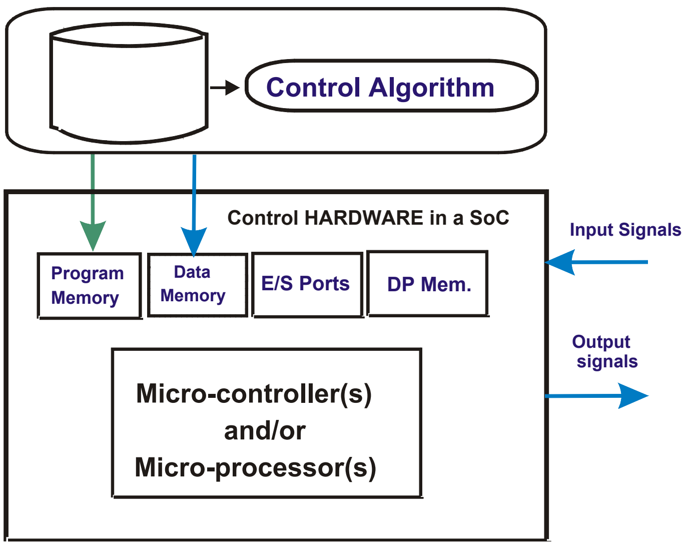

Figure 5.3.2 Architecture of the ECU.

5.3.2 The Dependability Benchmark Target

The DBT is the applicative software, i.e. the Control Algorithm. In our case study the control algorithm is implemented using a Real Time Operating System (RTOS) as well as without using an RTOS. So, we consider two SUB's intending to compare them.

In any case, the control algorithm constitutes the ECU functionality. The ECU functions can be described as follows.

The ECU gets from the sensors:

1. The pressure in the common rail.
2. The engine speed.
3. The throttle position.
4. The intake air pressure.
5. Crankshaft angle.
6. Crankshaft position.

Depending on these values the ECU evaluates:

- ➢ The quantity of diesel that must be injected into the diesel engine.
- ➢ The injection angle at which the diesel must be injected.
- ➢ The injection duration time.
- ➢ The new pressure in the common rail.
- ➢ The new intake air pressure.

By means of the actuators the ECU controls this parameters.

As a summary the ECU functions are:

- ➢ Get values from the sensors.
- ➢ Evaluate the new parameters of the engine
- ➢ Control the actuators to obtain the right conditions in the system.

In order to implement the ECU control algorithm, it can be considered as two independent processes: the pressure injection control loop (ICL) and the air management control loop (AMCL).

The ECU control algorithms for both processes can be described by means of five tasks:

- *First task:* In this task it is necessary to *lookup data* (lookup in a table*)* and to perform *data interpolation*. Its aim is to calculate the amount of fuel to inject. The input parameters are *engine speed* and *throttle position*. The *engine speed* is calculated synchronously while the *throttle position* is obtained from a sensor through an analogue to digital converter channel.
- *Second and third tasks:* Crankshaft's tooth management and angle to time calculation that run synchronously with the crankshaft of the engine. The first of them continuously recalculates and updates engine speed, while the second schedules fuel injection pulses according to this updated value and the results of the table lookup and interpolation tasks.
- *Fourth task:* This task is a control loop. The pressure injection control loop (PICL) regulates the common-rail pressure. Its output signal actuates over a discharge valve to keep the pressure on the reference.
- *Fifth task:* This task controls air management. This is a bit manipulation task that actuates on waste-gate and swirl valves to optimise the performance of the engine.

These tasks run concurrently on the micro-controller (Motorola PowerPC MPC 555). Tasks two and three (tooth management and angle to time calculation) run synchronously with the rotation of the engine, and thus, the worst case for these tasks is the case in which the engine runs at its top speed.

On the other hand, tasks one, four and five (table lookup and interpolation, PI control loop and bit manipulation) are executed at predefined periods to update the engine parameters.

5.3.3 Dependability Benchmarking Configuration

In order to inject faults and evaluate performance the dependability benchmarking configuration will consist of two PC's and a PCB with the SoC which has the embedded automotive application. The printed circuit board (PCB) which holds the SoC is necessary in order to provide the power supply to the SoC and connect the SoC pins to the NEXUS (see below) connection and with the PC ports.

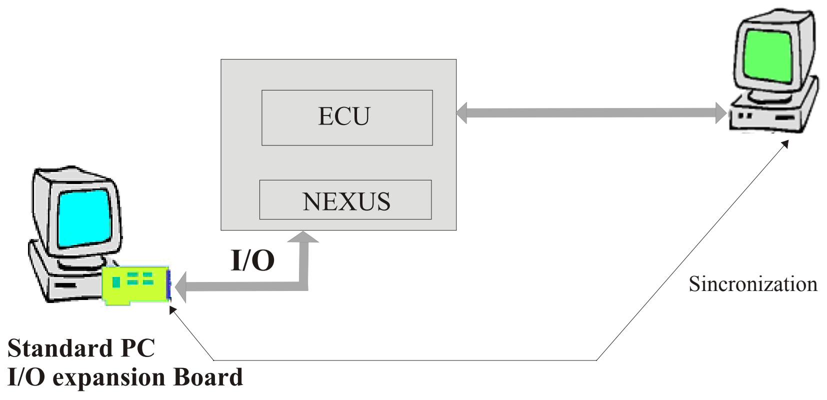

Figure 5.3.4 Dependability Benchmarking Configuration

In order to access the engine status register for fault injection we are going to use NEXUS [Nexus].
Taking into account NEXUS capabilities we will use a NEXUS-based SWIFI technique for injecting faults.

In the case of the synchronisation, a personal computer (PC) (with NEXUS) will take charge of deciding when and where the fault is going to be injected and its temporal duration (transient or permanent in the case of hardware faults). So in the case of hardware faults the PC will randomly choose the time of the injection and the fault location: CPU registers, code or data memory. NEXUS has interesting characteristics such as the ability to set breakpoints and watch-points, read and write memory locations without stopping the execution and with a minimal impact over the system, or to access to the instruction trace information with acceptable impact to the system under development. We are going to exploit these NEXUS characteristics to develop a new software fault injection strategy. With the trace capabilities of the NEXUS standard the PC will recollect the execution trace of the application after the fault is injected in order to compare with a golden run. Also, the fault tolerance mechanisms of the micro-controller and the OS will detect some of the errors.

5.3.4 Workload

In this application we have two control points to stress: the *engine speed* (R.P.M.) that reduces the cycle to do the big part of the application and the *timer* that specifies the time loop control for the common rail pressure. With high engine speed and short time for the common rail pressure loop we can obtain a heavy stress load.

On the other hand, to implement the tasks that constitutes the control algorithm of the embedded automotive application we will use a set of functions which are used for benchmarking purposes in automotive industry. Such functions are a subset of the EEMBC [EEMBC] benchmark. EEMBC is a set of algorithms that are the most used in car and industrial applications.

The functions we will use to implement the five tasks defined above for the *ECU* are:

1. Table lookup & interpolation
2. Angle to time conversion
3. Pulse width modulation (PWM)
4. Tooth to spark
5. Road speed calculation
6. Bit Manipulation.

5.3.5 Faultload

In embedded applications we mainly have to consider hardware faults and Operating System faults. There are no operator faults to be considered because a typical embedded industrial application has no interaction with an operator.

In the special case of the embedded automotive application, the memory of the system and the I/O lines are the best place to inject faults. However, it is important to notice that in most cases faults injected on the I/O lines will be represented by faults in memory. Therefore, using SWIFI fault injection methods could be sufficient to generate the fault load. Faults will be injected into system memory since the state of the system is stored in system memory.

The critical data needed by the engine is stored in a special register, called the engine status register.

In the case of OS faults, the PC will take charge of deciding which system call to corrupt. When the system call is chosen, the PC will also decide the parameter to corrupt (in case the system call has more than one parameter). As in the first case, after the injection is produced the system can detect the error via the fault tolerance mechanisms of the micro-controller or the OS return error codes. Also the trace capabilities of the NEXUS standard will allow us to compare the execution after the fault with a golden run.

As soon as NEXUS can be used to synchronize and to observe the behaviour of the system in presence of faults we can use also pin level forcing techniques for injecting faults if needed.

6. Conclusion

We discussed some of the many challenges which one encounters when designing suitable work- and faultloads for dependability benchmarks. We also presented some examples of experiments in the focused area. When we have performed the experiments, more information on dependability benchmarking processes is, hopefully, available which helps to meet the challenges.

Succinctly put, selection of a good workload, and more so of a good faultload is, in general, a non-standard pragmatic process based on knowledge, observations, and reasoning about the system functionalities and structure, and about the constraints induced by the operational environment. Dependability benchmarking is about dependability *and* performance; and it is the process of measuring and comparing. Unless the system services, its architecture, its organisation and its operational environment is very well understood, no one can solve the (difficult) basic problems of defining dependability benchmarking processes.

Nevertheless, some general guidelines for selecting work- and faultloads can be extracted from the above. For example:

- In order to get a realistic evaluation of the system under benchmarking a realistic application has to be chosen that can run on top of the SUB.
- Widely accepted performance benchmarks can provide the basis for selecting application specific workloads also for dependability benchmarking.
- Basing a benchmark on the services provided by the workload is one way of trying to ensure that the results are fair. In order to make the benchmark effective it is important to find a faultload/background load that perturbs the parts of the SUB that actually have an impact on the services provided.
- Identify the relevant stresses for the components of the SUB that are most heavily used by the workload, these are the most likely places to affect the services provided by the SUB.
- Ideally, when benchmarking a system all existing faults should be considered in the first place. However, what faults to incorporate in the faultload depends, for example, on whether an external or an internal benchmarking experiment is to be performed.
- The selected faultload for a specific dependability benchmarking experiment should be widely acceptable.
- Knowledge of representativeness of faults is necessary for selecting a ‘good’ faultload.
- If the aim of the benchmark is to identify weakness, even faults with small rates may be important. If the aim is to evaluate availability, faults with high rates are important.
- When selecting a faultload one has to pay attention that faults can be related to their consequences, if they give rise to erroneous behavior.
- It is necessary that the selected faultload can be implemented.
- Thus, we have to distinguish between the set of recommended faults that should be considered in a benchmarking experiment and the actual implementable faultload for the experiment.
- A typical hardware faultload includes bit-flips in a machine's random access memory or central processing unit, defect blocks on storage devices (hard-disk, cd-rom, floppy etc.) and network send and receive failures.
- Which specific faultload considered in each benchmarking experiment depends on the scope of the experiment. For example, if the scope is to view the server's behaviour from the client's point of view, the considered erroneous behaviour can be classified into the following failure modes: faulty response (faulty record data), delayed response, server error response, server crash or hang.
- Other types of faults are system configuration faults, site or environmental faults and interaction and operational faults.
- If we have some a-priory knowledge on the target system or target application area, we might selectively use only some parts of the work- and faultload.

7. References

[CF1 2001] J. Arlat et al., "Conceptual Framework : State of the Art", Deliverable CF1 DBench Project IST 2000–25425, August 2001

[CF2 2001] H. Madeira et al. "Preliminary Dependability Benchmark Framework" DBench Project IST 200-25425, September 2001

[ETIE1] Deliverable on *Measurement,s* DBench Project IST 2000-25425, June 2002

[ETIE2] Deliverable on *Fault Representativenes,s* DBench Project IST 2000-25425, June 2002

[BDEV1] Deliverable on *Dependability Benchmark Definition: DBench prototypes*, DBench Project IST 2000-25425, June 2002

http://www.laas.fr/dbench/delivrables.html.

[Arlat et al. 2001] J. Arlat, K. Kanoun, H. Madeira, J. V. Busquets, T. Jarboui, A. Johansson and R. Lindström, State of the Art, Available at http://www.laas.fr/dbench/delivrables.html, DBench Project IST 2000-25425 Deliverable, N°CF1, September 2001 (Also LAAS Report 01-605).

[Arlat 2002] J. Arlat, From fault injection experiments to dependability benchmarking, position paper, workshop on Challenges and Direction for Dependable Computing, IFIP WG 10.4, 2002

[Berge 1997]: S. Berge, O. Jirlow, P. Rathsman, F. Schéele. "Advanced Attitude Control on Swedish Small Satellite Odin", 48th Internaltional Astronautical Congress, Oct 6-10, 1997.
.

[Bovet 2000] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 2000.

[Buchacker 2001] Kerstin Buchacker , Volkmar Sieh, Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects 2001, Proceedings Third IEEE International High-Assurance System Engineering Symposium HASE-2001,pp 95-105

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

[Conn 2001] Conntectix Corporation. Virtual PC. URL: http://www.connectix.com/, 2001.
.

[DeVale 2002] J. DeVale and P. Koopman " Performance Evaluation of Exception Handling in I/ O Libraries", Int. Conf. on Dependable Systems and Networks (DSN 2001), p. 519-525, 2001

[EEMBC] http://www.eembc.org EEMBC is a registered trademark of the Embedded Microprocessor Benchmark Consortium

[Höxer 2002] H.J. Höxer, K. Buchacker, V. Sieh, UMLinux: a tool for testing a Linux system's fault tolerance, IMMD3 University of Erlangen-Nürnberg, June 2002

[Koopman 2000] P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems" 29th Int. Fault-Tolreant Computing Symposium (FTC 1999), .p. 30-38, 1999

[Madeira et al. 2001] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johanson and R. Lindström, Preliminary Dependability Benchmark Framework, http://www.laas.fr/dbench/delivrables.html, DBench Project IST 2000-25425 Deliverable, N°CF2, September 2001 (Also LAAS Report 01-604).

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

[Miller 1995] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revised: A re-examination of the reliability of UNIX utilities and services. Computer Science Technical Report 1268, University of Wisconsin-Madison, 1995.

[NEXUS] www.ieee-isto.org/Nexus5001/standard2.html

[POSIX] IEEE Std. 1003.1b 1993. IEEE Standard for Information Technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API) -- Amendment 1: Realtime Extension [C Language], 1994.

[POSIX]: B. Nichols, D. Buttlar, J. Farrell. “Pthreads Programming”, O’Reilly & Associates, 1996.

[Rosenblum 1995] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The simos approach. *IEEE Parallel and Distributed Technology*, Fall, 1995.

[Sampson 1998] J. R. Sampson, W. Moreno, and F. Falquez. A technique for automatic validation of fault tolerant designs using laser fault injection. In *Proceedings of the 28th IEEE International Symposium on Fault Tolerant Computing*, pages 162-167, 1998.

[Scheele 1996] F. von Scheele. “The Swedish Odin Satellite to Eye Heaven and Earth”, 47th International Astronautical congress, IAF, Oct 7-11, 1996.

[Sieh 2002] Volkmar Sieh, Kerstin Buchacker, Testing the Fault-Tolerance of Networked Systems 2 International Conference on Architecture of Computing Systems, ARCS-2002, pp 37-46

[SPEC00] Standard Performance Evaluation Corporation SPECweb99 Release 1.02, 2000 http://www.spec.org

[TPC01] Transaction Processing Performance Council
TPC Benchmark [tm] C, Standard Specification, Revision 5.0, February 26, 2001

[Weiderman 1989] Nelson Weiderman. "Hartstone: Synthetic Benchmark Requirements for hard Real-Time Applications". Technical Report CMU/SEI-89-TR-23 ADA219326, Software Engineering Institute (Carnegie Mellon University), 1989.

[Zhan 2000]Wengson Zhang, Linux Virtual Server for Scalable Network Services, 2000, Ottawa Linux Symposium. www.linuxvirtualserver.com