DBench

Dependability Benchmarking

IST-2000-25425

Dependability Benchmark Definition: DBench prototypes

Report Version: Deliverable BDEV1

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), University of Coimbra (Portugal), Friedrich Alexander University, Erlangen-Nürnberg (Germany), LAAS-CNRS (France), Polytechnic University of Valencia (Spain).

Sponsor: Microsoft (UK)

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

Table of Contents

Dependability Benchmark Definition: DBench prototypes

Authored by: H. Madeira++, J. Arlat*, K. Buchacker◆, D. Costa+, Y. Crouzet*, M. Dal Cin◆, J. Durães++, P. Gil◆◆, T. Jarboui*, A. Johansson**, K. Kanoun*, L. Lemus◆◆, R. Lindström**, J.-J. Serrano◆◆, N. Suri**, M. Vieira++

* LAAS ** Chalmers ++ FCTUC ◆ FAU ◆◆ UPVLC +Critical

June 2002

Abstract

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. The main components of a dependability benchmark are measures, workload, faultload, procedures & rules, and the experimental benchmark setup. Thus, the definition of dependability benchmark prototypes consists of the description of each benchmark component.

This deliverable presents the dependability benchmark prototypes that are being developed in the project, which cover two major application areas (embedded and transactional applications) and include dependability benchmarks for both key components (operating systems) and complete systems (systems for embedded and transactional applications).

We start by proposing a set of guidelines for the definition of dependability benchmarks that have resulted from previous research in the project. The first step consists of defining the dimensions that characterise the dependability benchmark under specification in order to define the specific context addressed by the benchmark (i.e., a well defined application area, a given type of target system, etc). The second step is the identification of the dependability benchmark measures. The final step consists of the definition of the remaining dependability benchmark components, which are largely determined by the elements defined in the first two steps.

We propose two complementary views for dependability benchmarking that will be explored in the prototypes: external and internal dependability benchmarks. The first ones compare, in a standard way, the dependability of alternative or competitive systems according to one or more dependability attributes, while the primary scope of internal dependability benchmarks is to characterize the dependability of a system or a system component in order to identify weak parts. External benchmarks are more demanding concerning benchmark portability and representativeness, while internal benchmarks allow a more complete characterization of the system/component under benchmark.

The dependability benchmarks defined include four prototypes for the application areas addressed in the project (embedded and transactional) and one prototype for operating systems, which is a key component for both areas. This way we cover the most relevant segments for both areas (OLTP + web-based transactions and automotive + space embedded systems) and provide the consortium with means for a comprehensive cross-exploitation of results.

1 Introduction

This deliverable describes the dependability benchmark prototypes that are being developed in DBench. As planned, these prototypes cover two major application areas (embedded and transactional applications) and are being developed for two different families of COTS operating systems (Windows and Linux). Additionally, a real-time operating system will be also used in at least one of the prototypes for embedded applications.

The starting point for the definition of the dependability benchmark prototypes is the framework initially defined in the deliverable CF2 [CF2 2001]. This framework identifies the various dimensions of the problem and organizes these dimensions in three groups: categorisation, measure and experimentation dimensions. The *categorisation* dimensions organize the dependability benchmark space and define a set of different benchmark categories. The measure dimensions specify the dependability benchmarking measure(s) to be assessed depending on the categorization dimensions. The *experimentation* dimensions include all aspects related to the experimentation steps of benchmarking required to obtain the benchmark measures. Additionally, dependability benchmarks have to fulfil a set of properties (identified in CF2) such as portability, representativeness, and cost. In this context, the definition of a concrete dependability benchmark (such as the prototypes presented in this deliverable) consists of the instantiation of the framework to a specific application domain or to a particular kind of computer system or component.

In addition to the dependability benchmark framework initially defined in WP1, the benchmark prototypes described in the present deliverable are a direct result of the research work developed in WP2 concerning benchmark measures, fault representativeness, and workload and faultload generation. In this sense, WP2 deliverables identify the different possibilities to be explored in the benchmark prototypes during WP3, not only to refine the conclusions of the research developed in WP2 but also to provide real benchmarking environments to validate the different approaches and techniques that have resulted from WP2.

The structure of this deliverable is the following: Section 2 presents the guidelines proposed for the definition of dependability benchmarks and the following sections are devoted to the definition of the dependability benchmark prototypes. Section 3 presents the dependability benchmark prototype for general purpose operating systems. Section 4 defines the prototypes for transactional applications, addressing both On-Line Transactional Processing (OLTP) applications and web server applications. Section 5 presents the prototypes for embedded applications, including two types of embedded applications: space and automotive control applications. Section 6 summarises the deliverable.

2 Guidelines for the definition of dependability benchmarks

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. The computer system/component that is characterized by the benchmark measures is called the system under benchmark (SUB). The main components of a dependability benchmark are:

- Measures

- Workload
- Faultload
- Experimental benchmark setup
- Procedures and rules

In this way, a dependability benchmark consists of the specifications of the benchmark components. This specification could be just a document. What is relevant is that one must be able to implement the dependability benchmark (i.e., perform all the steps required to obtain the measures for a given system or component under benchmarking) from that specification. Obviously, the benchmark specification may include source code samples or even tools to facilitate the benchmark implementation.

The way dependability benchmark dimensions were organized in the dependability benchmark framework [CF2 2001] implicitly provides a natural sequence of steps for the definition of a specific dependability benchmark (this can also be seen in [ETIE1 2002]).

1. **First** we have to define the set of dimensions that characterise the dependability benchmark under specification. That is, a real benchmark has to be defined for a very specific context (a well defined application area, a given type of target system, etc), which is described by a specific instantiation of the categorization dimensions.

2. The **second** step is the definition of the dependability benchmark measures, which should be the first dependability benchmark element to be specified. Different dependability benchmark categories will have different set of measures.

3. In the **third** step we define the remaining dependability benchmark elements (workload, faultload, procedures & rules, and the experimental benchmark setup). Although the specification of these benchmark elements is not always directly dependent on the benchmark measures, some dependencies are expected on the measures defined in the second step. That is, the definition of the workload and faultload, for example, is normally related to the measures of interest.

2.1 Defining categorization dimensions

A very important decision in the definition of the categorization dimensions is the choice about the benchmark scope (see discussion in [ETIE1 2002]). As defined in [CF2 2001], benchmark results can be used **externally** or **internally**.

External use means that benchmark results are standard results that fully comply with the benchmark specification and can be used to compare the dependability of alternative or competing solutions. Internal use means that the dependability benchmark is used as a tool to characterize the dependability of a system or a system component in order to identify weak parts, help vendors and integrators to improve their products, or to help the end-users to tune their systems.

This distinction is very important for the definition of dependability benchmarks and lead to two very different families of benchmarks:

- **External benchmarks**: the main goal of this type of benchmarks is to compare, in a standard way, dependability aspects of alternative solutions (either systems or components). These benchmarks are the most demanding ones concerning benchmark portability and representativeness of results, as these properties are of utmost importance to have easy and meaningful comparisons. In a way, the definition of actual external benchmarks represents the ultimate goal of the research work in DBench.

- **Internal benchmarks:** the primary goal of internal benchmarks is to assess dependability attributes of computer systems/components in order to validate specific mechanisms, identify weak points, etc. As most of the dependability attributes are related to specific features of the system under benchmark, this type of benchmarks need to know the target system in detail to allow the evaluation of quite specific measures such as dependability measures directly related to fault tolerance mechanisms available in the system. For this reason, internal results are not generally portable and are difficult to be used to compare different systems.

External benchmarks involve standard results for public distribution while internal benchmarks are used mainly for system validation and tuning. Thus, external benchmarks aim at finding the best existing solution and the 'best practice' by comparison while internal benchmarks aim at improving an existing solution. A drawback of external benchmarks is that they may not yield enough information for improvements; a drawback of internal benchmarks is that certain weak points may not get detected when looking at only one possible solution. Therefore, internal and external benchmarks should complement each other (note that internal benchmarks may be used for comparison across systems as well, but in a much more limited and focused way than external benchmarks).

In addition to the external versus internal benchmarks, the **application area** is another categorization dimension that has a strong impact on the benchmark definition (thus, it must be clearly defined in the first step mentioned above). In fact, the division of the application spectrum into well-defined application areas is necessary to cope with the huge diversity of systems and applications and to make it possible to make choices on the other dimensions and benchmark components. In fact, most of the dimensions and dependability benchmark components are very dependent on the application area. For example, the benchmark measures, the operational environment, the most common (external) faults that may affect the systems, and the workload (just to name a few of them) are very dependent on the application area.

The relevance of the application area in the dependability benchmark definition is also related to the fact that application area is very important to characterize the **system under benchmark** (SUB), which is the third categorization dimension that has to be defined in the first process step.

Although the application area is very important to characterize the SUB, the way the SUB is defined in a dependability benchmark specification is very dependent on the type of benchmark: external or internal. The following guidelines for the definition of the SUB in dependability benchmark specifications are proposed:

- For external benchmarks the SUB is normally defined in a **functional way** in terms of typical functionalities and features, including dependability features that are expected to be found in the set of possible targets addressed in the benchmark. For example, for

transactional applications one can assume that the SUB can be any system able to execute transactions according to the typical transaction properties (atomicity, consistency, isolation, and durability, known as ACID properties), which include, for example, data recovery features (durability property). As it is easy to see, is an external benchmark was based on a structural view of the SUB its portability would be very reduced, as different systems tend to have different structures.

- Internal benchmarks may assume a much more detailed knowledge of the SUB. Nevertheless, the description of the SUB (in the benchmark specification) has to be limited to a give abstraction level. In practice, the SUB architecture can be described as a set of components that perform specific functions in the target system (i.e., it include a structural view of the system to a given abstraction level). In addition to the generic layered description of computer systems (e.g., hardware, operating system, middleware, applications), which can be more or less detailed (e.g., the operating system is composed by a micro kernel, drivers, etc.), we will be particularly interested in the description of the key components for dependability benchmarking. This way, an internal benchmark may explicitly assume that the target system has specific error detection components, fault diagnosis, system reconfiguration, error recovery, etc. Although very specific implementation details on these components should not be considered (otherwise the benchmark will be reduced to zero), the benchmark may assume the knowledge of specific techniques that are used (or not) in the SUB. For example, the error detection can be based on structural redundancy and voting or it can be just a set of behavioural checks.

As a summary, the first step in the definition of a dependability benchmark is to choose the three most important categorization dimensions:

- **Result scope**, which leads to two different families of benchmarks: external and internal.
- **Application area**, which focus the benchmark definition in specific application segment and helps the definition of many benchmark components.
- **System under benchmark:** identifies the concrete object that is supposed to be characterized by the benchmark measures (defined in the second step).

Obviously, all the other characterisation dimensions [CF2 2002] have to be identified in order to define a dependability benchmark. What we argue here is that the three dimensions above are the most important ones for the benchmark characterization.

2.2 Definition of benchmark measures

Dependability benchmark measures are detailed in [ETIE1 2002]. In this section we just propose a set of simple guidelines to help the definition of measures in real benchmarks. The first rule is that the type and nature of the measures are very dependent on the type of benchmark (external or internal). The following points present the general guidelines behind the definition of measures for each type of benchmark:

- Measures for external benchmarks:

  - Based on the functional view of the SUB. That is, measures should be based on the service provided by the SUB during the benchmark process. Failure modes could be included as failure modes capture the impact of faults on service provided by the SUB.
  - Quantify the impact of faults (faultload) on the service provided.
  - Reflect an end-to-end perspective (for example, the end-user point of view)
  - Small set of measures and easy to implement and understand to facilitate comparison.
  - Measures should not be extrapolated or inferred: measures are computed from based direct observations from the SUB).

- Measures for internal benchmarks:
  - No specific restrictions in defining measures for internal benchmarks (i.e., we may have a very large number of possible measures).
  - Characterize dependability attributes, specific dependability features (efficiency), or failure modes (i.e., can characterize any measure defined in [ETIE1 2002], which is not the case for external benchmarks).
  - When running the benchmark, the user is allowed to collect just a subset of the measures defined in the benchmark (e.g., the measures just needed to identify a possible weak point in the system).

For a more details about the different types of benchmark measure see [ETIE1 2002].

2.3 Definition of benchmark components

In addition to the benchmark measures addressed in the previous section, the definition of a dependability benchmark includes the following major benchmark components: workload, faultload, experimental benchmark setup, and procedures & rules. The first three components are discussed in detail in [ETIE3 2002]. In this section we just summarize the key guidelines for the definition of these components and briefly discuss the definition of the procedures and rules that have to be included in the dependability benchmarks specifications.

The definition of both the workload and faultload are very dependent on the way the SUB is defined in the benchmark.

- SUB defined in general terms or based on a functional view (case of external benchmarks):
  - **Workload** defined in general terms or in functional way (e.g., a specification) or in a standard language. This is the way to deal with the lack of details about the SUB and to assure workload portability.
  - **Faultload** defined as general high-level classes of faults. For example, for hardware faults we could have bit-flip faults and stuck at faults. For software faults we could have mutations according to general software fault classes such as ODC classes (see [ETIE2 2002] for the discussion of representativeness of this type of faults). An important aspect concerning this high-level way of defining faultload is that a given

fault is not equivalent when ported from one system to another. The hypothesis that will be investigated in DBench prototypes is whether it is possible to achieve statistical equivalence when the same faultload is applied to different SUB or not. The size of the fault sets used in the faultload is another important aspect, as it has a clear impact on the time needed to perform the benchmark.

- SUB defined in a structural way to a given abstraction level, including details about specific components (case of internal benchmarks). In this case the workload and the faultload don't have the limitations mentioned above for the case of external benchmarks. This is particularly relevant for the faultload, as in this way the faultload can include very specific faults related to the evaluation/validation of specific mechanism in the SUB. For a more detailed discussion on faultload and workload selection for this case see [ETIE3 2002].

One of the things that must be clearly defined in the benchmark specification is the complete set of systems, components, and tools that are required to perform the benchmark experiments and get the measures. This is called the **experimental benchmark setup.** The precise set of elements required for the experimental benchmark setup is dependent on the specific dependability benchmark. However, the experimental benchmark setup always includes at least the following:

- **System under benchmark**, as defined above. This is the system where the measures apply. Note that the SUB could be larger than the component or subsystem that the benchmark user wants to characterize. For example, if the SUB is a transactional server the measures characterize the whole server, which includes the transactional engine, the operating system, and the hardware (just to name the key layers). This does not mean that the goal of the benchmark is to characterize the operating system or the hardware platform used by the transactional server. It means that the SUB is composed by all the components needed to execute the workload. In order to isolate the effects on the measures of a given component of the SUB it is required to perform benchmarking in more than one specific SUB configurations, where the only difference between each configuration is the component under study. This is also the model adopted in existing performance benchmarks.

- **Benchmark management system** (BMS), as the system in charge to manage all the benchmarking experiments. The goal is to perform the benchmarking in a completely automatic way. The tasks assigned to the BMS must be clearly defined in the benchmark specification (but are very dependent on the actual benchmark).

The last aspect is related to the definition of the **procedure and rules** required to implement and run the dependability benchmark. This is, of course, dependent on the specific benchmark but the following points give some guidelines on specific aspects needed in most of the cases:

- Standardised procedures for "translating" the workload and faultload defined in the benchmark specification into the actual workload and faultload that will apply to the system under benchmarking.

- Uniform condition to build the experiment benchmark set up, perform initialization tasks that might be defined in the specification, and run the dependability benchmark according to the specification (i.e., apply the workload and the faultload).

- Rules related to the collection of the experimental results. These rules may include, for example, available possibilities for system instrumentation, degree of interference allowed, common references and precision for timing measures, etc.
- Rules for the production of the final measures from the direct experimental results, such as calculation formulas, ways to deal with uncertainties, errors and confidence intervals for possible statistical measures, etc.
- Scaling rules to adapt the same benchmark to systems of very different sizes. These scaling rules would define the way the system load can be changed. At first sight, the scaling rules are related to baseline performance measures and should mainly affect the workload, but one task of the experimental research planned for the prototypes is to investigate the need of scaling up or down other components of the dependability benchmark to make it possible to use the same benchmark in systems of quite different sizes.
- System configuration disclosures required to interpret the dependability benchmark measures. In principle, this will be particularly needed for the baseline performance measures, but some dependability measures might also include requirements or disclosures involving all the factors that affect dependability.
- Rules to avoid "gaming" to produce optimistic or biased results.

The next sections present the different dependability benchmark prototypes that are being developed in DBench. The present descriptions correspond to the preliminary definitions of the prototypes. The research work planned for the rest of WP3 will evolve these prototypes into eventually true examples of dependability benchmarks.

3. Dependability benchmark prototype for operating systems

Operating systems (OSs) form a generic software layer that provides basic services to the applications through an application programming interface (API). Other main interactions of interest span the underlying hardware layer and the communication links with the drivers.

Operating systems may not only differ in their internal architecture and in their implementation, but also in the set of services they offer to applications. Nevertheless, the OS API forms a natural location through which the OS robustness with respect to applications can be assessed.

In addition, the impact of faults affecting the supporting hardware layer on the behaviour of the OS is worth investigating, too. Based on previous related research and as was further confirmed by the investigations carried in WP2 (e.g., see [ETIE2 2002], section 3), it can be confidently assumed that hardware faults can be emulated by the software-implemented fault injection (SWIFI) technique.

Furthermore, as was identified in WP2, in particular in the framework of the related experiments conducted for analysing techniques for generating faultoads that can emulate the effects of real faults that may impact an OS (see [ETIE2 2002], section 4), it is unlikely that the emulation of driver faults can be achieved via the API or via single bit-flips in the kernel space. Accordingly, more elaborate techniques focusing on the “inside” of the OS should be considered. As an alternative to the application of bit-flips affecting the parameters of the calls of the internal

functions of the kernel, that were investigated (which is seldom possible in the case of a "black box" kernel), as a preliminary and pragmatic approach, we propose to rely first on the application of bit-flips in the memory space of the drivers. Further investigations will be conducted to devise more advanced techniques (e.g., specification and development of a driver to be used as a "faultload generator" and application of other fault injection techniques, for example, mutation of the driver code).

Besides we will also consider faultloads consisting of bit-flips in the memory space of the kernel using the well-established SWIFI technique (e.g., see [Carreira 1998, Arlat 2002]), in the sequel, we focus on the presentation of benchmark prototypes featuring API-induced faultloads. As mentioned in the previous paragraph, further experiments targeting diver-related faults are also being conducted. We anticipate that such a faultload aimed at emulating driver-induced erroneous behaviours will be part of the set of benchmark prototypes targeting OSs that will be proposed by DBench.

At this stage, our intent is to propose and investigate benchmark prototypes that are aimed at providing means for the standard assessment of generic OSs (i.e., benchmarks that can be of suitable for external use purpose). Indeed, the proposed benchmark prototypes and the related experiments that will be carried out during WP3 will allow us to better appraise where we actually stand on that objective.

3.1 Measures and Measurements

Measures of interest encompass:

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

3.1.1 OS-level measurements

Among the first category, one can find some outcomes that have been put forward to classically characterize OS robustness:

- Reported failures: returned error code, exception raised,
- Non (explicitly) reported failures: OS hang or OS crash.

Besides they characterize the behaviour of the OS in presence of faults (independently of the service being delivered at the application level), these outcomes are usually diagnosed at the application layer by means of the observation capabilities of the *benchmarking management system* (e.g., see [ETIE1 2002]).

3.1.2 Application-level measurements

It is also possible to extend the assessment by considering higher-level failure by explicitly assessing some application-level services within the benchmarking management system. A reference behaviour (notion of oracle) is evaluated when the nominal workload is applied

alone (no faultload). The idea is then to compare the behaviours observed at the application level when the prescribed faultload is superimposed to the workload.

Although timing measurements — leading to useful analysis of the impact of faults on the executed workload — could be easily derived from performance benchmarks, (we are in particular interested in those focusing on OS services), on the contrary, discrepancies of the results and/or integrity checks with respect to a reference run (value and/or integrity failures) are far more difficult to diagnose in the case when performance benchmarks are essentially available as executable code packages (source code is not always made available).

Accordingly, we plan to use specific reference features (e.g., synthetic/generic type of service) in association with or in addition to the workload applied to activate the OS(s) being benchmarked.

It is worth pointing out that the ultimate form of such an application-level assessment is experienced in the case of the benchmarks targeting specific application domains that are described in the subsequent paragraphs.

3.1.3 Restoration time measurements

One specific class of timing measurements is very specific of a dependability benchmark; it corresponds to time taken by the OS to be able to provide its service after a failure has occurred (the so-called *reboot* and *restart* times).

Indeed, this time might be significantly impacted by: i) the type of failure experienced (two specific cases are OS crash and OS hang), and ii) the extent of the corruption of the internal state of the OS.

3.1.4 Additional OS-specific timing measurements

Besides it orchestrates the activity and scheduling of application processes, an OS is also a multithreaded entity that manages internal events by means of a complex state machine-like logical architecture. Accordingly, OS specific timing features that govern the transitions between internal states could be of interest. As examples of such timing characteristics one can think of i) the time for moving tasks between delay queues and ii) the ready queue of the kernel and (usually called “context switch time”) and of time to execute the kernel calls (“system call overhead”). See [Rodríguez 2002] for a comprehensive list of relevant real-time parameters .

Nevertheless, it is worth noting that such measurements require a very detailed analysis and knowledge of the architecture of the OS kernel. Besides such measurements might sometimes be difficult to obtain and might not be always useful in the case benchmarks concerning full-sized generic OSs, as the ones that we are considering here — except for system integrators; still, such analyses might be of interest for embedded systems (as shown in [Rodríguez 2002]). Nevertheless, as is also considered by performance benchmarks, “empty” kernel calls can be used to allow for the overall time spent for a kernel call to be readily measured (e.g., see [ETIE1 2002]).

3.1.5 Error propagation channels

As a specific measure, we consider the risk of propagation (via the OS) of errors between application processes that have a priori no functional and explicit communications, but that share the resources provided by the OS. Such an analysis is especially interesting when application processes with different levels of integrity share common resources (in particular, the data structures of the OS). What is feared is that the effects of errors in a low-integrity process impact a higher integrity process through the OS.

Such an error confinement capability can be considered as characterizing a specific form of robustness of the OS.

In this case, the data structure of the OS forms the set of common resources that may materialize such an error channel. Indeed, such a threat can be considered as weakness of the OS. It is important to note that such impacts encompass not only the propagation of erroneous data, but also denial of service that may result for example in the high-integrity process to hang.

Figure 3.1 sketches how the type of analysis that was briefly described can be carried out. As shown, the workload features two processes (a “low-integrity” process and a “high-integrity” process). Faults are injected on the parameters of the kernel calls issued by the LI process; more details on the faultload being considered are provided later on. The HI process features an oracle for diagnosis of application-level affecting the service it delivers (in addition to the hang failure mode of the HI process).

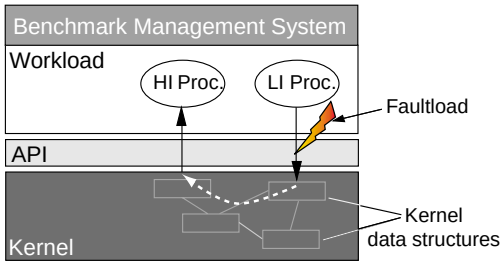

Figure 3.1 - Analysis of error propagation through the OS kernel

3.2. Workload

Concerning the workload to be used for activating the OS being benchmarked, we will of course consider the related performance benchmarks that are available (e.g., see [ETIE1 2002]). Accordingly, the basic type of workload that will be included concerns synthetic workload targeting specific OS services/functions (e.g., scheduling, memory management, synchronisation, etc.), as was used in the OS-related fault representativeness studies (see [ETIE2 2002], section 4). The main interest of such an approach is that it allows for a series of modular benchmarks to be developed (each focusing on a specific OS function). The corresponding results obtained for each such workload can then be analysed by the user either independently or on a weighted basis, according to the expected activation of the system in which the OS is to be integrated.

However, if the application area is known, the workload could include the application software itself, as is the case in the benchmarks investigated concerning embedded systems (see Section 3.3). The advantage is that the scope of the benchmark is more realistic to study features of interest to this application area. Of course application-dependent failure modes can be analysed.

As an alternative, the workload used to activate the OS(s) being benchmarked can be derived from the application requirement specifications as an application area trace. Such a trace provides in a concise way realistic activity that is representative of the application area. Nevertheless, as the trace only provides inputs to the kernel it is not suitable to be used as an oracle for the diagnosis of application-specific failure modes. This can be handled — to some extent by the provision of a specific set of oracle processes. Figure 3.2 illustrates this idea.

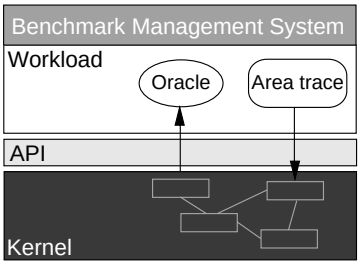

Figure 3.2 - Using an application area trace as part of the workload

3.3. Faultload

Based on the results obtained in WP2, and as mentioned earlier, we will first concentrate on API-level faultload. Such a faultload is mainly geared towards the assessment of the OS being benchmarked with respect to errors induced by faulty applications. As this is now usual, we will also include in the faultload considered in the prototype benchmarks under investigation the case of bit-flips in the kernel memory space, which are more related to hardware induced errors and to some extent representative of the effects caused by residual software faults of the kernel as was also confirmed by the experiments that were conducted to investigate the representativeness of this technique with respect to software faults (see [ETIE2 2002], section 5)[1].

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

The faults that will be injected will be transient in nature (the corruption will be applied only once when the targeted call will be executed). Indeed, such fault injection tends to provoke more subtle erroneous behaviours. We plan also to consider random selection among the kernel calls, the parameters and the bits for applying bit-flips. Single and multiple bit-flips will be investigated.

[1] We recall that we also intent to investigate the application of a specific faultload aimed at emulating OS drivers faults at a later stage of the development of WP3.

During the experiments, we plan to monitor the proportion of "no signalling" outcomes, as this might be considered as some form of "non significant test" and thus impair the efficiency of the benchmark.

We intend to discriminate the results according to the fault types being injected (invalid *vs.* erroneous parameters), as this will be allowed by the strategy we will consider to conduct the experiments aimed at evaluating the proposed prototype benchmarks.

3.4. Benchmark Conduct

For conducting the experiments aimed at evaluating the prototype benchmarks for generic OSs that we are putting forward, two main strategies can be identified:

1) All benchmarking experiments/runs are carried out independently and the OS is reset after each run. Each experiment features the application of the workload and of a basic element of the faultload: a single fault is applied (e.g., see [Arlat 1990]).
2) A full combination of workload and faultload is applied to the OS for a fixed time interval or for a prescribed number of elementary faults. OS reboot/restart are only initiated by the target system and are explicitly part of the behaviour that is benchmarked (e.g., see [Tsai 1996]).

Hereafter we briefly identify the main differences between these two strategies[2]. We plan to further investigate these during the experiments that will be carried out in WP3.

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

The second strategy is basically aimed at obtaining a specific experimental measure that encompasses various facets of the behaviour of the OS in presence of faults into a *single figure* (along the same line as what was proposed for benchmarking transactional systems in [ETIE1 2002]. It is worth pointing out that this strategy actually pays-off if the OS exhibits sufficient fault tolerance features (i.e., when it is likely that it will its execution unaffected in presence of faults); indeed, if this is not the case then this strategy is not very much different from the first one. Such a strategy provides a practical means to obtain a measurement of the impact of the faultload on time taken by the OS to restore its state (and of the running processes at the

[2] These differences can be related to the differences that exist between independent Bernouilli trials (e.g., see [Arlat 1990]) and experiments based on Monte Carlo simulations, that rely on a specific distribution of the faultload (e.g., see [Sieh 1997]).

application layer, if any). As an example of a pragmatic measure in this category one can consider the number of successive faults being applied before the OS has crashed. Here as well, the synchronization of the faultload with the workload and the ordering of the faults being applied are major characteristics of the benchmark, and as such, should be explicitly revealed and by all means maintained unchanged for reproducibility. However, it is worth noting that the results obtained this way are very much dependent of the way the workload and faultload are applied and combined together and especially how they fit the application area of interest for the end-user. Indeed, in this case, the selective analysis of the impact of each of the faults applied is much more difficult and in many cases impossible. Accordingly, such a strategy is much less attractive from an integrator point view.

From a practical point of view, it is worth noting that the first strategy that has been widely used in the case of the experimental evaluation of fault-tolerant systems, can be extended to include the "restoration times" that separate two consecutive runs in the measurements to be considered as part of the benchmark.

These experiments will be carried out on two OSs: Linux and Windows 2000. In order to have a common reference for comparison, we will take advantage of the fact that both OSs feature the POSIX API. It is also assumed that a host computer is available (as part of the benchmarking management system) to control the launch of the benchmark and control its execution (e.g., for assessing the severe failures of the OS (crashes) and resume the operation in that case).

This experimental and analysis step will be useful for studying the issues attached to how an OS dependability benchmark proposal can be implemented on two very distinct OSs.

Finally, we will consider that the primary performer of the benchmark is not the developer of the OS being benchmarked. So, we will assume limited knowledge about the internal structures of the OS and in particular that the performer may not have access to the source code[3]. As the performer is not always the user of the benchmark results, it is necessary that the benchmark procedures be very precisely specified and the results provided easily and correctly interpretable (notion "external use purpose" — e.g., see [ETIE1 2002, ETIE2 2002]).

4 Dependability benchmarks for transactional applications

The goal of this section is to present the dependability benchmark prototypes for transactional applications. This dependability benchmark is particularly targeted to OLTP (On-Line Transaction Processing) systems, which constitute the kernel of the information systems used today to support the daily operations of most of the business. Some examples include banking, e-commerce, insurance companies, all sort of travelling businesses, telecommunications, wholesale retail, complex manufacturing processes, patient registration and management in large hospitals, libraries, etc. These application areas comprise the traditional examples of business-

[3] This is not the actually the case for the two OSs that we will be targeting, as we have access to the source code of both Linux (of course) and Windows 2000. This will be of great help in tuning the experiments for evaluating the proposed benchmark prototypes.

critical applications, which clearly justify the interest of choosing them as target for the first dependability benchmark for transactional applications.

Two prototypes are being built. The prototypes have in common the fact that both of them include the key elements of a typical OLTP system. For instance, they will have one or more machines working as a database server, application server machines (used for instance to run a web server or a transaction load balance application), a network, and the clients (simulated by one or more machines). Furthermore, both prototypes use the same base workload (TPC-C workload, as explained below) and share common database management systems (DBMS), namely the commercial Oracle DBMS and the open source product MySQL. Both Windows and Linux operating systems are being used.

The most significant difference between the prototypes is in the fact that while one prototype is a realistic implementation of an OLTP system, the second runs over the UMLinux framework [Buchacker 2001, Sieh 2002], which is a Linux simulator. Additionally, while the first prototype is primarily a classic OLTP system (client-server or three tier system) the second one explicitly assumes a web interface for the end-user clients. Together, both prototypes will provide us with means to perform a wide range of benchmarking experiments and to extend experiments already performed in WP2. The fact that the prototypes are being implemented in two different sites will facilitate the evaluation of portability aspects, which include the implementation of the same benchmark in different systems by different teams. Additionally, it also facilitates the cross-exploitation of results between these two dependability benchmark prototypes.

The presentation of the benchmark prototypes in this section follows the guidelines for external benchmarking (see section 2), as we are targeting real external dependability benchmarks for OLTP systems. Subsection 4.6 discusses how the definition of the prototypes can be extended for internal use.

4.1. Benchmark setup

The key elements of the experimental environment required to run the dependability benchmarks are represented in Figure 4.1.

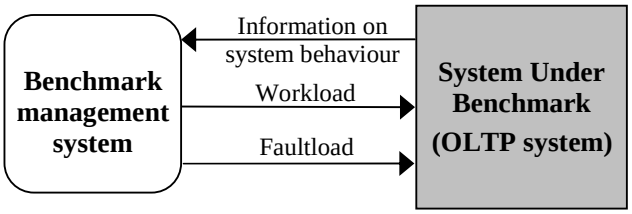

Figure 4.1 – Key elements of the benchmark setup.

A typical OLTP environment consists of a number of users managing their transactions via a terminal or a desktop computer connected to a database server via a local area network or through the Web. Thus, an OLTP system is typically a traditional client-server system or a multi-tier system. Both classical client application and web-based clients are considered.

The goal of the benchmark management system is to control all the aspects concerning the benchmark run, during which it submits the transactions that compose the workload (emulating a set of external users), injects the faultload, and collects information on system behaviour. Measures are afterwards deduced by analysing the information collected during the benchmark run.

The SUB represents a client-server system (either a client-server or a multi-tier system) fully configured to run the workload. A large variety of architectures can be used ranging from centralised systems to different configuration of parallel and distributed systems. From the benchmark point of view, the SUB is the set of processing units used to run the OLTP application adopted by the dependability benchmark as a transactional workload and to store all the data processed by the transactions. That is, given the huge variety of systems and configurations used in practice to run OLTP applications, the definition of the SUB is tied to the transactional workload instead of being defined in a structural way. In other words, the SUB can be any system (hardware + software) able to run the workload under the conditions specified by the dependability benchmark. Additionally, all the details on the SUB internals needed to obtain and understand the benchmark measures must be provided along with the benchmark results.

Most of the transactional systems available today use a database management system (DBMS) as transactional engine. For that reason, the DBMS is the most important component in the SUB. However, the SUB includes all the elements required to execute the workload and not just the DBMS.

4.2. Workload

The notion of transaction used in OLTP systems corresponds to the usual business meaning of the word “transaction”. In fact, a transaction is the set of operations in a computer system required to support **business transactions** such as common commercial exchanges of goods, services, or money. A typical computer transaction includes normally reading from, writing to, or updating a database system for things such as inventory control (goods), airline reservations (services), or banking (money). At the database level, a **database transaction** is a set of database operations that comply with the well-known ACID properties: atomicity, consistency, isolation, and durability. A business transaction is comprised of one or more database transactions.

Considering the application area addressed, our first choice is to adopt the workload of the well-established TPC Benchmark™ C (TPC-C). This workload represents a typical database installation. The business represented by TPC-C is a wholesale supplier having a number of warehouses and their associated sale districts, and where the users submit transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. This workload includes a mixture of read-only and update-intensive transactions that simulate the activities of most OLTP application environments, including transactions resulting from human operators working on interactive sessions.

The TPC-C workload includes an external driver system, which emulates all the terminals and their emulated users during the benchmark run. In our case the external driver system specified in the TPC-C benchmark [TPC 2001] will be included in the benchmark management system.

It is worth noting that some research is required to assess if the TPC-C workload fully satisfies the specific needs of a dependability benchmark or if some changes are needed. For the time being and for the preliminary experiments already performed [Vieira 2002] we are using the TPC-C workload without any changes. In addition to possible changes in the TPC-C workload (to fulfil specific dependability benchmarking needs that may be identified), we also consider to perform experiments with a simplified prototype with web services only, using the SPECweb99 [SPEC 2000] as workload.

4.3. Faultload

The faultload represents a set of faults and stressful conditions that emulates real faults experienced by OLTP systems in the field. A faultload can be based on three major types of faults: operator faults, software faults, and hardware faults. Although some of the published studies on the analysis of computer failures in the field are not directly focused on transactional systems, available studies clearly point operator faults and software faults as important causes for computer failures [Gray 1990, Sullivan 1991, Lee 1995, Kalyanakrishnam 1999, Sunbelt 1999]. This is particularly true for very complex systems such as typical OLTP systems. For this reason, we will give particular attention to these two classes of faults: operation faults and software faults. Additionally, hardware faults will be also considered, as they are consistently reported as a (still) significant cause of failure in transactional systems, especially in what concerns to faults in devices such as disks (in spite of the popularity of RAID systems), network interface devices, power sources, etc.

Operator faults in database systems are database administrator mistakes. The great complexity of database administration tasks and the need of tuning and administration in a daily basis, clearly explains why operator faults (i.e., wrong database administrator actions) are prevalent in database systems. Several interviews with database administrators of real databases installations conducted on behalf of our research work also confirmed the prevalence of operator faults in database systems.

The complete elimination of software defects during software development process is very difficult to attain in practice. In addition to well-known technical difficulties of the software development and testing process [Musa 1999, Lyu 1996], practical constraints such as the intense pressure to shrink time-to-market and cost of software contribute to the difficulties in assuring 100% defect free software. Therefore, the actual scenario in the computer industry in general (and, consequently, in the transactional systems world) is having systems in which software defects do exist but no one knows exactly where they are, when they will reveal themselves, and, above all, the possible consequences of the activation of the software faults. As components of the shelf (COTS) are being used more and more to build large OLTP systems, the residual software faults represent a growing risk and should be addressed in our benchmark prototype.

Traditional hardware faults, such as bit flips, will not be considered in a first stage has they are not generally considered an important class of faults in OLTP systems (although the ultra-high densities of new silicon technologies are bringing back transient faults inside chips as an important class of faults). We will therefore concentrate on high-level hardware failures such as hard disk failures, network interface failures, or failures of the interconnection network itself.

We will also consider the possibility of emulating certain high-level hardware failures through operator scripts and specific programs.

4.3.1. Operator faults in DBMS

Most of today's transactional systems use a database management system (DBMS) as transactional engine. For that reason, transactional systems are strongly influenced by database technology, which turns the problem of operator faults in transactional systems essentially into a problem of operator faults in DBMS.

Operator faults in database systems are database administrator mistakes. End-user errors are not considered, as the end-user actions do not affect directly the dependability of DBMS. Database administrators manage all aspects of DBMS. In spite of constant efforts to introduce self-maintaining and self-administering features in DBMS, database administration still is a job heavily based on human operators (and this picture is not likely to change in the near future).

The injection of operator faults in a DBMS can be easily achieved by reproducing common database administrator mistakes. That is, operator faults can be injected in the system by using exactly the same means used in the field by the real database administrator (i.e., we do not emulate faults as it is usual in traditional fault injection: we really reproduce operator faults). In order to make the procedure fully automatic, faults can be injected by a set of scripts that perform the wrong operation at a given moment (the fault trigger). As usually happens in traditional fault injection, the fault trigger can be defined in such a way that operator faults can be uniformly distributed over time or can be synchronized with a specific event or command of the workload.

Different DBMS include different sets of administration tasks and consequently have different sets of possible operator faults. However, as shown in [ETIE2 2002], it is possible to establish equivalence among many operator faults in different DBMS. In other words, a faultload based on operator faults is fairly portable across typical OLTP systems (see [ETIE2 2002] for a detailed discussion on operator faults portability in three leading DBMS, respectively Oracle 8i, Sybase Adaptive Server 12.5, and Informix Dynamic Server 9.3).

We propose the following steps to define a faultload based on DBMS operator faults:

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 for each fault type according to the number of times the correspondent administration task is executed. A 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 DBA introduces inadvertently a security fault the system continues to work normally until another person maliciously take advantage from that to break into the system (this is a second event). Fault types that need a second event to show are not interesting to be part of a faultload.

In a distributed environment where the SUB is composed by several machines the faults should be distributed uniformly through all the machines.

An example of preliminary dependability benchmarking using operator faults defined in the way proposed in this section can be found in [Vieira 2002].

4.3.2. Software faults

The technique proposed for the injection of software faults is called Generic Software Fault Injection Technique (G-SWFIT) [Durães 2002] and consists of modifying the ready-to-run binary code of software modules by introducing specific changes that correspond to the code that would have been generated by the compiler if the software fault were in the high-level source code. A library of mutations previously defined for the target platform guides the injection of code changes: the target application code is scanned for specific low-level instruction patterns and selective mutations are performed on those patterns to emulate related high-level faults. Depending on the size of the low-level mutation library and the settings specified by the user, a number of versions of the original target application are created, each one containing an emulated high-level software fault. Although not strictly necessary, a commercial disassembler can be used to produce assembly listing to assist guidance of the fault injection process.

A crucial aspect of the proposed technique (concerning its use for benchmark faultloads) is in the fact that it works at the machine-code level and it does not required the availability of the source code of the target program. This feature makes it possible to apply the proposed technique to virtually any program, which is particularly relevant for dependability benchmarking of COTS components or COTS based systems.

The accuracy of the injected faults and the generalization of the proposed method (concerning high-level languages, compilers, compilers optimization options, processor architecture) has been studied in [ETIE2 2002] and the available results suggest that the technique is very accurate and can be easily ported to practically all type of systems (we mainly need to define specific parts of the fault library to accommodate the emulation of faults in a new kind of system).

Although a detailed discussion of the G-SWFIT representativeness and portability is presented in [ETIE2 2002], it is worth noting that the proposed approach introduced the concept of **educated mutations** to improve the representativeness of the injected faults as much as possible. The idea

is to define a subset of selective faults based on available field data on bug reports and common programming language pitfalls. In this way, injected faults are not arbitrary mutations of the code but they do correspond to the most common programming mistakes.

The software faults injected are organized in general classes according to the Orthogonal Defect Classification (ODC) [Chillarege 1995]. This classification 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 software faults have been corrected). As shown in [ETIE2 2002], faults in a given ODC class are further detailed in subtypes, which is useful to study the representativeness of injected faults and the equivalence of the same classes of faults injected in different binary codes.

The hypothesis under research to define faultloads **for external** dependability benchmarks based on the G-SWFIT is that faults of same ODC classes are statistically equivalent for large enough sets of faults. For **internal** dependability benchmarks the problem of assuring equivalence of faults in the same classes when injected in different binary codes is not very important, as the primarily goal of internal benchmarks is not system or component comparison.

One fundamental aspect of the proposed methodology is the clear separation between the software component subject to the injection of faults and the rest of the SUB. That is, software faults are always injected in a given component to evaluate the impact on the rest of the SUB (not in that component). This means that when we benchmark different configurations of the SUB in order to isolate the effects of a given component on the benchmark measures, the isolated component (i.e., the one that we want to characterize) cannot be the component in which faults are injected. This way we avoid the problem of changing the component/system we want to characterize, as the emulation of software faults is based on the introduction of small changes in the target component.

A last aspect concerning the use of software faults in benchmark faultloads is related to the tools needed to inject the faults. Two possibilities are under research: 1) the use of a specific mutation tool to inject faults and 2) the use of the workload (a specific part of the workload) to perform the injection of the faults (i.e., to insert the mutations in the right components).

4.3.3. Hardware faults

We will consider high-level hardware faults, which are characterized by their location and the time they remain in the system. We distinguish persistent faults, i.e. those that remain in the system indefinitely, from faults which disappear after a certain amount of time. We call the former permanent faults, the latter duration faults. We will investigate injecting only single faults and having faults accumulate during longer test runs.

When injecting hardware faults, we will specifically target vulnerable locations of the system, such as the hardware components where the data is kept, those necessary for transmitting the data or requests from server to client and vice versa, or even an entire machine.

The data is kept on storage devices. For fast access in large servers, arrays of hard-disks, often in a RAID configuration, are used. We will inject different kinds of hard-disk faults. We can make a single hard-disk completely inaccessible or make certain blocks on the hard-disk faulty.

Depending on when and how the data is accessed, the injected fault will affect the server at once, at some later point or even not at all. The client monitors the reaction of the server.

In distributed client-server systems, the interconnection network is an integral part. Therefore, we will specifically target it with hardware faults. We will inject faults into the machines network interfaces as well as into the interconnection network itself. Faults injected into a machine network interface include inability to send or receive network packets or sending/receiving corrupted network packets. Faults injected into the interconnection network include broken cables and unavailable routes. The server behaviour as viewed from the client's perspective is the same as for hard disk failures.

An entire machine or even a number of machines connected to the same power supply can become unavailable on a power failure. We will therefore also investigate the reaction of the system to power failures of server machines.

To perform the hardware fault injection experiments, we will be using UMLinux, which provides an environment to set up the system under benchmarking on virtual machines. The hardware faults will be injected into the (virtual) hardware of those machines. Additionally, the emulation of high-level hardware faults by operator scripts and programs (similar to the ones used to simulate operator faults) is under investigation and will be used to define faultloads based on high-level hardware faults for real OLTP systems (i.e., not in a simulation environment).

4.4. Measures

The measures will be obtained through a set of experiments. The SUB will run the workload and the experiments include three main phases. In a first set of experiments the workload is run without (artificial) faults and in a second set of experiments the same workload is run with a faultload. Finally, a set of consistency tests (in addition to the normal consistency texts included in the workload) can be performed on the data processed by the transactions to detect possible integrity violations in the final data resulting from the transactions.

The measures must be taken in the different experiment phases shown in Figure 4.2.

- **Phase 1** - First run or set of runs of the workload with no faultload. This run will be used to collect the baseline performance measures. These performance measures represent the performance of the system with normal optimisation settings. The idea is to use them in conjunction to the other measures (see below) to characterise system dependability (and not as a measure of system performance).

- **Phase 2** – In this phase the workload is run in the presence of faults with the goal of measuring the impact of faults on the transaction execution. Two different approaches will be investigated.

  In the first approach the idea is to run the workload during an extended period of time (e.g., 8 hours or more) and inject the faults specified in the faultload. After injecting one fault the SUB may continue executing transactions normally if nothing wrong is detected. Otherwise, the SUB may start a recovery procedure, depending on the fault effects and the detection and diagnosis mechanisms in the system, or may need to be restarted

completely if recovery is not enough. The driver system will restart the database system after a fault whenever the establishment of client sessions with the target database fail. The complete restart of the target systems (i.e., operating system + database) will be performed whenever the benchmark management system detects a target system crash. One important aspect is that the explicit restart of the SUB by the benchmark management system is only done when the SUB crash or close the connection with the benchmark management system. In this way, the effects of more than one fault may accumulate in the system.

The second approach that will be explored in the prototypes consists of decomposing phase 2 in several independent injection slots. In this case, the SUB will be explicitly restarted in the beginning of each injection slots and the effects of faults do not accumulate in the system. This decomposition of phase 2 in injection slots is extremely important if the faultload injected is based on software faults, as the injection of a software fault changes the software module where the fault is injected (this may be also he case for some hardware faults). For faultloads based on operator faults this does not happen. In this second approach, the benchmark management system will have to keep the notion of accumulated time in the experiment in order to make it possible measures such as number of transactions per unit of time. This approach was followed in preliminary experiments [Vieira 2002] and our goal is to compare the feasibility of both approaches.

An important goal is to assure that phase 2 is completely automatic and can be performed without user intervention.

- **Phase 3** – This last phase consists of running a set of application consistency tests (specified by business rules in the TPC-C specification) to check possible data integrity violations. These integrity checks will be performed on the application data (i.e., the data in the database tables after running the workload) and will use both business rules (defined in the TPC-C specification) and the database metadata to assure a comprehensive test. Possible integrity violations detected in this phase represent application data errors from the end-user point of view. It is worth noting that these tests are performed in addition to the normal integrity test included in the workload and in the database engine that may also detect integrity errors during phase 2.

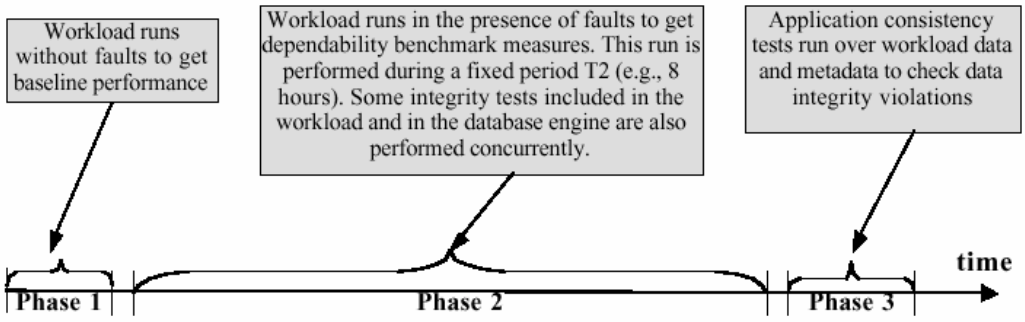

Figure 4.2. - Dependability benchmark for transactional applications

The dependability benchmark measures to be collected during the three phases are the following:

- **Tb** - Baseline transactions per minute. This measure corresponds to the classical TPC-C performance measure known as tpmC. However, Tb represents a baseline performance instead of an optimised performance measure (as is the case of tpmC) and should consider a good compromise between performance and recoverability.

- **€/Tb** - Price per transaction. The price is established using a set of pricing rules specified in the TPC-C specification.

- **Tf** - Number of transactions executed in the presence of the faults specified in the faultload during the time T2 (see Figure 4.2). It measures the impact of faults in the performance and favours systems with higher capability of tolerating faults, fast recovery time, etc.

- **€/Tf** - Price per transaction in the presence of faults of the faults specified in the faultload during the time T2. It measures the (relative) benefit of including fault handling mechanisms in the target systems.

- **Avt** - Availability in the presence of the faultload during the time T2. It measures of system availability during the period T2 from the end-user point of view. This measure corresponds to Avt = Tav/T2, where Tav is the total time the system is available from the point of view of the end-user and T2 is the testing period of phase 2 (if phase 2 is composed of a set of injection slots this time will be the sum of the transaction processing time in all the slots). Several classes of availability can be considered as a specialisation of this measure. For example Avt can represent the case when all terminals are unavailable; Avt50 the availability when less than 50% of the terminals are unavailable; Avt10 the availability when less than 10% of the terminals are unavailable, etc.

It is worth noting that the measures proposed for this preliminary dependability benchmark follow the well-established measuring philosophy used in the performance benchmark world. In fact, the measures provided by existing performance benchmarks give relative measures of performance (i.e., measures related to the conditions disclosed in the benchmark report) that can be used for system/component improvement and tuning and for system comparison. It is well known that performance benchmark results do not represent an absolute measure of performance and cannot be used for planning capacity or to predict the actual performance of the system in field. In a similar way, the measures proposed for this first dependability benchmark must be understood as benchmark results that can be useful to characterise system dependability in a relative fashion or to improve/tune the system dependability.

The proposed set of measures has the following characteristics/goals:

- Focus on the end-user point of view (real end-user and database administrators).
- Focus on measures that can be derived directly from experimentation (i.e., following the general tradition of the performance benchmark world that relies on direct measures).
- Include both dependability and performance measures.
- Measures must be easily understandable (in both dependability and performance aspects) by database users and database administrators.

All the performance and dependability measures are collected from the point of view of the emulated users. In other words, the measures correspond to an end-to-end characterisation of performance and dependability from the end-user point of view.

4.5. Procedures and rules

The TPC-C benchmark, used as starting point for the prototypes of dependability benchmarks for OLTP, includes a detailed set of procedures and rules that must be observed when performing benchmarking [TPC-C 2001]. Although some of these procedures and rules are very related to the specific benchmarking philosophy of the Transactional Performance Processing Council, some other rules and procedures do apply to our case. Additionally, new procedures related to dependability aspects of benchmarking will have to be introduced. One important goal of the experimental research planned for the benchmark prototypes is just the refinement of the set of procedures and rules that have to be included in a real dependability benchmark specification.

4.6. Benchmarks for internal use

As mentioned before, the major differences between a benchmark for external use and a benchmark for internal result resulted from the fact that system/component comparison is not the primarily goal for the internal use of dependability benchmarks. This dramatically increases the degree of freedom in changing and adjusting the benchmark elements defined in the benchmark specification. The major new aspects in the use of the OLTP benchmark prototype for internal use are:

- **Measures** – We will have more measures, including new measures related to specific dependability features of the system under benchmark.
- **Workload** – The workload can be changed according to the benchmarking goals. For example, new modules can be added to the workload (e.g., using synthetic programs) to exercise in a more intensive way a given set of functionalities. In our case, the transaction profile can be changed to include new types of transactions that might be useful for the evaluation purposes.
- **Faultload** – The two major constraints related to the faultload, which are **representativeness** and **portability**, can be relaxed for internal benchmarks and the faultload can be adapted (e.g., by including new types of faults) to validate specific dependability features.

The extended set of measures is probably the most evident aspect the use of the proposed dependability benchmark for internal purposes. In addition to the measures presented above, in a benchmark for internal use it is also possible to get measures on the behaviour of specific system components in the presence of faults. For example, the injection of a fault may activate the error detection mechanisms in the system under benchmark (e.g., server error detection mechanisms). In practice, errors can be detected at the following levels:

- HW + OS level
- Database engine level
- Application level (e.g., workload consistency tests)

Specific measures such as error detection efficiency and error detection latency (global or at a given level) can be easily included in an internal benchmark for transactional systems.

It's worth noting that the error detection example mentioned above is in fact just an example. A given system component can be characterized according to several specific dependability measures. For example, specific dependability measures for the database engine are (see [ETIE1 2002] for a complete description of specific measures):

- Error detection efficiency
- Error detection latency
- Data integrity violations
- Data recovery efficiency
- Average recovery time
- Transaction properties violations due to faults.

Summarizing, the dependability benchmark prototype for OLTP will be, first of all, case studies for the development of future external benchmarks. Additionally, we will explore the prototypes to investigate the internal use of a dependability benchmark in the dependability characterization of computer systems and components.

5. Dependability benchmarks for embedded applications

5.1 Example of Embedded Control System for Space

The aim of this benchmark prototype is to compare the dependability across OS's with a small control and navigation application running on top of it. The workload uses OS services provided through the POSIX pthread interface [POSIX]. The basic building blocks of the benchmark are depicted in Figure 5.1.

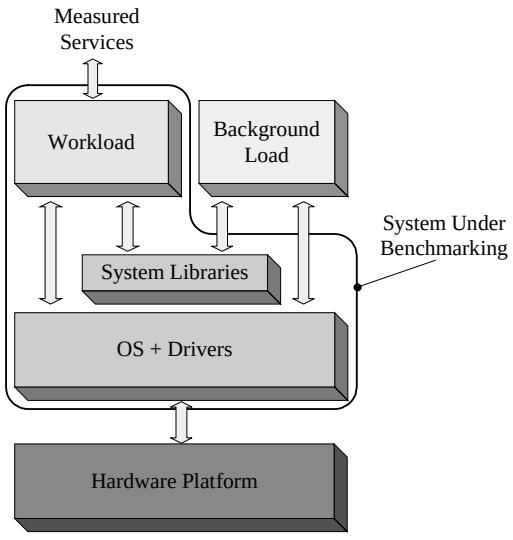

Figure 5.1 - Conceptual view of the benchmark platform. Running on the OS as a part of the benchmark is both a workload and a background load.

5.1.1 Dimensions

The benchmarking platform consists of a PC platform, the OS and an application on top as shown in Figure 5.1. The system under benchmarking is the OS together with the application. This choice was made because the measures chosen apply to the complete system, i.e., the measures are based on the services provided by the application (see the following section).

It is important to note that even though the hardware is not included in the system under benchmarking it very much influences the result of every benchmark experiment. As the intent is to compare OS's the impact of the hardware should be reduced. This is achieved by keeping the hardware the same across benchmarking experiments using different OS's[4]. On the other hand, the impact of the application is desired, since this contributes to making the benchmark representative. In our case the application is a control system which makes use of the underlying support of the OS in a special manner giving rise to a particular profile. This profile is similar to that of other control systems which make it representative in that area.

In order to reduce the complex problem of representativeness, one solution is to divide applications into different areas. Our application is indeed a space application, which makes it attractive to categorize it as a space-application. This is however not useful, instead a categorization based on functionality is preferred. There are many connotations to space-applications that are not easily implemented in a benchmark, such as special-purpose hardware and environment simulators etc. We have therefore focused on the more general area of control system, which our space application is an example of. We do acknowledge that our application alone does no represent the entire area of control systems and therefore a real benchmark should include several control applications of different nature. By including several applications the impact of implementation details and characteristics is reduced.

A benchmark can be used in different phases of a products life-cycle. We have focused the benchmark at the design/specification phase, where a suitable OS is to be selected from a set of candidates. The benchmark is used to aid in the decision process. It is of course important that the benchmark represent the actual application area of the product, this is what representativeness is about. Here it might be helpful if a similar product exists, as an earlier model/prototype of the product. This model can then be used in the benchmark. The scope of such a benchmark would be for internal use, only within one company/organization. The contrast is benchmarks for external use, which have results publicly available for all (or for those who are willing to pay). The focus has been on a benchmark for mostly external use, but the concepts should be possible to apply for internal use as well.

The intended user of the benchmark is a system integrator. He/she wants to select a suitable OS to be used in a system. The benchmark might be of some use to other user categories as well, such as designers of OS's. However, an OS vendor has much more detailed knowledge of the OS, such as access and capability to change the source code, which can be of use when tuning the system. Therefore the usability of our benchmark for OS designers is limited. The actual

[4] The results using different hardware platform potentially gives different results, even on a relative basis, since OS's (and applications) can be optimized for certain platforms. Therefore, the specification of the benchmark platform must be included in the benchmark result.

performer of the benchmark might be the integrators themselves or third-parties acting on behalf of the integrators.

For a benchmark to be useful it must fulfil some basic properties. These properties include representativeness, portability, non-interference with the system under test, short execution time and accuracy. All of these play a vital role in the selection of workload and faultload.

5.1.2 Measures for Dependability Benchmarking

The measures used in this benchmark are defined by the services provided by the application. This means that it is the behaviour seen by the users of the application that forms the basis for the benchmark measurements. We have defined four high-level abstract measures that we will consider in the benchmark. These are all generic and common to control systems as well as other systems. Most of them are performance-related, i.e., they capture how the performance of the services provided is affected by the presence of a stressful environment. For each of these high-level measures, appropriate lower-level measures are defined based on the application used as workload.

The measures we are considering at this point are:

Throughput: Maximum number of external events[5] that can be serviced per second

Response time: Response time of events coming aperiodically measured in seconds

Overload capacity: Number of other concurrent tasks and their load running in parallel until the tasks miss their arrival time

Jitter: The jitter in response time values, measured as the variance of the response time and the jitter in arrival times for the periodic tasks, measured as a variance of the arrival time

For each measure, there must be a method for obtaining it. The effect stresses can have range from serious system failure, such as an OS crash necessitating a reboot, to less severe failures causing performance degradation for the services provided. The different effects, or failure modes, can be categorized according to a scale. Many different scales have been used for similar experiments, for instance the CRASH scale introduced in Ballista [Koopman 1999] or the categorization used in MAFALDA [Fabre 1999]. Since our measures are based on the services provided by the application some failure modes can possibly be bundled together, e.g., whether the OS hangs or crashes does not really matter as the services have failed in both cases (although the issue of reboot and recovery may be different), also the difference between application hang and crash is small.

The list of considered failure modes consists of:

- OS crash/hang
- Application crash/hang
- Incorrect service delivery (in value and/or time domain)

[5] The term event means in this case simulated interrupts from the environment simulator. These events simulate interrupts occurring in the real system.

- Performance degradation of service

Also, the services can be unaffected by the stresses introduced, which for many cases is the expected and desirable response. Note that the list presented here is valid for external purposes. For internal use, a more fine-grained classification may be desirable (for instance to separate OS crash and hang). Some of the failure modes have associated values and some have not. For OS crash/hang there is no value associated, the same goes for application hang or crash. Performance degradation can be measured in percent of time increase and for incorrect service delivery data such as the fault response or the failure in time is recorded.

One problem, which is not taken into account at this stage, is how the results should be interpreted. As the result is a string of values (failure modes together with values from the measures above), where some results are times and some are capacity measures, the interpretation is not straightforward.

Most of the proposed measures are in the time domain. We will therefore device methods for measuring these times as accurate as possible. The basic approach is to first measure baseline performance and then measure the performance in presence of stresses. This approach is similar to that taken for transactional systems. One aspect that at this stage cannot be tackled is whether two or more of the measures can be assessed at the same time. There is possibly a trade-off between accuracy in time resolution and the number of measures handled for each experiment round. Assessing several measures in the same round makes it possible to reduce the total number of rounds needed in a benchmark, i.e., the total execution time is reduced. Since a short execution time is a desirable property of a benchmark this problem must be examined more closely.

5.1.3 Workload Definition

The application running on top of the OS has been chosen in order to get a realistic evaluation of the OS. The application is a control and navigation system for a satellite. It was chosen because it is a realistic example of an embedded control system.

The application 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 C-compiler 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 portable.

The workload of the benchmark consists of the control application running and the requests sent to the application by an environment simulator. The application itself consists of four threads that are created, synchronized and handled by POSIX calls.

The system consists of five main tasks of execution, 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. An overview of the system is shown in Figure 5.2.

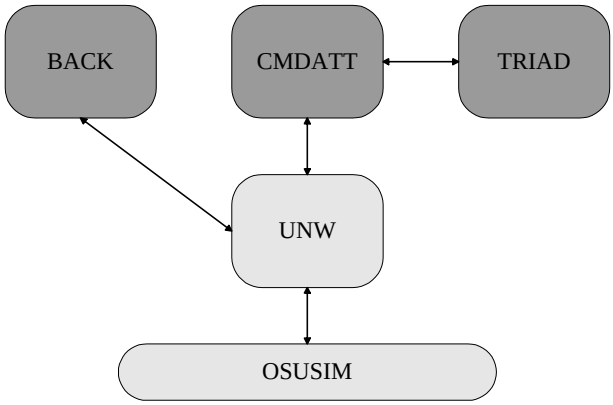

Figure 5.2 - An overview of the software system used as workload in the benchmark. CMDATT and TRIAD are responsible for the control algorithms and execute at 16 Hz and 2 Hz respectively. UNW simulates the onboard network used for communication. BACK is a background process running on low priority. OSUSIM is the environment simulator driving the system by sending data on the simulated bus.

5.1.4 Faultload Definition

The selection process used for finding a suitable stressload is outlined in the accompanying document "Workload and Faultload Selection" [ETIE3 2002]. We have chosen not to include any internal stresses, where internal is relative the OS. We have also chosen not to include any operator mistakes as an operator is not present in this embedded system. Stresses are chosen among external hardware stresses and external software stresses.

Suitable locations for injecting faults are the I/O communication between the application and the environment simulator and in applications themselves (however, then the measures based on service provision may become obsolete and other measures must be used). Interesting characteristics for the entire system include fault containment (including error propagation) and how the system handles stressful situations. Examples of faultloads for such tests are injecting faults in parallel tasks/processes and resource exhaustion tests.

There is a need for increasing the efficiency of the benchmark experiments; therefore some synchronization is needed between the workload and the faultload/background load. As a first step, a basic level of synchronization is used, where 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. As further benchmark experiments are conducted, a need for tighter synchronization between workload and fault/background load may arise.

5.1.5 Procedures and Rules for the Benchmark

For a benchmark it is important that the process by which one performs the benchmark is well documented. At this stage of the project a high level description of the benchmark process can be described as a three step process:

- The benchmark is executed with no stresses present in the system. This is done to get an oracle, i.e., baseline measures, to be able to both check the correctness of the responses (or at least the consistency) and the relative degradation in performance.
- Each measure is evaluated with the stresses identified. Some measures may be possible to measure at the same time, some not. The effect of the stresses is observed and logged. Each measurement may need to be repeated several times in order to get enough confidence in the result.
- For each measurement, all failures are classified according to the failure modes and some preparation of the data is done to facilitate a fair comparison and interpretation of the result.

Clearly, step number two is the most difficult one. There are some issues that need to be resolved. We will, as a start, insert stresses in a stream, one after the other, without restarting the system between insertions. This can hopefully lead to a shorter execution time. There are of course instances where restarts are needed, for instance if the OS or application has hanged or crashed. Also, some synchronization is needed between the insertions of two stresses, such that the effect of one stress is recorded before the next one is inserted. Inserting stresses in a stream means that the effect of long latency faults may be observed, i.e., if an inserted stress has a very long latency before it has any effect on the system, that effect may arise when the system is stressed further. This makes the effects a little more complex to interpret, but at the same time makes it a bit more realistic. We will only insert stresses one at a time, although multiple stresses could be of interest they will make the experiments more complex.

The logging of data will be made in normal files residing on the local hard drive. There are both advantages and disadvantages to such an approach. The recording of data is made simpler since it does not rely on a network connection and a second physical machine for data recording. However, there is a risk that data may be overwritten if the OS is malfunctioning due to some stress. This risk has to be assessed later and the choice of data recording method revised accordingly.

5.2. Embedded system for automotive application

The extreme diversity of embedded applications makes generalizations difficult. Nonetheless, there has been an emerging interest in the entire range of embedded systems (e.g., [Cole 1995, Gajski 1994].)

Phil Koopman [Koopman 1996] states one possible organization for an embedded system. In addition to the CPU and memory hierarchy, there are a variety of interfaces that enable the system to measure, manipulate, and otherwise interact with the external environment. See Figure 5.3.

The physical implementation of embedded systems can be done using a micro-controller or a micro-processor as CPU's. There is little difference between a micro-processor and a micro-controller. Basically, a micro-processor can be defined as a embedded processor in a single package. Microprocessors need specialised components to communicate with external devices.

These components must be able to send and receive information using communication protocols (chips for 803.2 protocol, RS-232 interface, parallel ports, USB ports, etc.) These components are mounted on PCBs (Printed Circuit Boards) and use modems or coaxial cables to communicate with the hardware of the application or external devices.

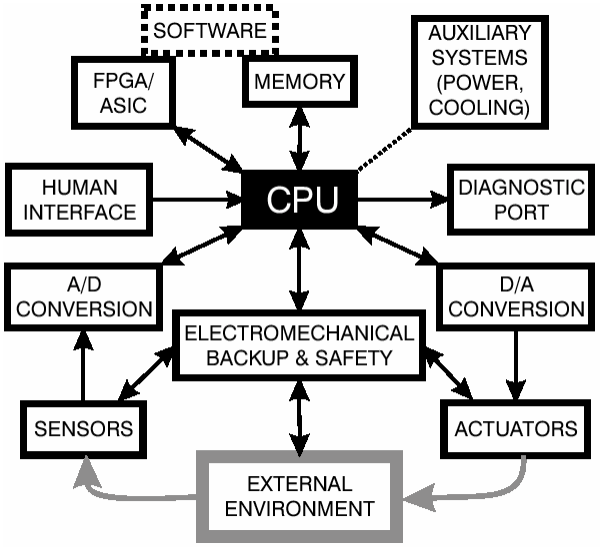

Figure 5.3 - An embedded system [Koopman 96] encompasses the CPU as well as many other resources.

On the other hand, the basic concept behind a micro-controller is the integration in a single package of a micro-processor, memory, and additional hardware (parallel or serial ports and pins) to gain communication capabilities. This integration reduces the space requirements and increases the reliability of the system.

Since this idea has been very helpful in the development of applications, it was proposed to implement the whole system in a single package. This option is called System on a Chip (SoC). In this case, most components of the embedded system are integrated in a single package.

In the case of embedded industrial applications one possible way of developing the industrial application software is using a function library. These functions can access the hardware to control using the serial/parallel ports. The hardware is in charge of taking the new values of the variables to the serial/parallel ports or to the dual port memory inputs and generating an interruption for the control software to update the variables in memory.

Another way of developing the control software is using micro-kernels. The main goal of a kernel is to facilitate the control algorithm development grouping the tasks that must be performed. A kernel consists of a task scheduler to switch from task to task and a set of functions to access the micro-processor/micro-controller input and output ports.

Finally, it is possible to use real time operating systems (RTOS). Although the aim of a RTOS is to provide the programmer with a standard interface, there is no standard API for embedded

application development due to the wide diversity of embedded industrial systems and applications. There is a lot of work in progress in this area.

Briefly, the software development for industrial applications can make use of:

- A set of functions to access the system hardware directly. These functions are hardware dependant (figure 5.4).
- A micro-kernel that provides a set of functions to access the hardware and a task scheduler and another set of functions to create new tasks and switch from task to task (figure 5.5).
- A RTOS. In this case, the application development is more elegant and easily portable to other RTOS in principle (figure 5.6)

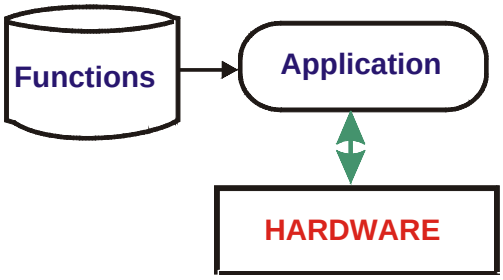

Figure 5.4 - Application built using a set of basic functions to access hardware directly.

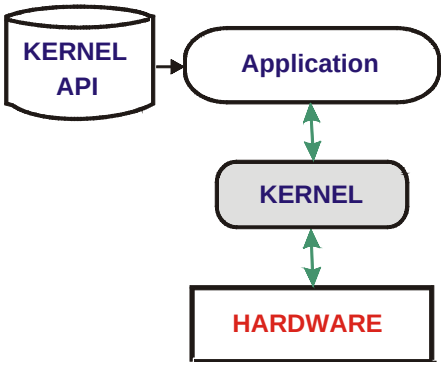

Figure 5.5 - Application built using a KERNEL to access hardware.

The most common way of information exchanging with the external environment is variable sharing. These variables are located in predefined memory areas. The vital information to keep control of the process is stored in these variables.

The process control is described by means of algorithms that show the different tasks the hardware must perform.

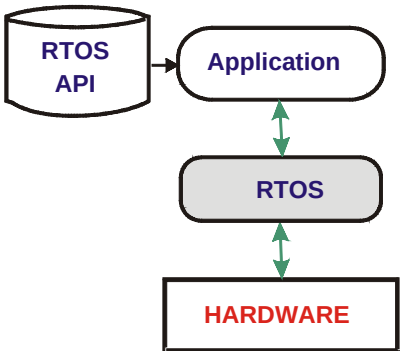

Figure 5.6 - Embedded Application built using the services of a RTOS.

5.2.1. Considered system, benchmarking context, measures

In our case of study, the application under study is a reduced version of a "diesel engine electronic control unit" (ECU). It is implemented as a SoC. Figure 5.7 shows the inputs and outputs of the ECU.

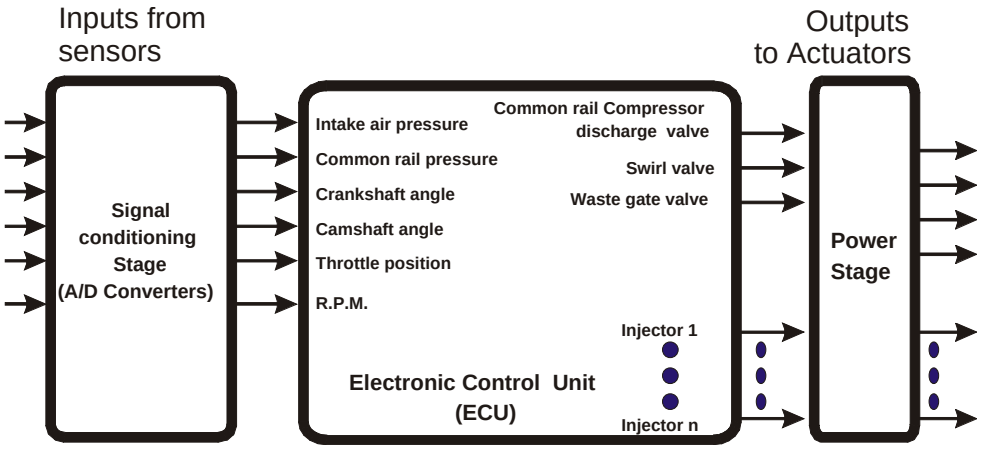

Figure 5.7 -Inputs and outputs of the "diesel engine electronic control unit" (ECU.)

The physical implementation of the embedded control application is made using the architecture shown in figure 5.8.

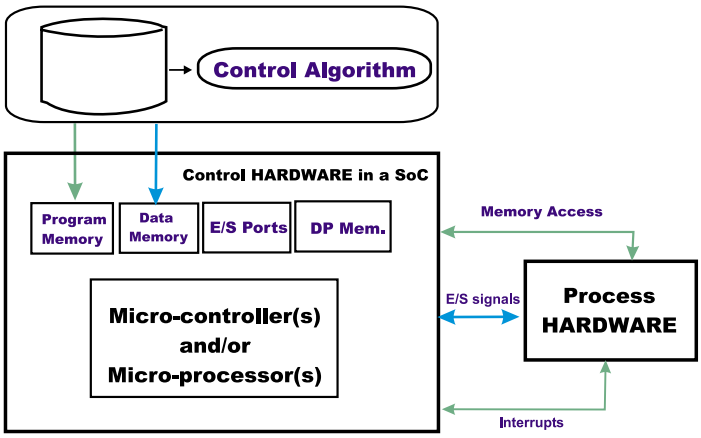

Figure 5.8 - Physical implementation of the embedded automotive control application.

The micro-controller MPC 555 will be used as processor. Its architecture is shown in Figure 5.9.

As established in deliverable [ETIE3 2002] the ECU controls both the fuel injection system and the air management system by means of two independent processes: The injection control loop (ICL)and the air management control loop.

The ECU control algorithms for both processes can be described by means of five tasks. They are described below. Figure 5.10 shows their dependences.

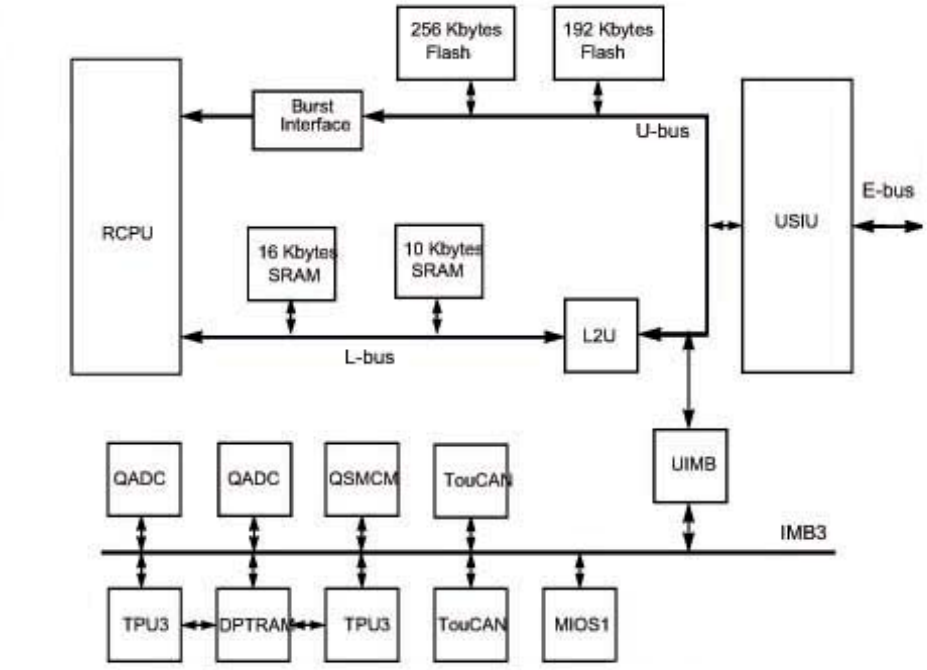

Figure 5.9 - Block Diagram of the Motorola Power PC 555.

First task: In this task it is necessary to *lookup data* (lookup in a table*)* and perform *data interpolation*. The aim of this task 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. The 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 are implemented as functions and they run concurrently on the micro-controller. 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.

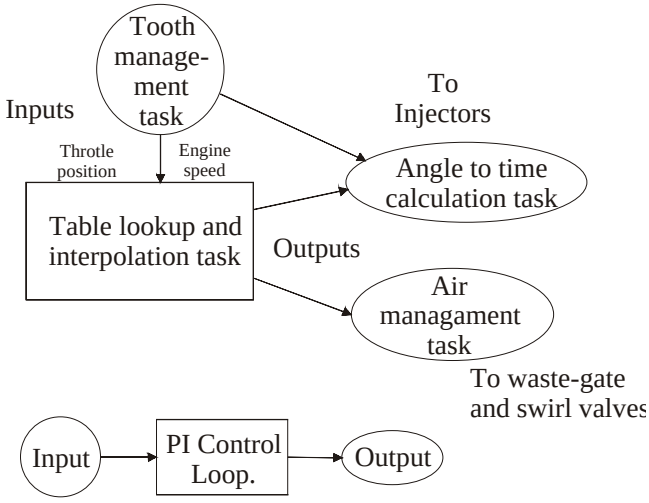

Figure 5.10 - Workload Tasks.

In the application we have two control points to stress; engine speed that reduce the cycle time to do the big part of the application and the timer that specify the time loop control for the common rail pressure. At high engine speed and short time for the common rail pressure loop we can obtain a heavy stress load.

The control algorithm (software) is implemented using a kernel API and using a RTOS. Under such circumstances there are two different SUBs.

a) The embedded industrial control system implemented using a Kernel API

b) The embedded industrial control system implemented using a RTOS.

5.2.2. Short description of the benchmarking set-up (BPF)

In order to study the SUB we have implemented an experimental platform for the benchmarking experiments (see figure 5.11).

The platform is composed by:

- The evaluation board for the MPC565 Micro-controller.
- The NEXUS BDM emulator with the trace board.
- A PC for managing the emulator and some standard I/O boards with a PC to emulate the inputs. We need these signals to take the possibility of injecting faults into the inputs signals.

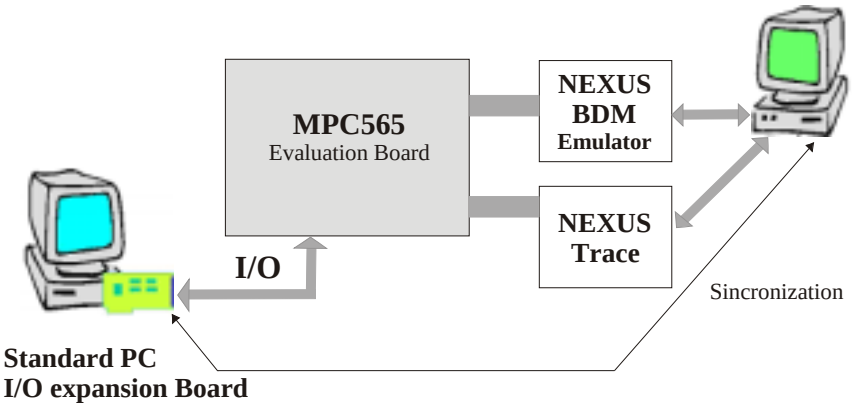

Figure 5.11 - Benchmarking set-up.

5.2.3. Measures

The aim of the fault injection is to obtain the behaviour analysis of the SUB running in the SoC with the injected faults. Measures considered include comprehensive measures such as performance in presence of faults and time to restart the system and specific measures such as error detection latency/efficiency, time of error diagnosis, etc.

From the point of view of a real time system we can consider the response time characterisation of the application, the throughput specification and the incorrect or incomplete delivery of results.

1. We can experiment with two important control-loops (see deliverable [ETIE3 2002]): the pressure injection control loop located in the common rail pressure task, and the main loop. From the point of view of response time, it will be interesting to stress the common rail pressure task.
2. The EEMBC [EEMBC] benchmark is a set of algorithms which are the most used in car and industrial applications. It has been foreseen that the diesel engine application can be implemented using the following algorithms:
    a. Table lookup & interpolation
    b. Angle to time conversion
    c. Pulse width modulation (PWM)
    d. Tooth to spark
    e. Road speed calculation
    f. Bit Manipulation

    These six algorithms constitute a subset of EEMBC benchmark, in such case, the workload of the diesel engine application can be analysed using EEMBC benchmarks.
3. In order to analyse the behaviour of the system in presence of faults it could be interesting to analyse the number of times that the loops are executed in a predefined time (throughput), the time needed to execute each task, the time necessary to store data in memory, etc.

5.2.4. Workload

In the case of the embedded system for automotive application the workload is the embedded application described above.

5.2.5. Faultload

In the embedded system for automotive application we mainly have HW faults and Operating System faults. There are no considered operator faults because a typical embedded industrial application has no interaction with an operator.

The memory of the system and the I/O lines are the best places to inject faults.

However it is important not to forget that in most cases the faults injected on the I/O lines will be represented by faults in memory. So, using SWIFI fault injection methods could be enough to generate the fault load. And it will be directed to the system memory because in the memory system it is stored the state of the system.

In the case of the diesel engine control, we will carry out two types of experiments. They will offer the possibility to study two different types of fault: *hardware faults, software faults and if applicable operating system (OS) faults.*

In the first case we emulate the effects of *hardware faults* in the system: hardware faults are due to either external perturbations or incorrect function of internal components.

In the second case (operating system faults) we will study the OS behaviour in the presence of faults in the system calls (APIs).

5.2.6. Fault injection method (implementability)

In the special case of the ECU:

- The critical data needed by the engine is stored in a special register, called the engine status register.
- In order to access the engine status register we are going to use the NEXUS debug standard mechanism implemented in the Motorola MPC565 micro-controller (it will be included in future micro-controllers as debug mechanism).
- A personal computer (PC) (with the NEXUS emulator) 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. The MPC565 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 MPC565's characteristics to develop a new software implemented 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.

  In the second case, OS fault, the PC will take charge of deciding the 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.

Taking into account NEXUS capabilities we will use a NEXUS-based SWIFI technique for injecting faults.

6. Conclusion

The development of a comprehensive set of dependability benchmark prototypes is an important step to advance the research issues of DBench in a truly dependability benchmarking environment. This deliverable describes the dependability benchmark prototypes that are being

developed in DBench. As planned, these prototypes cover two major application areas (embedded and transactional applications) and are being developed for two different families of COTS operating systems (Windows and Linux).

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. The main components of a dependability benchmark are measures, workload, faultload, procedures & rules, and the experimental benchmark setup. Thus, the definition of the dependability benchmark prototypes consists of the description of each benchmark component.

General guidelines for the definition of dependability benchmarks were proposed, following results available from previous research in the project. These guidelines include three major steps:

1. Identification of the dimensions that characterise the dependability benchmark under specification. This means that a real benchmark has to be defined for a very specific context (a well defined application area, a given type of target system, etc), which is described by a specific instantiation of the categorization dimensions [CF2 2002].

2. Definition of the dependability benchmark measures, which should be the first dependability benchmark element to be specified. Different dependability benchmark categories will have different set of measures.

3. In the third step we define the remaining dependability benchmark components, which are largely determined by the elements defined in the first two steps. These components are the workload, faultload, procedures & rules, and the experimental benchmark setup.

Two complementary views of dependability benchmarking were proposed: external and internal dependability benchmarks. The first ones compare, in a standard way, the dependability of alternative or competitive systems according to one or several dependability attributes, while the primary scope of internal dependability benchmarks is to characterize the dependability of a system or a system component in order to identify weak parts. External benchmarks are more demanding concerning benchmark portability and representativeness, while internal benchmarks allow a more complete characterization of the system/component under benchmark.

External benchmarks involve standard results for public distribution while internal benchmarks are used mainly for system validation and tuning. Thus, external benchmarks aim at finding the best existing solution and the ‘best practice’ by comparison while internal benchmarks aim at improving an existing solution. A drawback of external benchmarks is that they may not yield enough information for improvements; a drawback of internal benchmarks is that certain weak points may not get detected when looking at only one possible solution.

All the dependability benchmark prototypes under development in DBench will be used as research platforms for both the external and the internal dependability benchmark perspectives. Furthermore, the benchmark prototypes will provide realistic benchmarking environments to validate the different approaches and techniques that have resulted from WP2, and eventually evolve the prototypes into true examples of dependability benchmarks.

Table 6.1 summarizes the key aspects of the five prototypes under development (each line in the table represents one prototype). The profusion of measures and the multiple possibilities for the

workload and faultload reflect the fact that the prototypes will be used in a first phase as an experimentation environment, and not as truly dependability benchmarks, as mentioned before.

Application area	**Type of SUB**	**Type of measures** (in presence of faultload)	**Workload**	**Faultload**
General purpose	Operating system	▪ OS robustness ▪ OS-level specific measures ▪ Workload-level measures ▪ Timing measures	▪ Synthetic programs ▪ Real applications ▪ Realistic applications	▪ Bit-flip and invalid API parameters ▪ Memory bit-flip + low-level educated mutations ▪ Memory bit-flip
OLTP	Transactional system	▪ Transaction throughput ▪ Availability ▪ DBMS-level specific measures ▪ DBMS-level specific dependability measures	▪ TPC-C based workload	▪ Scripts simulating real operator faults ▪ Low-level educated mutations ▪ Operating system simulation ▪ Operating system simulation + hardware faults emulated by scripts
Web-based OLTP	Transactional system	▪ Transaction throughput ▪ Availability ▪ DBMS-level specific dependability measures ▪ Network-level measures	▪ TPC-C based workload ▪ SPECweb99	▪ Scripts simulating real operator faults ▪ Low-level educated mutations ▪ Operating system simulation ▪ Operating system simulation + hardware faults emulated by scripts
Automotive control	Embedded system	▪ Application performance-related ▪ Specific dependability measures	Diesel engine electronic control unit	▪ Stressful workload ▪ Memory (single and multiple) bit-flips
Space control/ navigation	Embedded system	▪ Application performance-related ▪ Specific dependability measures ▪ Failure modes	Control and navigation application for a satellite	▪ Stressful workload ▪ Memory (single and multiple) bit-flips

Table 6.1 – Summary of key aspects of the five prototypes under development.

One relevant aspect in the measures column in table 6.1 is that the dependability benchmarks for the application areas share similar types of measures (see the two prototypes for transactional applications and the prototypes for embedded applications), which suggests that we will have a reasonably base for cross-comparison of results.

The faultload of the different prototypes also share similar fault injection techniques, which constitute an excellent opportunity to compare different fault injection methods in a large variety of systems.

References

[Arlat 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", IEEE Transactions on Software Engineering, 16 (2), pp.166-182, February 1990.

[Arlat 2002] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Transactions on Computers*, 51 (2), pp.138-163, February 2002.

[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.

[Carreira 1998] J. Carreira, H. Madeira and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Transactions on Software Engineering*, 24 (2), pp.125-136, February 1998.

[CF1 2001] J. Arlat, K. Kanoun, H. Madeira, J. V. Busquets, T. Jarboui, A. Johansson, R. Lindström S. Blanc, Y. Crouzet, J. Durães, J.-C. Fabre, P. Gil, M. Kaâniche, J. J. Serrano, J. G. Silva, N. Suri and M. Vieira, "*Conceptual Framework, Deliverable CF1, State of the Art*", DBench Project, IST 2000-25425, August 2001.

[CF2 2001] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johanson and R. Lindström, "*Preliminary Dependability Benchmark Framework*", DBench Project, IST 2000-25425, August 2001.

[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.

[Cole 1995] Bernard Cole, "Architectures overlap applications", Electronic Engineering Times, March 20, 1995, pp. 40,64-65.

[Durães 2002] J. Durães, H. Madeira, "Emulation of software faults by selective mutations at machine-code level", Technical Report, TR-DEI-003-2002, Departamento de Engenharia Informática, FCTUC, ISSN 0873-9293, February 2002.

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

[ETIE1 2002] "*Measurements*", DBench Project, IST 2000-25425, June 2002.

[ETIE2 2002] "*Fault Representativeness*", DBench Project, IST 2000-25425, June 2002.

[ETIE3 2002] "*Workload and Faultload Selection*", DBench Project, IST 2000-25425, June 2002.

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

[Gajski 1994] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan & Jie Gong, Specification and Design of Embedded Systems, PTR Prentice Hall, Englewood Cliffs NJ, 1994.

[Gray 1990] J. Gray, "A Census of Tandem Systems Availability Between 1985 and 1990", *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 409-418, October 1990.

[Kalyanakrishnam 1999] M. Kalyanakrishnam, Z. Kalbarczyk, R. Iyer, "Failure Data Analysis of a LAN of Windows NT Based Computers", *Symposium on Reliable Distributed Database Systems, SRDS18*, October, Switzerland, pp. 178-187, 1999.

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

[Koopman 1996] Phil Koopman. "Embedded System Design Issues". Proceedings of the International Conference on Computer Design (ICCD 96)

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

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

[Musa 1999] J. Musa, "Software Reliability Engineering", McGraw-Hill, 1996.

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

[Rodríguez 2002] M. Rodríguez, J.-C. Fabre and J. Arlat, "Assessment of Real-Time Systems by Fault-Injection", in *Proc. European Safety and Reliability Conference (ESREL-2002)*, (Lyon, France), pp.101-108, 2002.

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

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

[Sullivan 1991] M. Sullivan and R. Chillarege, "Software defects and their impact on systems availability – A study of field failures on operating systems", *Proceedings of the 21st IEEE Fault Tolerant Computing Symposium, FTCS-21*, pp. 2-9, June 1991.

[Sunbelt 1999] Sunbelt International, "NT Reliability Survey Results", http://www.sunbelt-software.com/ntrelres3.htm, published March, 23, 1999.

[TPC 2001] Transaction Processing Performance Consortium, "TPC Benchmark C, Standard Specification, Version 5.0," 2001, available at: http://www.tpc.org/tpcc/.

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

[Vieira 2002] Marco Vieira and Henrique Madeira, "Recovery and Performance Balance of a COTS DBMS in the Presence of Operator Faults", *International Performance and Dependability Symposium, IPDS 2002, (jointly organized with DSN-2002)*, Bethesda, Maryland, USA, June 23-26, 2002.