Chapter 7

Cross Exploitation of Results, Lessons Learned and Recommendations for Future Work

Abstract

The previous chapters of this deliverable have introduced the basic concepts related to dependability benchmarking and the several benchmark specifications deployed by the DBench project in the domains of operating systems, embedded systems and transactional systems. This last chapter presents the general conclusions of this research. First, the major results of the project are cross-exploited according to the application domain of each dependability benchmark. Then, the different lessons learned with this work are discussed. The chapter concludes with recommendations for future work on dependability benchmarking and a short presentation of the work that is being started as a follow up of DBench.

7.1. Introduction

Although the benchmarks presented in previous chapters may be perceived as being completely unrelated at a first glance, they are all part of a global strategy aimed at exploring, under different perspectives and within different application domains, the vast and complex problem of benchmarking the dependability of computer systems. This is the only way to improve our understanding of the problem, which is an unavoidable step towards the harmonisation of the different approaches defined in this deliverable.

The main results derived from the research performed in DBench fold in four main categories: (i) the general concepts, which are common to all the benchmarks deployed in the project; (ii) the specifications of the benchmarks for the application areas and classes of systems considered; (iii) the prototypes, which have illustrated the feasibility of the proposed approaches while motivating their interest; and finally, (iv) the experimental results obtained, which can only be exploited (or compared) within the same class of target systems and under the same specific conditions: same measures and same experimental dimensions.

Due to the heterogeneity of the considered targets, the cross exploitation of the above results makes only sense when conducted under the perspective of the application domain to which these results apply. More specific comparisons, like those based on the measures defined by the benchmarks, are only feasible when all the benchmarking dimensions of the considered benchmarks are defined in the same terms. As stated in chapter 1 (see Figure 1-1), all DBench benchmarks share the same basic concepts and are built on top of three main dependability benchmarking dimensions called categorisation, measures and experimentation. The categorisation dimension defines the context of the benchmark; the measures dimension enumerates its measures and the last set of these dimensions describes the elements required for experimentation, mainly the workload, the faultload and the benchmark conduct rules. If one the three dimensions of a benchmark differs from the equivalent dimension of another benchmark, then these two benchmarks are different. This is why comparison of benchmark measures makes only sense when obtained from benchmarks (i) defined within the same application domain, and (ii) proposing similar benchmark conduct rules for the computation of such measures.

However, our experience shows that, despite their differences, all dependability benchmarks share the same set of elements required for experimentation, i.e. all of them require the definition of an execution profile (workload and faultload) to exercise de target systems and a set of benchmark conduct rules to drive the execution of the experiments. This issue is exploited in Section 7.2 in order to lead a cross-exploitation of the results issued from the DBench research. Then, Section 7.3 discusses the lessons learned with this work and finally, Section 7.4 provides recommendations for future work in the domain of dependability benchmarking.

7.2. Cross exploitation of results

The following subsections compare the various benchmarks defined all through the previous chapters of the book under the viewpoint of their benchmarking context. The discussion is thus conducted on the domain of operating systems, transactional (OLTP) systems and the embedded systems. In the context of these systems, the similarities, the differences and the complementarity of the proposed benchmarks are discussed.

7.2.1. Benchmarks for Operating Systems

For general purpose OSs (GPOSs, see Chapter 2), we put emphasis on (i) robustness with respect to application faults, which are manifested as corrupted parameter values in system calls, ii) OS reaction time in the presence of these faults and iii) system restart time in the presence of these faults. Also, we evaluate the workload execution time. For real-time kernels (RTK, see Chapter 3), the emphasis was put on characterizing the RTK function execution time in the presence of faults (faults that correspond also to application faults manifested as corrupted parameter values in system calls). In this second case, three metrics are associated to the execution time of the RTK: i) its divergence from the nominal time, ii) the frequency of cases for which this execution time is higher than the nominal time, and iii) its predictability (a measure of the probability that the execution time drops within the time constraints specified for the targeted kernel).

Indeed, both the GPOS reaction time and the RTK function execution time correspond to the time taken by the operating system to respond to a system call (system calls are usually named functions by OS providers). It has been referred to as OS reaction time in the first case, because the faults selected are expected to be detected by the OS API, which in turn should react by returning an error code or an exception. When an error code or an exception is returned, the system call is not actually executed. That is why it has not been called "function execution time". On the other hand, for RTK, the aim is to evaluate how predictable a function is responding within a time limit. The faults selected represent the range of possible values for a parameter type regardless of their semantic correctness. So even though the same measure is evaluated, using the same parameter corruption techniques, the goals of the two benchmarks are very different and thus the faultloads are different. GPOSs benchmark is mainly concerned by OS correct behaviour (in terms of correct returned values), while the RTK benchmark is concerned by correctness in time (responding within a time limit).

Of course, the workloads are different in these benchmarks. For GPOSs, it is possible to use the workload of a performance benchmark, while for RTK the workload has been developed for benchmarking purpose to meet the application domain requirements.

7.2.2. Benchmarks for Embedded Systems

Contrary to the number of different transactional systems and operating systems available in the market, the number of existing embedded systems is very low and most of times the existing systems are very difficult to obtain for evaluation purposes. Another important issue is the fact that little research has been performed so far on benchmarking embedded systems applications. This cumulus of concerns has prevented us from (i) conducting our benchmarking experiments on many different targets of the classes of embedded systems considered and (ii) basing the specification of our dependability benchmarks on other performance benchmarks existing in the domain. So, the benchmarks specified in DBench for embedded systems have been defined nearly from scratch by adapting the knowledge on benchmarking that currently exists in other application domains.

Embedded systems have been benchmarked from the perspective of their operating systems and their applications. In the first benchmark (see chapter 3), the goal was to characterize the determinism of the response times of real-time kernels (RTKs) in space domain applications. In the second one (see chapter 4), the aim was to characterise the safety of automotive engine control applications running inside electronic control units (ECUs). This diversity of benchmarking contexts and targets has enriched our study on embedded systems and justifies the differences existing between the proposed approaches. Despite these differences, the benchmarks can be compared from the viewpoint of their experimental dimensions.

From the viewpoint of the faultload, real-time kernels are evaluated with respect to various incorrect input parameters that applications running on top of them are liable to supply. On the other hand, engine control applications are evaluated with respect to faults that may affect their underlying hardware support. Although different, these faultloads are complementary as they can inspire extensions in the considered benchmarks. RTK related benchmarks can be extended, for instance, in order to consider the impact that hardware faults may have on the execution of real-time kernels. On the other hand, incorrect input parameter faults are of interest for extending ECU related benchmarks in order to consider more complex (distributed) automotive control applications, like the antilock braking control systems, that need to cooperate for managing components that are de-localized in a vehicle.

As far as real-time kernels are concerned, the workload is the application executing on top of real-time kernel considered as benchmark target. For engine control applications, the workload is constituted by the set of readings obtained from the sensors connected to the throttle and the engine. These readings are those that feed the benchmark target, which is the control software running inside the ECU. The reader should notice that, despite the differences between these targets, the workloads are defined with two different and complementary perspectives. In the RTK related benchmark, the benchmark target runs a software-oriented workload defined as a set of programs. In the ECU related benchmark, the workload is hardware-oriented, since defined in terms of sensor information. These software- and hardware-oriented workloads cover a large spectrum of the possibilities that one can find during the definition of an embedded system (dependability) benchmark.

Despite the differences perceived at a first glance, the dependability measures defined in the two benchmarks for embedded systems also maintain a certain relation among them. It is well-know that most critical embedded systems are subject to severe time constraints. Although important from a performance viewpoint, violations of these time constraints have a deep impact over the safety of these systems. In the space domain, real-time kernels have strict deadlines that must be respected in order to ensure that computations are performed within bounded intervals of time. This is essential in order to maintain the system in a safe computation mode. The same is true for an engine control application where missing a deadline may lead to the generation of an improper control output that may be unsafe for the vehicle engine (with possible catastrophic consequences). At this point, we must underline the importance that the application domain has on the definition of the benchmark measures. In the space domain, real-time processing is a great challenge, so it is important to predict the non-occurrence of the unsafe situations described above. This is why RTK focuses on the (mathematical) characterisation of this essential feature: the predictability of response time of kernel functions. In the automotive control domain, time failures (missing deadlines) have the same importance than value failures. In the case of an engine ECU, both types of failures are susceptible to lead the control software to compute and apply improper actions that may have unsafe consequences for the considered engine.

7.2.3. Benchmarks for Transactional Systems

We have defined two complementary approaches to benchmark transactional systems: while DBench-OLTP follows benchmarking tradition of reporting experiment measures only and is specifically targeted for the comparison of the benchmarked systems or components, the TPC-C-Depend uses available information on failure and repair rates and includes extra modelling steps to characterize stationary availability of the benchmarked system and the total cost of failures.

The DBench-OLTP benchmark evaluates performance and price per transaction with no faults (baseline TPC-C measures), the same measures in the presence of faults, and experiment dependability measures such as system availability (both from the server and clients point of view) and number of data integrity errors observed in the benchmark execution. All the measures are obtained from experimentation and are collected from the point-of-view of the system users, portraying an end-to-end characterization of performance and dependability from the end-user point-of-view.

The baseline performance and price per transaction measures of DBench-OLTP are inherited from the TPC-C performance benchmark. However, while in TPC-C these measures express optimised performance, in DBench-OLTP represent the compromise between performance and recoverability established by the benchmark performer. In fact, the system settings used to measure the baseline performance and price per transaction are exactly the same used to obtain the same measures in the presence of faults, forcing the benchmark performer to tune the system for meaningful performance measures instead of unrealistic peak performance.

TPC-C-Depend evaluates the system stationary availability and the total cost of failures. These measures are evaluated based on experimentation on the benchmarked system (to evaluate the distribution of the various failure modes of the system) and on the failure and repair rates that are usually obtained from outside the benchmark itself. Referring to Figure 1.3 of Chapter 1, DBench-OLTP is based only on the experimentation layer and provides experimental measures while TPC-C-Depend requires an analysis and a modelling step to obtain the benchmark measures.

7.3. Lessons learned

This section presents the general lessons learned from the work performed all through this project. Then, the discussion is specialized in order to provide the specific lessons learned in each one of the application domains considered in DBench.

7.3.1. General Lessons learned

Our experience confirms that the scope of dependability benchmarking differs from the scope of classical fault injection experiments as discussed in [Koopman 2002]. While fault injection experiments are designed and tuned for a specific system, dependability benchmarking aims at providing a fair basis for comparison of several existing solutions. Just like standard performance benchmarks, dependability benchmarks must be designed for a class of similar systems or for a specific application area. The portability aspects play an important role in the design of a dependability benchmark, imposing strict restrictions on other benchmark components (e.g., the fault load must contain only faults that can be reproduced and are meaningful in all the class of systems addressed by the benchmark). This was our first lesson learned.

Benchmark experiments must generate reproducible results obtained in a standard manner that complies with the benchmark specification. Therefore, rules of regulations are required on how to perform such experiments and what information concerning experiment details are to made public afterwards (disclosure rules). Such auditing and publication rules are necessary to ensure fair comparison between competitive systems.

Thus, a dependability benchmark consists of the specification of a number of different elements, such as a faultload, a workload and of a set of general rules unambiguously describing the conduct of benchmarking experiments, i.e. how to deduce the benchmark measures from experimentation.

Concerning the definition of faultloads, the research on the three major types of faults to be emulated by the faultload (operator faults, hardware faults and software faults) has lead to a variety of lessons and conclusions:

- The emulation of operator faults only makes sense when the benchmark target (BT) or the system under benchmark (SUB) includes complex interfaces used by human operators (typically system administrators). This is the case of transactional systems and partially the case of operating systems. Operator faults clearly do not apply to embedded systems. When applied, the emulation of realistic operator faults revealed to be very easy (and portable), as the injection of the faults just consists in the execution of the wrong commands that represent operator mistakes.

- Low level hardware faults such as the classic hardware bit-flips (and even other more elaborated hardware fault models such as bus faults, processor faults, etc) can only be used when the SUB timings allows experiments with very large number of faults, in order to assure statistical repeatability. In fact, it is difficult to inject this kind of faults in a repeatable way, especially in very complex systems, thus the faultload must include thousands of faults. Our experience has shown that this is possible for small SUB, such as the embedded systems, but it is definitely not recommended for the benchmarking of large transactional systems, when the time needed to restart the system and rump up to achieve nominal performance may reach tens of minutes. This way, we have used low level hardware faults (bit-flip faults) in the benchmark for embedded systems while the hardware faults used in the faultload of the benchmark for the transactional systems are based on component failures (board failure, network failure, disk failures, etc), representing possible consequences of low level hardware faults.

- The fact that low level hardware faults require a fault injection tool to apply the faultload is not a serious obstacle to dependability benchmark. The existence of debug and calibration standards for real-time embedded systems such as Nexus provides the necessary hook to perform fault injection. In the other side of the spectrum, our experience has proved that it is possible to built plug-and-play fault injector tools for typical servers, such the ones used in transactional systems [Costa 2003].

- The need to use software faults in the benchmark faultload posed a particular difficult challenge, as the emulation of software faults in a realistic way was not well established in the literature at the beginning of the project. The research undertaken in DBench lead to a new software fault classification based on field studies and a new technique to emulate software faults, specifically designed for dependability benchmarking of real systems (i.e., it assumes the source code of the software modules is not available, which is the rule in commercial systems). The lesson learned was that the emulation of software faults in the executable code of software modules is possible and can be easily incorporated in the benchmark specification (in practice, it is provided as a tool and a library of faults ready to download with the textual specification).

- A complementary approach to emulate software faults also used in DBench is the use of erroneous parameters in the APIs. That is, instead of injecting faults in the code of software modules, the faults are emulated by their possible consequences at the interface level. This approach has proved to be particularly well adapted to the benchmarking of operating systems, where the robustness aspects play an important role. In some sense, the injection of faults through an API is the software equivalent of the failures in hardware modules. In fact, the faults injected through an API represent the possible consequences of internal software faults, while component failures represent the possible consequence of low level hardware faults.

- One very important lesson learned concerns the fault locations. The faults must be injected in the SUB components/modules that are not part of the BT. This is essential, as the benchmark must not change the BT. This is particularly relevant for the software faults injected in the code, as they actually change the code, which consequently changes in the system. The conclusion is that the faults have to be injected in the SUB modules that do not belong to the BT, in order to observe the behaviour of the BT in an environment with faulty modules.

- The number of faults considered in a benchmark experiment, i.e. the size of the faultload, must be balanced on a representativity and cost-related basis. This size must be sufficient for providing an acceptable degree of confidence in the resulting benchmark measures, while remaining affordable (from both time and economical cost viewpoints) to the benchmark performer. Although our experience shows that this compromise must exist, how to handle it remains, for the time being, opened and it requires further research.

Regarding the workload definition, we were expecting to rely heavily on performance benchmarks for the definition of the workload. This was possible for transactional systems and the two benchmarks defined for such type of systems use the TPC-C performance benchmark workload. Even for general purpose operating systems, we were able to use the workload of TPC-C client. However, for embedded systems, we could not proceed with the same approach due to the lack of well accepted performance benchmarks in this domain. The approach in this case was to define workloads for dependability benchmarking based on others that are either used in certification processes or representative of the considered application domain. For automotive embedded systems, for instance, the workloads selected are those typically used in Europe for emission certification of light duty vehicles.

Another important issue in the specification of the workload is the definition of the outputs expected from the target system when running such workload. Contrary to pure performance benchmarks, a dependability benchmark requires such outputs in order to be able to decide whether or not the system under test is performing correctly in the presence of faults.

The comparison of computing systems based on the results issued from experimentation with different benchmarks must be performed with a lot of care. It must be noted that this comparison makes only sense when all the dimensions of such benchmarks (categorisation, measures and experimentation) are in phase. This means that benchmark measures are only comparable with others when (i) they are obtained targeting the same class of systems and (ii) they are retrieved following a similar benchmark procedure. Although these properties are easy to guarantee when the measures are retrieved using the same benchmark prototype, they are more difficult to ensure when obtained using different implementations of the same benchmark specification. At this level, variations in the interpretation of the benchmark objectives, the benchmark measures or the benchmark conduct may invalidate, from a comparison viewpoint, the results finally obtained. However, it must be noted that demonstrating to what extend different prototypes implement the same benchmark specification is a great challenge. The best way to carry out with this issue is to provide as unambiguous specifications as possible.

7.3.2. Lessons learned for Operating Systems

The work carried out with general purpose operating systems has shown that, from a conceptual view point, collecting information on the workload completion state (correct or erroneous) enriches, with any doubt, the definition of the OS benchmark. However, its implementation requires deep knowledge of the workload behaviour in order to distinguish between the correct and erroneous completion states. It appeared that TPC-C is not a very convenient workload with respect to this criterion. It is intended to solicit an external system (the database server) with transactions in order to evaluate the server performance. We do not have any means to check the workload state. The only way to check for correctness is to compare the OS response to a system call with a corrupted parameter value with its answer to the same system call with the correct parameter value. If the two answers are different, it can be deduced that most probably the workload completion will be erroneous. However, in some cases, no answer is expected from the OS and we cannot conclude. Moreover, the state of the workload may be altered later after execution of a few more system calls and there is no easy way to check for correctness. Thus the observation of the workload state impacts the selection of an appropriate workload.

Concerning real-time operating systems, it is very important to notice that, as far as we are aware, performance benchmarks for space domain applications are currently not available. This lack makes difficult the specification of the benchmark since we cannot rely on any previous work. One of the problems we have only partially solved concerns the definition of a representative workload for a real-time space kernel. As underlined several times in previous chapters, the workload must provide abstractions of the type of applications and algorithms used on the target (in our case, space) domain but these abstractions must remain as close as possible to the reality in order to remain representative. In our case, the proposed workload has been defined according to the set of functionalities present in typical applications, in almost all modern satellites, on top of the satellite real-time kernel. Although this is a first step towards the definition of a representative workload, more research and effort are required in order to define more complete and widely accepted workloads.

7.3.3. Lessons learned for Embedded Systems

If benchmarking the dependability of a system in general is difficult, then benchmarking the dependability of an embedded system is particularly a challenge. As shown in chapters 3 and 4, the current trend is to apply as higher scales of integration as possible in embedded systems in order to reduce their needs of space and improve their performance. On the other hand, these integration scales greatly limit the observability and controllability of the software (such as real-time kernels or control applications) running inside such "integrated" systems, which are most of times manufactured as integrated circuits or systems-on-chip (SoCs). This issue has a direct impact over the specification of benchmarks for such type of systems, since it makes complex the execution of the faultload and the monitoring of the target system.

DBench proposes two different solutions to the above problem. One consists in applying the faultload in the workload: the workload is first modified according to the set of faults defined in the faultload and then, this modified version of the workload is compiled and run in the target system. The other solution is to apply the faultload using the controllability features existing in most embedded systems for debugging and calibration purposes. This second solution also benefits from the monitoring capabilities of such features that can be exploited during the benchmark experiments in order to trace the execution of the target system.

7.3.4. Lessons learned for Transactional Systems

The extension of the transaction performance benchmarks philosophy to the dependability benchmarking goals proved to be very successful. The DBench-OLTP uses the workload of the TPC-C performance benchmark (the standard performance benchmark for OLTP systems), borrowed the TPC-C benchmark specification style, and introduced two new components: the measures related to dependability and the faultload. Another very positive aspect resulted from the performance benchmarks is the role played by the performance measures in the dependability benchmarks. In fact, DBench-OLTP kept the same performance measures of TPC-C and extend them to reflect the system performance in the presence of faults. This way, the benchmark portraits the impact of fault on the service provided by the transactional system and allows direct comparisons of different transactional systems.

The high complexity and the large amount of data managed by transactional systems pose specific difficulties to the design of dependability benchmarks for this type of systems. The experiments in these systems tend to take long time and all the phases of the benchmark run must be fully automatic. Our experience shows that it is possible to make the process fully automatic and that the benchmark run can be done in few days for systems of small and medium size. Another problem results from the very different sizes of transactional systems. The use of scaling rules to adapt the benchmark to the system sizes have some impact on the faultload, especially in the hardware failures faultload (larger systems have more disks and require more faults).

The use of software faultloads in these very large systems is particularly difficult, as the number of possible faults may reach several millions. At the same time, the faultload must be portable across the different systems. We solved this problem by defining a set of representative faults (i.e., faults defined according to the most frequent classes of faults observed in the field) that emulate operating system faults. That is, faults are injected in the operating system to evaluate the behaviour of the transactional engine. The benchmark uses specific software fault libraries for each operating system. That is, a given software fault library for the Windows operating system, another one for Linux, etc. This approach assures that the comparisons of different transactional systems are fair, as the set of faults is exactly the same for all the systems running on top of a given operating system.

The concrete examples of dependability benchmarking already performed using DBench-OLTP, benchmarking and comparing a large number of systems, using different databases (Oracle and PostgreSQL), different operating systems (Windows family and Linux) and using the tree types of faultloads (operator, software, and hardware faults) have shown that the dependability benchmarking of transactional systems for comparative purposes is possible and practical.

7.4. Recommendations for future work

We made a lot of progress in the project but a lot of interesting work is still required to reach the maturity of existing performance benchmarks. We have started the process of defining the benchmarking components in a standard way. Agreement could be reached only if industry adheres and helps. Dissemination is essential.

The general recommendations for future work basically focus on improvements that can be performed on the benchmarks defined in each application domain. We list these recommendations in the following subsections.

7.4.1. Recommendations for Operating Systems

General purpose operating systems need the definition of more appropriate workloads. These workloads must allow an easy observation of their behaviour after parameter corruption in order to analyse with a finer grain the activity of the target operating system. This could be an improvement of the current defined benchmark prototype.

Some work, has been done related to OS robustness with respect to faulty drivers by the partners [Durães and Madeira 2002] and [Albinet et al 2003], so it would be interesting to harmonise and integrate these works in order to make available a benchmark for general purpose OSs addressing their robustness with respect to software faults (application and driver faults) in a unified manner. Additionally, some effort has been devoted to hardware faults for the embedded automotive systems and for the OLTP systems. We should take advantage of these efforts in order to complete the OS benchmark by addressing hardware faults as well.

Regarding operating systems in general (either general purpose or real-time operating system), considering other measures, like error propagation, can be of interest. In order to do this, a background application could be considered in addition to the workload. The workload is to be used to solicit the OS under erroneous conditions and the background application is to be used to observe the impact of the OS on other applications running on top of the same OS (i.e., to observe the propagation of errors from one application to another one with which it does communicate explicitly, but both use the same OS.

7.4.2. Recommendations for Embedded Systems

As mentioned in the lessons learned section, one of the most important problems we have found during the definition of dependability benchmarks for embedded systems is the reduce number of standards in the domain. In addition the existing ones, like EEMBC (Embedded microprocessor benchmark consortium) benchmarks [EEMBC 2003], are mainly interested in benchmarking hardware features rather than features of the software running on top of the hardware. This absence has prevented the specification of benchmarks for embedded systems based on existing benchmarking work in the domain, and thus we have established the specification of all the benchmark elements nearly from scratch. Now, we have methodologies, tools and results that support our proposals and show the feasibility of our solutions in both the automotive and space domains. Thus, further work is required in order to study the possibility of integration of these benchmarks in the existing industrial software development processes.

More oriented to research, further work is required in order to improve the representativity of the benchmarks. On the one hand, we must define richer workloads that take into account the new functionalities being integrated in most modern spacecraft and engine control systems. In the case of automotive systems, for instance, the increasing number of embedded electronic control units makes primordial to study the dependability of such units not only in isolation, but also when they interact with other units them. The reader must notice that this issue also applies to the space domain, in which electronic units are omnipresent. So, other recommendation is to extend the research performed in DBench to more complex embedded systems in which components are physically delocalized and require applications to interact in order to coordinate their computation and reach their common goal.

On the other hand, we also advocate for further research focused on the evolution of the hardware platforms and the definition of more appropriate fault models. As stated in [Constantinescu 2002], the emergence of new technologies for improving the scales of integration in embedded systems trends to increase the ratio and importance of the hardware faults in such systems. Thus, research on hardware fault models will become more and more important and necessary for dependability benchmarking as far as these systems increase their scale of integration and enlarge their spectrum of application. In particular, our advice is to focus future research in this field on experimentation with new fault models (such as the pulse or multi bit-flip fault models) and new targets for applying the faultload (for instance, the hidden registers of the embedded system). This latter issue will also require additional research on how to improve the observability and controllability of the target embedded system. We are convinced that this is key topic will become of prime importance for the embedded systems industry in the next years.

7.4.3. Recommendations for Transactional Systems

When benchmarking transactional systems, it is essential to carefully distinguish between the benchmark per se (its specification) and its implementation. One has to make clear, who is responsible for what, e.g., who specifies the failure modes of the hardware platform components (including the disk subsystem), the operating system and the transactional engine, and who has to provide the corresponding fault rates. To this end, a full disclosure report should be provided with each benchmarking activity. Moreover, pricing information of performance benchmarks, like the one provided by the TPC-C benchmark, should be extended to cover financial loss in case of system failure as well as repair actions (which may be outside of routine maintenance).

The dependability benchmarks for transactional systems proposed by DBench focus on the evaluation of typical OLTP environments (i.e., a database centric systems) that consist of a number of users submitting their transactions via a terminal or a desktop computer connected to a database management system (DBMS) through a local area network or the Web. In a simplified view, the server is composed by three main components: the hardware platform (including the disk subsystem), the operating system, and the DBMS (as transactional engine).

However, more components may be considered in some transactional systems. Some examples are: web-servers and application servers. On the other hand, the use of distributed databases and parallel databases for high performance and availability is becoming extensive. As these issues were not considered in the dependability benchmarks defined in the DBench project, we propose the following topics for future research on dependability benchmarking for transactional systems:

– Benchmarking the dependability of web-servers: web-servers are becoming an important component of most transactional systems as most of the transactional applications are becoming web-based. Although the systems addressed in the benchmarks proposed by DBench may include a web-server, they do not focus directly on the characterization of this component (i.e., the benchmark target is the DBMS and not the Web-Server). This way, as an important part of a transactional system, the characterization of the dependability features of the web-server is becoming an important issue. The proposal of a dependability benchmark for Web-Servers is currently under research as a continuation of the work performed in the scope of DBench and the first results (comparing Apache and Abys web servers) have already been accepted for publication [Durães and Madeira 2003c].

– Benchmarking the dependability of application servers: many systems use an application server in order to move the business logic way from the client applications. An application server is a piece of software that runs on the middle of an n-tier environment and is responsible for translate raw data from the database into information with real meaning (according to the business logic) displayed on a Web browser or desktop application. Application servers allow the client applications to be soft applications that can run in machines without many resources. On the other hand, application servers provided the pooling of connections of the database and load balancing. As it is an important new component, the definition of a dependability benchmark for application servers is a very interesting topic for future work.

Extending the proposed benchmarks for distributed databases and parallel databases: distributing or parallelizing databases is a way to achieve higher performance and availability. Although the dependability benchmarks proposed by DBench can be applied to DBMS that support distributed or parallel databases, some more research is needed in order to work out how to apply these benchmarks on this type of systems. One of the key aspects to be addressed is how to apply the faultload in a distributed/parallel environment.