

# Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection\*

Diamantino Costa<sup>1</sup>, Tiago Rilho and Henrique Madeira

CISUC – Centre for Informatics and Systems of the University of Coimbra  
*http://cisuc.dei.uc.pt, Polo II, Pinhal de Marrocos, Coimbra P3030, Portugal*  
*{dino,henrique}@dei.uc.pt, rilho@student.dei.uc.pt*

## Abstract

*This paper presents and discusses observed failure modes of a common-off-the-shelf (COTS) Database Management System (DBMS) under the presence of transient operational faults induced by SWIFI. The standard Transaction Processing Performance Council (TPC) TPC-C benchmark and associated environment is used here together with fault-injection technology, building a framework that discloses both dependability and performance figures. Over 1600 faults were injected in the database server of a Client/Server computing environment built upon Oracle 8.1.5 database engine and Windows NT running on COTS machines with Intel Pentium processors. A macroscopic view on the impact of faults revealed that: 1) A large majority of the faults caused no observable abnormal impact in the database server; In 96% of hardware faults and 80% of software faults the database server behaved normally. 2) Software faults are more prone to let the database server hanging or cause abnormal terminations. 3) Up to 51% of software faults lead to observable failures in the client processes.*

## 1. Introduction

The joint evaluation of performance and dependability of common off-the-shelf (COTS) systems is a manifold motivation research. There is an increasing trend today for using COTS technology in mission critical and business critical systems. Systems designers see COTS components as an opportunity to cut development costs and minimize the risk of infant defects. However, this attitude puts a tremendous demand on COTS behavior, since they are not designed with the harsh requirements of critical systems in mind. *Do COTS savings necessarily mean less robustness?* A bit of common sense and also some field findings lead us to an affirmative answer. But another question remains -

*How less dependable can a COTS system be?* This question is relevant since many applications can tolerate a system with reduced dependability requirements, provided that there is an estimation of how less dependable the system can be.

The lack of practical dependability metrics is one factor commonly pointed out by the industry and end buyers for disregarding dependability issues in either COTS product design or purchase decision. Typically, buying dependability is an all or nothing decision (and sometimes a question of faith). For instance, it is very difficult (or even impossible) to bind a COTS computing platform to a given availability class without having representative data on the way the system behaves in the field. We advocate that dependability metrics should learn from (and walk side by side with) established performance metrics. First, this will favor its acceptance by the vendors, since lots of previous investments may be reused. Second, performance losses, even minimal ones, are often the “Achilles heel” of several dependability improvement techniques, and represents the main reason why they are not adopted by the vendor industry. By giving a sharp image of the system performance/dependability figures, we will unveil designs that “sell” only raw performance but actually behave poorly in presence of (unavoidable) faults. Thus, the joint measurement of performance and dependability figures will enable us to tune the systems/designs for different tradeoffs between dependability requirements and performance rates.

Software faults are recognized as the major cause of system outages. Existing studies show a clear predominance of software faults [1][2][3][4]. Given the huge complexity of today’s software, the weight of software faults tends to increase, making clear that the evaluation of the dependability of COTS systems must take into account software faults.

Fault injection has become an established approach of validating specific fault handling mechanisms and assessing the impact of faults in actual systems, allowing the

\*Work supported by the Portuguese Foundation for Science and Technology (FCT), Program Praxis XXI, grant No. 2/2.1/TIT/1570/95.

<sup>1</sup>Presently at CISUC, on leave from Critical Software Lda, [www.criticalsoftware.com](http://www.criticalsoftware.com); Research supported by FCT, PRAXIS XXI, grant No. BD/5636/95.

estimation of fault-tolerant system measures such as fault coverage and error latency [5]. Given the increasing relevance of software faults it is clear that fault injection technologies should be extended to the injection of this kind of faults.

In spite of having good indications that the erroneous system behavior induced by hardware faults with SWIFI techniques also emulate (some) software faults, the fact is that the accurate emulation of software faults through SWIFI is still an open issue. Recent efforts towards this end [6][7] suggest that current SWIFI technology can model the errors induced by software faults to some extent, but there is not an established way to emulate general classes of software faults by SWIFI tools yet. Nevertheless, SWIFI seems to be the most effective method to assess the dependability of COTS systems. In this study we use the XceptionNT fault injection tool, which is a version of Xception [8] for the Windows NT operating system. XceptionNT has been enhanced with new features to support emulation of software faults. One distinctive advantage of the Xception family of tools is its capability to perform fault-injection without need for source code. This is a sound advantage to perform the evaluation of COTS systems.

In this experimental study we focus on a commercial DBMS as a case study for establishing an environment for the joint evaluation of performance and dependability of COTS. Multiple factors can be found for this specific research:

1. Database applications have traditionally been an area with fault-tolerance needs, concerning both data integrity and availability. Several mechanisms and techniques needed to achieve fault-tolerance, such as transactions, checkpointing, logging, and replica control management born from the research in the database area.

2. COTS DBMS are core components of enterprise information systems (EIS), ranging from LAN based two-tier client/server systems to WAN based three-tier systems. Assuring DBMS dependability is of utmost importance, not only because they represent the reliable storage of organizations critical data, but also because its failure can adversely impact distributed enterprise applications.

3. DBMS industry holds a reputed infrastructure for performance evaluation. The set of benchmarks managed by the TPC are recognized to be among the most successful benchmark initiatives of the overall computer industry. The addition of dependability clauses to the TPC benchmarks specifications, as suggested and partly experimented in our own study, was first claimed in the TPC milieu.

This paper presents and discusses observed failure modes of a strict COTS DBMS system under the presence of transient operational faults induced by SWIFI. The impact of faults is assessed at different abstraction levels, ranging from monitoring of machine exceptions at the operating system level to trace and log files and a consistency-checking tool at the DBMS level. Consistency

tests performed at the database application level have also been performed. Database integrity is measured using the set of semantic rules and associated consistency tests (“business logic”) defined in [9]. The MTTR of observed crashes is also calculated. To summarize, this paper gives the following contributions:

- First, and to the best of our knowledge, it is the first experimental fault-and-error-injection (FEI) study conducted over a strict COTS DBMS system.

- Second, it gives insight on the failure modes of a COTS DBMS due to transient hardware faults and software faults (bugs).

- Last, but not the least, it leverages the reputed infrastructure of TPC performance benchmarks, adding up a fault load and fault impact measurements, thus enabling joint evaluation of performance and dependability figures.

The paper is organized as follows: the next section describes the related work. Section 3 surveys COTS failures in general (and specific data on COTS DBMS failures) setting the ground for the fault model herein used and elaborates on the specific issues of emulation software faults with SWIFI technology. Section 4 details the testbed, including the target system, the workload, the set of faults and the set of measurements observed. It also describes experiment setup and execution. Section 5 presents and discusses the experimental results and Section 6 concludes the paper, outlining the most significant results and summing up the main contributions.

## 2. Related work

The only work found in the literature so far that specifically cross relates experimental evaluation of both performance and dependability is [10]. This study addresses the evaluation of fault-tolerant systems and used synthetic workloads to exercise the available fault-tolerant mechanisms.

In [11] a simulation study was performed to assess formal figures for availability, reliability and mean transaction time for repairable database configurations. In [2] a study on failure data collected in a WindowsNT LAN is presented. A comprehensive survey on field outages and their root software errors in two COTS DBMS products with large distribution (IMS and DB2) is presented in [12].

A substantial research work cross relating fault injection and databases has been done in the RIO project at the University of Michigan. The project pursues the thesis that non-volatile memory (NVRAM) could be as reliable as disk on application and OS failures. In [13], a study focused on the evaluation of main memory resistance to operating system software crashes estimated that only around 2% of the injected faults lead to data corruption in the file cache. In [14], a simulation study was made to measure the contribution of transactions to the fail-stop behavior of a POSTGRES database. This work, which shares almost the same test bed with [13], concluded that 7% of the faults caused fail-stop violations (in the sense that faulty data was

written to stable storage) without transactional support. When transactions were added this figure dropped to 2%.

In [15] another fault injection study focused on a database buffer cache corruption is presented. This study provides valuable feedback to improve the design of the evaluated DMBS (a database mainly targeted to telecommunication applications) and presents measurements of performance degradation in presence of faults.

In [16] a comprehensive robustness benchmarking study of 15 POSIX compliant COTS OS is carried out. Combinations of exceptional and valid input parameters are passed to OS system calls and C library routines. System behavior is monitored to collect error notifications and failure symptoms. The most pertinent conclusion to our own study is a revealed rate of 6% to 19% of *silent failures* per single OS, i.e., situations where “an OS returns no indication of error on an experimental operation which clearly cannot be performed”. While *abort failures* were prevalent, this clearly indicates that one should not rely exclusively on crash failures when modeling COTS software faulty behavior.

From all the referenced work, the work done at University of Michigan and the study in [15] hold the closest relationship to the work presented here. However, these evaluation studies were motivated by different goals. In [15] the focus is the evaluation of the system behavior in the presence of data buffer corruption with the goal of improving the error detection mechanisms of the target system. Because of this, the errors injected were very specific, which somehow limit the applicability of the results to other systems and evaluation objectives. Ng and Chen [13] evaluated the relative reliability/robustness of main memory and disk in the presence of OS crashes. In this work, runs that did not quickly lead to OS crashes were intentionally discarded from the set of analyzed faults. Furthermore, with the purpose of accelerating the rate of activated faults, multiple faults were injected in a single run (for instance, in [24] 10 faults per run were injected). This setup is perfectly acceptable if the goal is to cause OS crashes but is somehow less accurate if we are running after other failure modes.

In our study, more than stimulating the SUT to fail quickly in order to evaluate recovery code performance and availability, we are interested in making it fail in more subtle ways. These “subtle” failure modes are a paramount means for assessing the integrity of the SUT. In fact, field studies revealed that failures with higher impact on system availability are less likely to affect system integrity [4]. Last but not least, we stick here to the single-fault model in order to be able to do root failure analysis.

### 3. Dependability evaluation of COTS systems

This section starts with surveying field data on COTS failures followed by a discussion on the specific issues of emulation software faults with SWIFI technology. Our main purpose here is to settle the ground for understanding both

the fault model and the relevant measurements made in this experimental evaluation.

#### 3.1 Fault model

The experimental evaluation of COTS dependability requires the fault models used in the experiments to be representative of actual faults. In other words, it means that the injected faults/errors must induce erroneous system behavior similar to the one caused by real faults.

Table 1 summarizes available results characterizing operational failures and its main underlying causes. The rightmost column represents failures whose root causes could not be found, failures due to environmental reasons or other minor causes. We include the evaluation works done by Gray [1] and Lee [4] despite the fact that the systems assessed do not qualify as COTS systems. This was done mainly because as far as we know similar studies on plain COTS do not exist yet. The same reason, i.e., lack of suitable data, justifies the inclusion of outcomes from [17].

While cross comparison between the different studies should be made very carefully, one clear pattern emerging from the data in Table 1 is the predominance of software

Table 1. Observed field failures by root cause

Study	Hardware	Software	Operations	Unknown/ Other
[1] Fault-tolerant Tandem product line;	7%	62%	15%	16%
[4] Tandem GUARDIAN90 OS;	4%	77%	5%	14%
[17] Survey based on input from system administrators of Windows NT servers;	20%	42%	10%	28%
[2] 70 Windows NT based mail servers;	9%	18%	22%	29%

faults. The share of operation faults (mainly due to end user or system administrator) causing system failures is also increasing. The gap between software and other causes is particularly evident in the two fault-tolerant systems [1][4]. This feature was somehow expected since these systems were specially built to address hardware faults. As a conclusion (for this subsection), the experimental evaluation of COTS dependability should rely on a mix of all these classes of faults, but with a clear predominance of software faults and operation faults.

#### 3.2 Software fault injection in COTS

Typical software faults found in the field are originated in the design/coding phases and remain undetected throughout the testing procedures thus affecting the operational phase of a software product life cycle. This class of faults has been characterized in several studies [3][4][18].

The adoption of SWIFI techniques for fault forecast and robustness evaluation of computer systems in presence of software faults is emerging. Two basic approaches can be

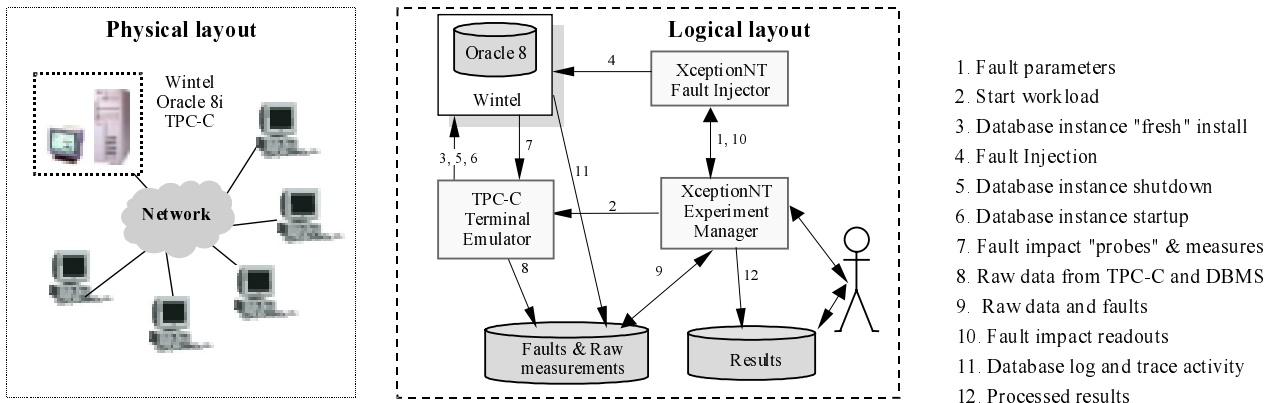


Figure 1. Testbed layout

used: run-time software implemented fault-injection (RTSWIFI) or compile time instrumentation (CTSWIFI).

Techniques based on compile time instrumentation (CTSWIFI) can only be used when the target system source code is available, which is a major drawback as the source code is not available for most COTS systems. Even a successful open source component such as Linux is clearly being powered by the increasing availability of traditional (non open source) software products running on top of this operating system.

The second approach (RTSWIFI) is basically the extension of low-intrusive SWIFI tools to the injection of software faults. This method can be used even when the source code is not available and has less setup and execution costs than compile-time instrumented fault injection. However, as the faults/errors are injected at very low level (assembly code level; processor register and memory), the mapping between the induced low level errors and the software faults defined at higher levels is not clear. Nevertheless, a recent study [6] comparing the cost and accuracy of CTSWIFI and RTSWIFI show that RTSWIFI is up to 4 times less expensive in terms of execution time while the accuracy of both techniques is almost the same (RTSWIFI is slightly less accurate).

The approach used in this work is RTSWIFI. The enhancements introduced in the XceptionNT tool to accommodate the injection of software faults consist basically of new fault triggers and fault types (see used fault sets in the next section). A new fault trigger technique using on-the-fly instruction decoding coupled to simple temporal or spatial triggers was introduced. This technique significantly reduces the setup cost by choosing the actual fault location and fault type at run-time.

#### 4. Experiment definition and setup

Figure 1 shows the testbed layout used in these experiments. The key components such as the target system, workload, set of faults and the set of measurements are described in the following subsections. The fault-injection tool is XceptionNT, a port of Xception [8] to the WindowsNT/Pentium platform. Since Xception has already

been used extensively in the past, no dedicated section is used to describe the intricacies of the Xception fault-injection process [19].

#### 4.1 Target system

Oracle™ was chosen as target DBMS because it is widely used and it represents quite well a typical (and very sophisticated) DBMS. The same reasons apply to the choice of a Wintel based platform. Table 2 details the target system configuration, together with data on OS and hardware infrastructure.

#### 4.2 Workload

TPC-C is the standard benchmark of the Transaction Processing Performance Council for the evaluation and comparison of the performance and the price/performance in database systems due to OLTP loads. This benchmark models a typical business environment where a large number of users are submitting light to medium-weight business transactions against a database system. The company illustrated by TPC-C is a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. Several transaction types are modeled, ranging from stock level checking, to placement of new orders of assets, delivery, tracking and payment of orders. A detailed description of the implementation of TPC-C is out of the scope of this paper and it can be found in [9]. The implementation used in this work is a standard

Table 2. Target System Configuration

Server	Hardware Platform	Intel Pentium II processor based machines with 128 MB RAM
	Operating System	Windows NT 4.0 Server build 1381 with SP3
	DBMS	Oracle 8 Server Release 8.1.5.0.0 Oracle Net Server 8.1 Dedicated server session (IP-lite)
Client	Intel Pentium based machine with 64 MB RAM WindowsNT 4.0 Workstation with SP3 Oracle Net Client 8.1	
Network	10 Mbit Ethernet with TCP/IP	

**Table 3. Hardware faults model**

Location	Type	Trigger	Duration
CPU registers; {EAX,...,EDX, EBP; ESP, EFlags, EIP, CS, DS, ES, FS, GS, ESI, EDI}	Bit-flip. One bit at a time.	[1..60] seconds. Start of time is first instruction of DBMS process	1 CPU instruction

implementation derived from best practices found in recent full disclosure reports available at the TPC web site. Five concurrent clients were used to submit transactions during one minute per injection run. According to TPC-C rules one checkpoint was made per injection run (here at 30 seconds from workload start time).

### 4.3 Faults

**4.3.1 Hardware transient faults.** Hardware transient faults are emulated by injecting bit flip errors in the processor registers. Bit flip faults have been injected at random concerning both fault locations and fault triggers. At first sight, relying exclusively in the register model may seem too much restrictive. However, it should be stressed that the Pentium processor is far from the RISC "load store" memory access model. In fact, the Intel x86 architecture is rich in instructions with mixed memory and register operands (and compilers use it a lot). This way, it is possible to interfere with memory operands and memory access (address generation included) by manipulating register contents. Table 3 summarizes the hardware fault parameters used in this experimental work. A total of 1078 hardware faults were injected.

**4.3.2 Software faults.** Software faults are injected at run-time via on-the-fly instruction decode. The trigger process works in a similar way to hardware faults but the context corruption procedure is different. Instead of performing a previously defined (at setup time) context corruption, the fault is mimicked according to the target instruction type. A total of 555 software faults were injected. See Table 4 for details (including fault distributions).

Assign faults are injected when the target instruction is a

**Table 4. Software fault and error types**

Fault Type	Target instruction (assembly level)	Error type	Assembly level
Assign	<i>Mov mem imm</i> <i>Mov reg, imm</i>	Missing/ wrong initialization	NOP (50%) Corrupt source operand (50%)
Check	All conditional branch instructions.	Missing check. Off-By-One Negation.	NOP (25%) Off-by-one (50%) Negation (25%)
Overlay	Repeat store string instructions	"Overlay"	ECX corrupted: - Add one (80%) - 2-12 bytes (10%) - 4186 (10%)
Pointer	All load and store instructions.	Corrupt target/ source address	Same as in Assign fault type

**Table 5. Assign and pointer arithmetic faults (error types)**

Operand Corruption type	Distribution	High level error mimicked
Boundary	70%	Add or subtract one
Change-to-zero	10%	Supports NULL pointers (wrong/missing initialization; wrong pointer arithmetic)
Change-to-one	10%	Forces a TRUE condition; wrong/missing initialization and wrong pointer arithmetic
Random	10%	Wrong/missing initialization

move operation from an immediate operand to either a memory or register operand (mimics most of the high level initializations, e.g., the C language construct *int i=0*). The assignment operation is faulted by either replacing the instruction with a NOP, which mimics missing initialization, or by corrupting the source operand (wrong initialization value). The source operand is corrupted with the error "type" distribution shown in Table 5.

Checking and string overlay faults are injected in a similar way to the one described for the assignment faults. The main difference is that the target instructions are instructions related to checking instructions and store string instructions. The distributions for the corruption types are described above in Table 4.

### 4.4 Readouts, errors and failures

Table 6 summarizes the measurements along with its classification in terms of two main criteria: level of abstraction and if they were built-in or developed on-purpose.

Possible machine level exceptions rose within the Oracle process as well as exit code values were saved for each injection run. The exit code is used to conclude whether Oracle had terminated normally or abnormally. By default, on the Win32 platform certain kinds of exceptions (memory access violations and illegal instructions for instance) force the termination of the issuing threads, which in turn might cause the owner process to terminate.

DBVERIFY is described as a database administrator (DBA) helper tool capable of pinpointing database data file corruption [20]. It performs a physical data structure integrity check on an offline data file. Its primary use is

**Table 6 – Error, failure and monitoring techniques**

Name	Abstraction level	Built-In ?
Exception handling	OS	Yes
Exit code	OS	Yes
Oracle internal checks	DBMS	Yes
DBVERIFY	DBMS	Yes
OdBit	DBMS	No
Gold database	DBMS/Application	No
TPC-C consistency tests	Application	No

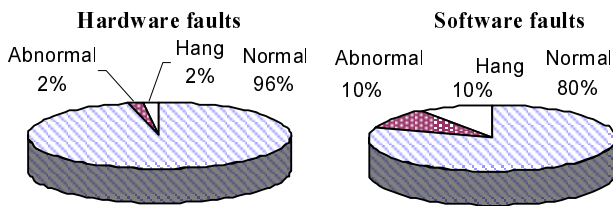


Figure 2. Impact of faults in the Oracle server process

when you need to insure that a backup database (or a data file) is valid before it is restored or as a diagnostic aid when you have encountered data corruption problems. We have used it here to assess if database files (both data and log files) got corrupted.

Oracle performs an extensive logging and error reporting activity [21]. This activity is materialized in several kinds of trace files per database instance. The *alert* file records information about internal errors and administrative activities, such as database instance startup, shutdown, recovery and backups. Oracle also writes additional information about internal errors to the alert file, such as the location and name of any additional trace files generated because of the error. A suite of Perl scripts was developed to scan the trace files and summarize the relevant data for this experimental work. It should be stressed that no special configuration was used for benefit of error logging. While a more detailed time scale would be very convenient to measure internal error detection latency (and Oracle gives that under a special configuration) we decided to keep up with the default configuration in order to avoid interference with typical Oracle execution.

OdBit [22], which stands for *Oracle DataBase Integrity Testing*, performs consistency and integrity checking at the database instance level. The metadata held in a relational DBMS such as Oracle is rich in rules describing the relations between data objects such as tables, views or even PL/SQL procedures [23]. Those rules are maintained by the core database engine to enforce referential integrity and consistency. OdBit extracts those rules from the Oracle data dictionary and checks them against the stored data. For the readers familiar with database technology, data objects checked include synonyms, dependencies, clusters or tables. With some checks, we are simply verifying that a particular

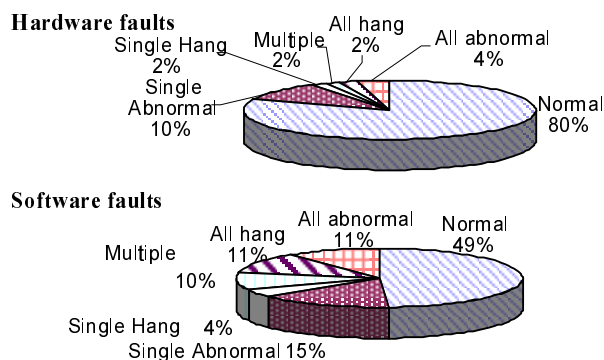


Figure 3. Impact of faults in the TPC-C client

object exists. For instance, if we have a synonym, then the object the synonym refers to must exist. Checking table integrity is a little bit more complex. When addressing a table, several rules are verified, such as check constraints, primary keys, foreign keys or unique constraints.

A *gold database* was also used to support error detection. The *remote terminal emulator* (RTE), part of the TPC-C benchmark, was slightly modified to submit the random transactions to a *gold* database instance as well. This gold database instance was located in a separate machine and, for each injection run, it was kept in the same initial state as the target database. At the end of an injection run the target and the gold databases were compared for mismatches. The comparison is made through the SQL SELECT MINUS construct that is able to pinpoint differences per table and on a record basis.

Last but not least, all consistency tests recommended in the TPC-C specification were implemented. These tests reflect the so-called “business logic”, i.e., rules involving data and business processes that typically are difficult to model in terms of relational constraints. For instance, one of such consistency conditions states that the year-to-date sales volume of orders in a warehouse must be equal to the sum of year-to-date volume values per district.

## 5. Results

We start by presenting and discussing the “broad picture” results. The observations made mimic the ones that are (or might be) made by typical users or system/database administrators when something goes wrong with its applications or systems. Next, it is presented a model of error, failure and recovery in our SUT. The wealth of information available from several sources (Oracle logs, OS logs and Xception database) is condensed in several models by abstraction level and fault type. This perspective on the impact of faults is more useful to give feedback to system designers, since it provides quantitative insights on first errors, how they propagate and what failure types they cause.

We end up by presenting and discussing dependability metrics that can be drawn from this experimental framework and how they can be improved (both of them) to be a component of a dependability benchmark environment.

### 5.1 First insights: a macroscopic vision

Figures 2 and 3 show the macroscopic view of the impact of faults in the Oracle server process and TPC-C workload client processes, respectively. The impact of faults is evaluated through process failure modes, considering normal termination, abnormal termination, and process hang. For the TPC-C client processes, single hang and single abnormal conditions refer to failure modes of a single client process. All abnormal and all hang are limit conditions where all five client processes terminate abnormally or enter in a frozen state. Runs classified as *multiple* experienced other combinations of hanged and aborted clients.

The first evident result is that the software faults have a more severe impact in the system than hardware faults. Another interesting (and surprising) result is that in the large majority of injected hardware faults – 96% of the faults – the Oracle server (apparently) behaves normally. It is worth noting that the platform is a regular Wintel machine with very few error handling mechanisms. A noticeable difference between hardware and software faults is that the latter set is more prone to cause *hang* and *abnormal* failures (10% each class) than the hardware faults (2% each class). Observable impact in the client processes follows the same basic trend observed for the Oracle server process but with some noticeable deviations. The *normal* condition is also dominant but the percentages observed for normal process termination are not so high as for the faults injected in the server process. An interesting observation is that the percentage of Oracle server hangs is basically the same as *all hangs* observed for the clients (i.e., all client processes hang as the consequence of the server failure). However, the same is not true for abnormal termination, as the percentage of *all abnormal* failures observed in the clients is larger than the percentage of abnormal terminations in server.

Figure 4 shows the impact of the faults with the SUT running both TPC-C and the test suite for database error detection (the error detection issue is analyzed in the next section; here we are interested in the global behavior of the system). The test suite includes the data consistency tests specified in the TPC-C benchmark, Oracle internal assertions, execution of the DBVERIFY command, and the execution of the OdBit tool to check the consistency of the data in the TPC-C tables. One interesting (and unexpected) side effect of running this test suite is that the observable behavior changed significantly. In fact, the extra test execution step at the end of the benchmark activated latent errors leading to a substantial increase in the amount of hangs observed, that otherwise would appear as normal termination. In some sense, the test suite represents a second workload. Considering that a typical database server executes SQL commands from different applications (each one is a workload) then the results observed with consistency tests may be more representative of real cases. However, this “workload” is more a stress test since it is particularly targeted at error detection as it scans database structures for inconsistency detection.

## 5.2 Error, Recovery and Failures

The percentages of errors detected by Oracle assertions are shown in Figure 5. For the large majority of faults, the errors remained undetected for the observed period. Several possible circumstances account for this: the error(s) stayed latent, their effects were overwritten or tolerated by normal system execution, or the induced error(s) could not be detected by the available means. The single and multiple error detection patterns for hardware and software faults are quite different. The more aggressive effects of the software

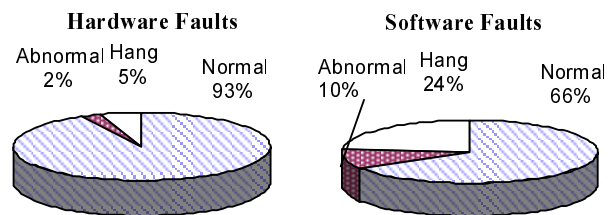


Figure 4. Impact of faults with the test suite

faults result in an higher coverage of multiple errors when compared to the same figure for hardware faults.

One unexpected outcome was the coverage obtained by the running Oracle DBVERIFY utility (see section 4 for details on DBVERIFY). Not a single database corruption was detected for the overall set of injections! This outcome was really surprising because we observed Oracle error messages alerting for data file corruption in nine injection runs and in none of these cases the DBVERIFY detected any error (We checked the effectiveness of the tool by doing direct corruption in the data files and in all cases the tool detected). We see three scenarios that might explain this apparent inconsistency: 1) data file corruption was detected in the in-memory image of the file and its propagation to the disk file was avoided. 2) Oracle incorrectly signaled an error that never really happened, i.e., the fault effects may have changed an error status variable or a checking instruction was affected. 3) The algorithm used by the online corruption checking is different from the offline (DBVERIFY) version. Part of our skepticism on accepting zero detection of data file structural corruption was motivated by common feelings that wild memory access due to pointer/addressing errors might cause such a corruption. Previous work in the RIO project [24] concluded that OS file cache corruption leads to disk file corruption for a significant number of times. However, differences in Oracle managed file cache may justify this behavior.

One very important result is that the execution of the OdBit utility to detect data inconsistencies in the data stored in the TPC-C tables has never detected any error. This means that none of the injected faults (in a total of 526 injected against a SUT with the test suite) caused a committed transaction to produce data that violated the relational constraints in presence. In other words, faults may cause the lost of some uncommitted transactions but all the committed transactions behave according to the transaction model. We have observed that the execution of OdBit often leads to a database hang or internal error detection (and subsequent abnormal termination). This can be seen as

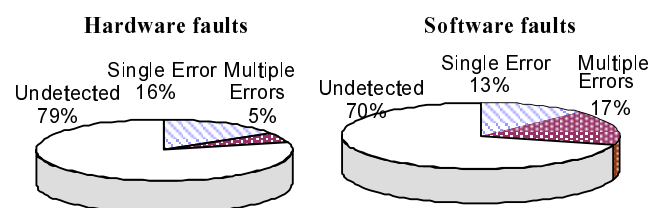


Figure 5. Coverage of Oracle internal consistency checks

OdBit's contribution to the "error activation" behavior already presented in section 5.1.

For 6.4% of injection runs (in a set of 526 runs with the test suite), TPC-C consistency tests were able to detect discrepancies in the database, i.e., the stored data did not verify (at least one of) the consistency clauses. Moreover, TPC-C consistency tests "detected" 0.95% of faults that were not "detected" by any other of the remaining error detection "mechanisms". This is a very interesting figure since it clearly indicates that "business logic" inconsistencies might show up as a result of transient hardware faults or software faults. Furthermore, in around 1% of the cases those inconsistencies might remain unnoticed unless dedicated auditing is performed. Most of the inconsistencies detected seemed to be due to "lost" payment transactions. This is reasonable since payment transactions account for nearly 50% of all transactions submitted.

The comparison between the target and gold databases revealed that in 28.3% of injection runs there was a discrepancy. However only in 4.9 % of runs was a discrepancy observed without any other detection from other error detection mechanisms. These discrepancies reflect the cases where one (or more) transaction(s) are submitted to both the target and gold databases but, due to the fault impact, are only completed in the gold database.

Figure 6 shows detailed error propagation for hardware and software faults, respectively. Rounded-corner rectangle

shapes represent states, arrows are transitions between states and the number next to each arrow is the observed probability for that transaction to occur. Memory access violation is (by a great distance) the dominant error trace observed at the operating system level. Only few of unhandled exceptions of types *STATUS PRIVILEGED INSTRUCTION* and *STATUS INTEGER DIVIDE BY ZERO* were found. These are unsuccessful status codes signaled by the OS kernel that translates the analogous low-level hardware exceptions. This is true for both hardware and software faults.

Surprisingly, the majority of memory access violation conditions did not lead to a premature termination or caused the Oracle to hang. While there is no available data that would easily contest or confirm it, we draw the following hypothetical scenario: First, most of OS exceptions might happen to be well handled internally by Oracle, which would assert a high robustness of the DBMS code. Second, the exceptions might have been rose (eventually) in Oracle server threads that act as application (user) peer entities. Oracle is designed to tolerate user or server thread failures [23]. A background thread, PMON, resolves the failure by rolling back the current transaction of the aborted thread and releasing any resources that this thread was using. Recovery of the failed user or server thread is automatic. However, if the aborted thread is a background thread, the database instance cannot usually continue delivering proper service.

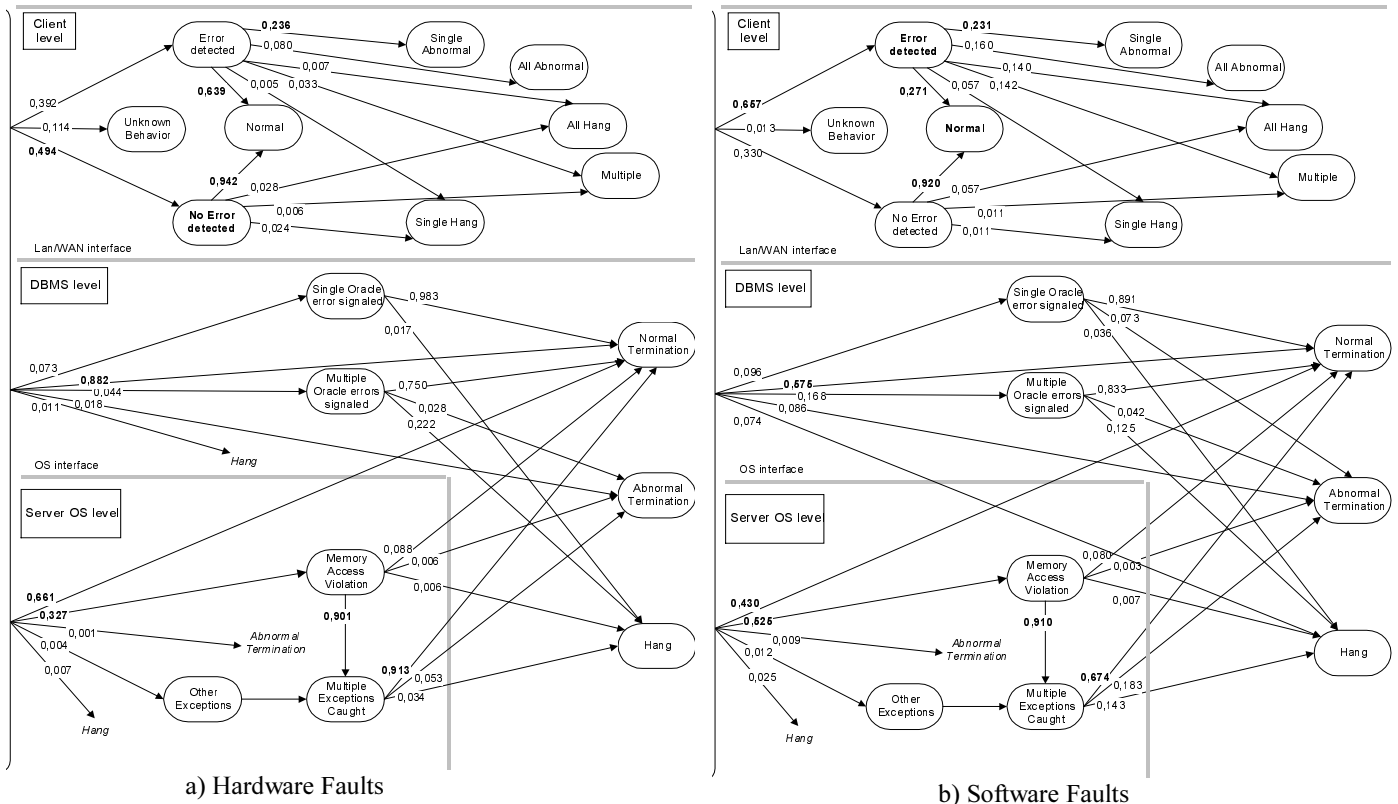


Figure 6. Error propagation model

It is worth mentioning that, concerning memory access violations, software faults behaved in a significantly different way: about one third (32.6%) of the runs with multiple exceptions caught lead to either abnormal or hang behavior while the analogous figure for hardware faults is only around 9%.

### 5.3 Performance degradation and recovery cost

The evaluation of the recovery mechanisms concerning performance should take into account two figures of merit: 1) performance degradation in normal operation due to the presence of the recovery mechanisms and 2) recovery cost, measured as the time needed to recover after a fault. These recovery measures are common ground in the database community. For example, TPC benchmarks include specific clauses to state that the performance measures must be taken with the overhead of the recovery mechanisms (but TPC do not consider benchmarking in the presence of faults yet). However, many DBMS allow a wide range of tuning configurations, affecting both performance in normal operation and the recovery cost. The experiments to assess the recovery measures (and dependability measures) in a wide range of configurations of the setup used in this work are under way (they will take several months). In this section we present the results corresponding to the two configurations evaluated so far: database with the archive log OFF and with the archive log ON.

**Table 7. Performance loss with archiving enabled**

No archive	Archiving log	Average performance loss
378 Tpm	293 Tpm	22.5%

The performance achieved in both configurations (see Table 7) is just the transactions per minute with raw tuning (i.e., we haven't tried to improve speed by doing specific tuning). The performance overhead due to the archive log is 22.5%. Naturally, this figure depends on the speed of the tertiary disk in which the archives are stored (there are several ways to tune the archiving mechanism to reduce this overhead).

The gains in availability due to the archive log cannot be meaningfully measured by injection of transient faults. In fact, the archive log just stores the content of the on-line redo log files in a permanent way. Having the archive log ON is possible to recover the database from any point in the past (since all the redo log entries are preserved) or to recover from total lost of the data files (together with data file backup).

The percentage of injected faults that cause the activation of the recovery mechanisms was 29.5%. The cost of recovery measured in time and number of Oracle data blocks (the size of each block is 8Kbytes) read and written from/to disk are represented in Table 8. As faults in a real environment are rare events, the recovery cost is really non-significant. However, the performance degradation in

normal operation due to the necessary recovery mechanisms is very important.

## 6. Conclusions and Future work

A joint evaluation of performance and dependability in a COTS DBMS done through fault-injection was presented. Errors that emulate transient hardware faults as well as software faults were injected. The system did run with a TPC-C compliant workload that mimics on-line transaction processing requirements available in today client/server computing environments. The impact of faults in the system was evaluated at several abstraction levels.

A macroscopic observation based on the termination of the server and client processes revealed that:

- 1) A large majority of the faults caused no observable abnormal impact in the DBMS. In 96% of hardware faults and 80% of software faults the database server terminated normally after workload execution.
- 2) Software faults are more prone to let the database server hanging or cause abnormal terminations.
- 3) The impact of faults in the TPC-C clients follows the same relative trends observed in the server but malevolent

**Table 8 – Recovery cost**

Recovery time (seconds)		Count of blocks read and written during recovery	
Mean recovery time	7.1	Average blocks read	235
Standard deviation	2.4	Maximum blocks read	878
Maximum recovery time	13.0	Minimum blocks read	1
Minimum recovery time	4.0	Average blocks written	195
Percentage of faults that caused Oracle recovery	29.5%	Maximum blocks written	821
		Minimum blocks written	1

effects are more noticeable; up to 51% of software faults lead to observable failures in the client processes.

4) The insertion of a test suite that executed after the TPC-C workload significantly modified the observable impact. For software faults, the hang failures raised from 10% to 24% while the normal figure dropped from 80% to 66%. This behavior agrees with the well-known workload effect in experimental evaluation, since the test suite can be seen as another (albeit special) workload. The test suite awaked latent errors that otherwise would remain hidden.

A detailed analysis conducted over several sources, ranging from machine level exception handling to business logic in the TPC-C environment, further lead to the following results:

- 1) Memory access violations are the dominant error trace observed at the operating system level, detecting 52.5% and 32.7% of the injected software and hardware faults (respectively).
- 2) Oracle does a good job on recovering from failures due to *invalid memory access* conditions. The majority of errors found were tolerated and ultimately led to normal termination.

3) Oracle internal checks detected errors, due to hardware and software faults, in 21% and 30% of injection runs, respectively. These figures show that by carefully monitoring database trace files, DBAs can achieve a wider understanding on failure status than would be attained by simply monitoring processes up/down state.

4) No (durable) database corruption was detected. Despite the fact that in some injection runs Oracle internal checks signaled data file corruption, off-line tests performed on all data files did not detect any corruption.

5) The relational constraints as well as the internal consistency of the metadata that Oracle uses to describe user data were not affected by the faults injected. In other words, all the committed transactions behaved according to the transaction model.

6) Transient hardware and software faults might however cause errors that are only detected at the "business logic" level, i.e., by inspecting a set of rules that reflect data consistency at a higher abstraction model than the relational one. In 0.95% of injection runs, the induced errors were only detected by the TPC-C consistency tests. This shows that periodic inspection of a COTS DBMS should be made in order to certify its integrity, since apparent "failure" free behavior is not enough to reach a confident judgment.

The tradeoff in terms of performance loss in normal operation to enable no-loss recovery in presence of media faults (archive log ON) was found to be 22.5%. We foresee that this figure might be improved with dedicated database tuning. A significant number of injected faults – 29.5 % - triggered the database recovery mechanism. The MTTR was found to be 7.1 seconds. While this MTTR seems inadequate to fulfill requirements of telecommunications database, for instance, it seems to fit less stringent application areas.

Reasoning on all the results presented, the usage of a COTS DBMS like Oracle seems to be an effective way to benefit from COTS components economics while holding a fair confidence on service reliability. However, in certain business areas where data integrity is a must, customized "business logic" checking should be implemented. In fact, the experimental work here described shows that, due to transient hardware or software faults, there is a chance that data gets corrupted in a very subtle way, so that detection by built-in checking techniques was not possible.

Future work will leverage the evaluation framework to address dependability benchmarking of COTS DBMS. This includes support for new fault models (notably operations faults) and emulation of specific failures that are hard to trigger by fault/error injection, new system workloads and configurations (TPC-D, distributed database, primary and standby database, etc) as well as migrate the evaluation building blocks to other platforms.

## 12. References

[1] J. Gray, "A Census of Tandem Systems Availability Between 1985 and 1990," IEEE Trans. on Reliability, Vol.39, No.4, pp.409-418, Oct., 1990.

[2] M. Kalyanakrishnam, Z. Kalbarczyk, R. Iyer, "Failure Data Analysis of a LAN of Windows NT Based Computers", Proceedings of SRDS'18, October, Switzerland, pp. 178-187, 1999.

[3] M. Sullivan and R. Chillarege, "Comparison of Software Defects in Database Management Systems and Operating Systems", Proceedings of FTCS-22, pp. 475-484, July 1992.

[4] 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.

[5] J. Arlat et al, "Fault Injection and Dependability Evaluation of Fault Tolerant Systems", IEEE Transactions on Computers, Vol. 42, No. 8, pp. 919-923, August 1993.

[6] J. Christmansson, M. Hiller, and M. Rimén, "An Experimental comparison of Fault and Error Injection", Proceedings of ISSRE'98, pp. 369-378, USA, 1998.

[7] H. Madeira, M. Vieira and D. Costa, "On the Emulation of Software Faults by Software Fault Injection," Proc. of Intl. Conference on Dependable Systems and Networks, June 25-28, New York, USA, 2000.

[8] J. Carreira, H. Madeira, J. Silva, "Xception: A technique for the evaluation of dependability in modern Computers", IEEE Transactions on Software Engineering, Vol.24, No.2, pp. 125-136, February 1998.

[9] Transaction Processing Performance Council, "TPC Benchmark C, Standard Specification, Revision 3.3," 1998.

[10] T. Tsai and R. K. Iyer, "An Approach to Benchmarking of Fault-Tolerant Commercial Systems", Proc. of the 26th IEEE Fault Tolerant Computing Symposium, FTCS-26, Sendai, Japan, pp. 314-323, June 1996.

[11] T.J. Teorey and W.T. Ng, "Dependability and Performance Measures for the Database Practitioner", IEEE Transactions on Knowledge and Data Engineering, Vol.10, No. 3, May/June, 1998, pp.499-503.

[12] M. P. Sullivan, "System Support for Software Fault Tolerance in Highly Available Database Management Systems," Ph.D. Thesis, 1992.

[13] W.T. Ng and P. M. Chen, "Integrating Reliable Memory in Databases", In Proceedings of VLDB 1997, pages 76-85, August 1997.

[14] S. Chandra and Peter M. Chen, "How Fail-Stop are Faulty Programs?", FTCS'28, Munich, Germany, 1998, pp. 240-249.

[15] M. Sabaratnam et al., "Evaluating the Effectiveness of Fault Tolerance in Replicated Database Management Systems", Proceedings of FTCS'29, June 15-18, Madison, Wisconsin, 1999, pp. 306-313.

[16] P. Koopman and J. deVale, "Comparing the robustness of POSIX Operating Systems", Proceedings of FTCS'29, June 15-18, Madison, Wisconsin, 1999, pp. 22-29.

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

[18] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults Based on Field Data," presented at FTCS'26, Sendai, Japan, 1996, pp. 304-313.

[19] D. Costa and H. Madeira, "Experimental Assessment of COTS DBMS Robustness under Transient Faults", Proc. of the Pacific Rim Dependable Computing Symposium, 17-18 December, HongKong, China, 1999.

[20] Oracle Corporation, "Oracle8i Utilities Release 8.1.5", Part no. A67792-01, February, 1999.

[21] Oracle Corporation, "Oracle8i Error Messages Release 8.1.5", Part no. A67785-01, February, 1999.

[22] T.R. Silva, D. Costa, H. Madeira, "OdBit: A Tool for Oracle Databases Integrity Testing", Technical Report, CISUC, University of Coimbra, 1998.

[23] Oracle Corporation, "Oracle 8i Server Concepts Manual", 1999.

[24] W. T. Ng and P. M. Chen, "The Systematic Improvement of Fault Tolerance in the Rio File Cache", Proc. of FTCS'29, June 15-18, Madison, Wisconsin, 1999, pp. 306-313.