Benchmarking The Dependability of Windows NT4, 2000 and XP*

Ali Kalakech, Karama Kanoun, Yves Crouzet and Jean Arlat
LAAS-CNRS, 7, Avenue Colonel Roche 31077 Toulouse Cedex 4, France
{kalakech, kanoun, crouzet, arlat}@laas.fr

Abstract

The aim of this paper is to compare the dependability of three operating systems (Windows NT4, Windows 2000 and Windows XP) with respect to erroneous behavior of the application layer. The results show a similar behavior of the three OSs with respect to robustness and a noticeable difference in OS reaction and restart times. They also show that the application state (mainly the hang and abort states) significantly impacts the restart time for the three OSs

1. Introduction

System developers are increasingly resorting to off-the-shelf operating systems (commercial or open source), even in critical application domains. However, any malfunction of the Operating System (OS) may have a strong impact on the dependability of the global system. Therefore, it is important to make information about the OS dependability available, despite the lack of information issued from its development. The current trend is to use dependability benchmarks [1-3].

The aim of an OS dependability benchmark is to objectively characterize the OS behavior in presence of faults. A dependability benchmark is based on experimentation on the OS. Its results are intended i) to characterize qualitatively and quantitatively the OS behavior in the presence of faults and ii) to evaluate performance-related measures in the presence of faults. These results can help in selecting the most appropriate OS, based on the benchmark measures evaluated, in complement to other criteria (e.g., performance, maintenance, etc.).

The work reported here is part of the European project on Dependability Benchmarking, DBench [4, 5], whose objectives are to i) define a framework for designing dependability benchmarks for computer systems and to ii) implement examples of benchmark prototypes. Our previous work, [6], gives the specification of an OS dependability benchmark and presents the experimental framework as well as some preliminary results related to Windows 2000. This paper is aimed at further exploring the portability and suitability of the proposed benchmark by applying it to two other OSs from the same family, namely Windows NT4 and Windows XP professional.

Several relevant attempts have been previously proposed to help characterize the failure modes and robustness of software executives. A comprehensive analysis of the issues linking robustness and dependability can be found in [7]. The executives targeted in these studies encompass real time microkernels and general purpose OSs [1, 8]. The work reported in [9] specifically addressed the robustness of the Win32 application programming interface which is the case of our experiments.

The remainder of the paper is organized as follows. Section 2 summarizes the benchmark and describes a particular prototype for Windows family. Section 3 presents comparison results obtained using this prototype. Section 4 concludes the paper.

2. OS Dependability Benchmark Summary

A dependability benchmark should define clearly: i) the benchmarking context, ii) the benchmark measures and measurements to be performed on the system for obtaining them, iii) the benchmark execution profile to be used and iv) the set-up and related implementation issues required for running a benchmark prototype.

The benchmark results can be meaningful, useful and interpretable only if all these items are provided with the results. The detailed definition of these items, related to the OS benchmark used in this paper are given in [6]. They are summarized hereafter to allow understanding of the results presented in this paper.

2.1. Benchmarking Context

The benchmark target corresponds to an OS with the minimum set of device drivers necessary to run the OS under the benchmark execution profile. The three OS targets are Windows NT4 with Service Pack 6, Windows 2000 Professional with Service Pack 4 and Windows XP Professional with Service Pack 1. All the experiments have been run on the same platform, composed of an Intel

* This work is partially supported by the European Commission - IST DBench project (IST-2000-25425)

Pentium III Processor, 800 MHz, and a memory of 512 MB. The hard disk is 18 GB, ULTRA 160 SCSI.

Our dependability benchmark is a robustness benchmark. Robustness can be viewed as OS capacity to resist/react to faults induced by the applications running on top of it, or originating from the hardware layer or from device drivers.

We emphasize in this work the OS robustness as regards possible erroneous inputs provided by the application software to the OS via the Application Programming Interface (API). We mainly consider corrupted parameters in system calls. For the sake of conciseness, such erroneous inputs are shortly referred to as *faults*. Results concerning the robustness with respect to faults in device drivers can be found in [10-12].

The benchmark addresses the user perspective, i.e., it is primarily intended to be performed by (and to be useful for) someone or an entity who has no in depth knowledge about the OS and whose aim is to significantly improve her/his knowledge about its behavior in presence of faults. In practice, the user may well be the developer or the integrator of a system including the OS. The OS is considered as a "black box" and the source code does not need to be available. The only required information is the description of the services provided by the OS and the description of the OS in terms of system calls.

2.2. Benchmark Measures

Corrupted system calls are provided to the OS through the Win32 environment subsystem, as the three considered OSs cannot run without it [13]. Win32 is thus the API considered in our current benchmark environment.

The OS behavior is characterized by the various outcomes at the API level, while the impact of OS on the application behavior is observed at the workload level. After execution of a corrupted system call, the OS is in one of the states defined in Table 1.

Table 1: OS outcomes

SEr	An *error code* is returned
SXp	An *exception* is raised, processed and notified to the application
SPc	*Panic* state
SHg	*Hang* state
SNS	None of the above situations is observed (*No-signaling* state)

The ***OS Robustness Measure*** is defined as the percentage of experiments leading to any of the outcomes listed in Table 1.

Reaction Time (Texec) corresponds to the mean time necessary for the OS to respond to a system call in presence of faults, either by signaling an exception or by issuing an error code or by executing the required instructions.

Restart Time (Tres) corresponds to the mean time necessary for the OS to restart after the execution of the workload in the presence of faults.

Texec and *Tres* are also observed in absence of faults, for comparison purpose. They are respectively denoted $\tau exec$ and τres.

The benchmark temporal measures are primarily evaluated as a mean time over all experiments categorized by a specific outcome. However, standard deviation is of prime interest as well. Table 2 recapitulates these temporal measures.

Table 2: OS temporal measures

τexec	Time for the OS to execute a system call in absence of faults
Texec	Time for the OS to execute a system call in presence of faults
τres	Duration of OS restart in absence of faults
Tres	Duration of OS restart in presence of faults

The workload is characterized by one of the following outcomes: i) the workload completes with correct results, ii) it completes with erroneous results and iii) the workload is aborted or hangs. Clearly, the workload can end up in any of the three states irrespective of the outcomes of the OS. Conversely, whenever the OS is in the *Panic* state, this can only lead the workload to abort or hang, while an OS *Hang* necessarily leads the workload to hang. In [6], we have detailed all possible combined outcomes and defined a set of measures characterizing the OS taking into account the workload states. In this paper, we mainly use information on the workload final states to examine the impact of the workload state on system restart time.

2.3. Benchmark Execution Profile

In the case of performance benchmarks, the benchmark execution profile is simply a workload that is as representative as possible for the system under test. For dependability benchmarks, the execution profile includes in addition corrupted parameters in system calls. The set of corrupted parameters is referred to as the faultload.

From a practical point of view, the faultload can be either integrated within the workload (i.e., the faults are embedded in the program being executed) or provided in a separate module. For enhanced flexibility, we made the latter choice: the workload and the faultload are implemented separately.

The prototype we have developed uses a TPC-C client [14] as a privileged workload to be in conformance with the experiments performed on transactional systems in DBench [15, 16]. We simply use the TPC-C client as a workload, but we do not use the performance measures specified by TPC-C as they are far from being suitable to characterize the behavior of an OS.

The *faultload* is defined by: i) the technique used for corrupting the system call parameters and ii) the set of system calls to be faulted.

Parameter Corruption Technique: We use a parameter corruption technique similar to the one used in [17], relying on thorough analysis of system call parameters to define *selective substitutions* to be applied to these parameters. A parameter is either a data element or an address. The value of a data can be substituted either by an *out-of-range* value or by an *incorrect* (but not out-of-range) value, while an address can be substituted by an *incorrect* (but existing) address (containing usually an incorrect or out-of-range data). We have used a mix of these three corruption techniques.

System Calls Corrupted: Ideally, and without any time limitation, all system calls used in the workload with parameters should be corrupted. For small workloads this might be possible. However, for workloads such as TPC-C client (that involves more than 130 system calls, with several occurrences in the program), this would require several weeks of experimentation. In addition, all system calls are not necessarily interesting to be corrupted. Indeed, using a fully automated benchmark set up, an experiment lasts 5 minutes on average and, roughly speaking, about 1400 experiments can be achieved in 5 days. This leads to consider 30 to 60 system calls to be corrupted for a 5-day fully automated benchmark execution. Accordingly, we have targeted system calls related to the following components: *Processes and Threads*, *File Input/Output*, *Memory Management* and *Configuration Manager*. Thus 28 system calls have been targeted, for which 75 parameters have been corrupted leading to 552 corrupted values, hence to 552 experiments using the benchmark experimental set-up presented hereafter.

2.4. Benchmark Set-up

Since perturbing the operating system may lead the OS to hang, a remote machine is required to reliably control the benchmark experiments. This machine is called the benchmark controller. Accordingly, for running an OS dependability benchmark we need at least two computers: i) the Target Machine for hosting the benchmarked OS and the workload, and ii) the Benchmark Controller that is primarily in charge of diagnosing and collecting data in case of a hang or an abort. Furthermore, as we are using a TPC-C client as workload, the (Oracle) Data Base Management System (DBMS), that processes the TPC-C client requests, can be installed on the benchmark controller or on another machine. Accordingly, Figure 1 illustrates the various components that characterize the proposed OS dependability benchmark prototype, for Windows 2000. The same set-up is used for the three OS targets, only the benchmark target is changed.

To intercept the Win32 functions (i.e., system calls), we have modified the "Detours" tool [18], a library for intercepting arbitrary Win32 functions on x86 machines. This modification was made to facilitate their replacement by substitution values. Also, we have added several modules in the library to observe the reactions of the OS after parameter substitution, and to retrieve the required measurements.

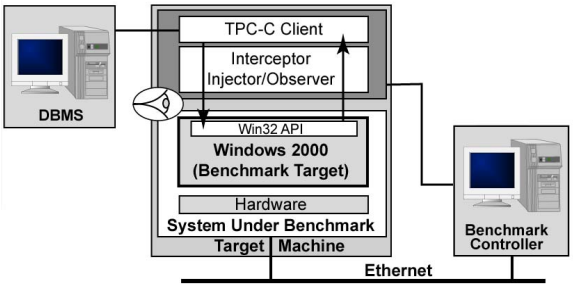

Figure 1. Experimental set-up

The experiment steps are illustrated in Figure 2 in case of workload completion. In case of workload abort/hang state, the end of the experiment is provided by a watchdog timeout. As the average time necessary for the OS to execute the TPC-C client is about 70 seconds when no faultload is applied, the timeout is of 5 minutes.

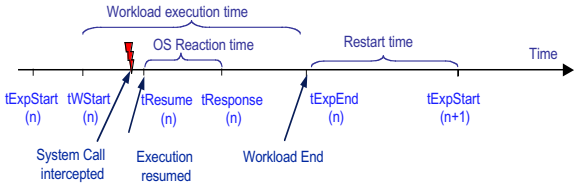

Figure 2. Benchmark execution sequence

2.5. Benchmarking Time

The benchmarking time corresponds to the benchmark implementation time and to the benchmark execution time.

The implementation of the benchmark itself was not very time consuming.

- The TPC-C client implementation used in the current set up is the same as the one used by other DBench partners (see e.g., [19]). The installation took three days.
- The implementation of the different components of the controller took about 10 days.
- The implementation of the faultload took one week, during which we have i) defined the set of the values related to the 28 system call with their 75 parameters to be corrupted and ii) created the database of the corrupted values.

The duration of an experiment with workload completion is less than 3 minutes (including the time to workload completion and the restart time), while it is about 7 minutes without workload completion (including the workload watchdog timeout of 5 minutes and the restart time). Thus, on average, an experiment lasts less

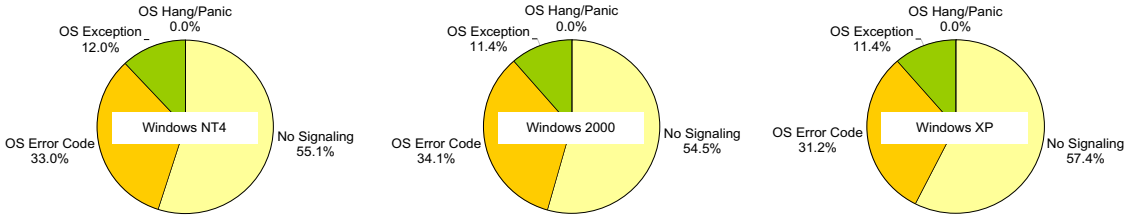

Figure 3: OS Robustness measure using a mix of the three corruption techniques

than 5 minutes. These experiments are fully automated and the whole benchmark execution duration (552 experiments for each OS) is thus about 46h for each OS.

3. Comparison of the three OSs

The benchmark defined in the previous section is used to compare the behavior of Windows NT4, 2000 and XP. We first evaluate the three benchmark measures defined in Section 2.2 (robustness, reaction time and restart time). These measures give information on the global behavior of the OSs. We will then show how they can be refined and complemented by making sensitivity analyses taking into account the workload states after execution of a corrupted system call.

3.1. Benchmark Measures

The robustness measures are given in Figure 3. No panic and hang states were observed for the three OSs. Exceptions have been notified in 11.4 % to 12 % of the cases, while the number of experiments with error code return varies between 31.2 % and 34.1 %. More than half of the experiments led to a No signaling outcome.
Figure 3 shows a *similar behavior* for the three OSs with respect to robustness. Sensitivity analyses with respect to the faultload selection is performed in Section 3.2.1.

The system reaction time in absence of faults, $\tau exec$, is evaluated as the mean reaction time of the 28 selected system calls whose parameters are being corrupted for the experiments. Table 3 shows that, in absence of faults, the three OS have different reaction times.

Table 3: OS reaction time

	Windows NT4		Windows 2000		Windows XP	
	Mean	SD	Mean	SD	Mean	SD
$\tau exec$	344 µs		1782 µs		111 µs	
Texec	128 µs	230 µs	1241 µs	3359 µs	114 µs	176 µs

The OS reaction time in the presence of faults, Texec, corresponds to the mean reaction time of the selected 28 system calls. Table 3 shows that the shortest time is obtained for Windows XP while the longest one corresponds to Windows 2000. For Windows XP, this time is slightly longer than the reaction time in absence of faults while it is significantly lower for the two others. This may be explained by the fact that in about 45% of cases the OS detects the injected fault. It does not execute the faulted system call and returns an error code or signals an exception. The standard deviation (SD) is significantly longer than the mean for the three OSs. Section 3.2.2 will provide more detailed information to explain the various behaviors.

The system restart time is given in Table 4 which shows that Windows XP restart time is 70% of that of Windows 2000, without faults and 73% of this time in the presence of faults. For all systems, the restart time is only few seconds longer than without faults.

Table 4: System restart time

	Windows NT4		Windows 2000		Windows XP	
	Mean	SD	Mean	SD.	Mean	SD
τres	92 s		105 s		74 s	
Tres	96 s	4 s	109 s	8 s	80 s	8 s

Summary
The above results show that the Windows NT4, Windows 2000 and Windows XP kernels have equivalent robustness. This is not surprising as the three OSs are from the same family. They also show that Windows XP has the shortest system call execution time as well as the shortest restart times, both with and without faults. These results do not contradict well-known information about Windows XP's behavior in absence of faults. They confirm that they also hold in the presence of faults.

3.2. Measure Refinement

We will consider successively the three benchmark measures and show how they can be enriched by examining additional information that can be provided by the current benchmark set up.

3.2.1. Robustness Measure. The faultload used in the previous section includes a mix of the three corruption techniques presented in Section 2.3: i) out-of-range data (or out of the boundaries of accepted parameter values), ii) incorrect data (but within the boundaries of accepted parameter values) and iii) incorrect addresses. In total 552 corrupted values for the 75 parameters related to the

28 selected system calls. This faultload is referred to as *FL0*.

It can be argued that incorrect data is not representative of application faults that should be detected by the OS. In order to analyze its impact on the benchmark results, we have considered a reduced faultload *FL1* including only out-of-range data and incorrect addresses. Thus FL1 is composed of 325 corrupted values. The comparison shows that even though the robustness of each OS has been slightly affected by the corruption technique used, the three OSs have still very similar robustness.

Incorrect addresses usually point to out-of range or incorrect data. Taking a pessimistic view, let us assume that they only point to incorrect data and could be discarded as in FL1. We have thus considered a faultload, *FL2*, comprising only out-of-range data (composed of 113 corrupted values). Comparison also shows that using FL2 leads to similar robustness of the three OSs.

This latter result encourages corruption of the parameters of all system calls involved in the workload using only the out-of-range technique, without increasing significantly the benchmark run duration. We have thus considered a faultload, *FL3*, composed of only out-of-range data, targeting all of the 132 system calls with their 353 parameters. 468 experiments have been performed for each OS. The results show that the three OSs still have *similar robustness*, when corrupting all system calls involved in TPC-C client workload.

The faultloads considered are summarized in Table 5. FL0 to FL3 use a selective substitution technique.

Table 5: Faultloads considered

	Incorrect data	Incorrect address	Out-of-range data	Bit-Flip	# System calls	# exp
FL0	x	x	x		28	552
FL1		x	x		28	325
FL2			x		28	113
FL3			x		All (132)	468
FL4				x	28	2400

The sensitivity of the robustness measure to the parameter corruption technique can be further analyzed, using a bit-flip parameter corruption technique, referred to as *FL4*. We use it here to corrupt the same set of 75 parameters in a systematic way (i.e., flipping the 32 bits of each parameter considered). This leads to 2400 corrupted values (i.e., 2400 experiments). The results are given in Figure 4 for Windows 2000. This figure shows that the OS robustness is very similar using the two parameter corruption techniques, which confirms our previous work on fault representativeness [20].

We conclude that the results obtained for a subset of system calls related to the most frequently used functions of Windows (corresponding to *Processes and Threads*, *File Input/Output*, *Memory Management* and *Configuration Manager*) are similar to those obtained when considering all system calls. This is why we have targeted these four functions for the Windows family.

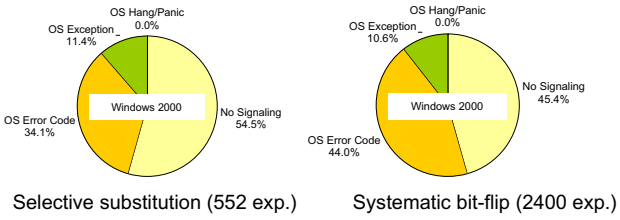

Figure 4: Sensitivity to corruption technique

3.2.2. OS Reaction Time. Table 6 completes the information provided in Table 3. It gives the OS reaction time with respect to OS outcomes after execution of a corrupted system call. It can be seen that i) the time to issue an error code is very short and comparable for the three systems, ii) the time to signal an exception is higher than that of error code return, but it is still acceptable for Windows NT4 and XP, but very large for Windows 2000 and iii) the largest execution time is obtained when the OS does not signal the error (SNS).

Table 6: Detailed OS reaction times

	Windows NT4		Windows 2000		Windows XP	
	Mean	SD	Mean	SD	Mean	SD
Error code	17 µs	18 µs	22 µs	28 µs	23 µs	17 µs
Exception	86 µs	138 µs	973 µs	2978 µs	108 µs	162 µs
No signaling	203 µs	281 µs	2013 µs	4147 µs	165 µs	204 µs

The very high standard deviation (SD) is due to a large variation around the mean. As an example, Figure 4 shows this variation in the case of SNS. This figure identifies the system calls that led to SNS with the mean execution time of each of them. The large standard deviation is mainly due to two system calls.

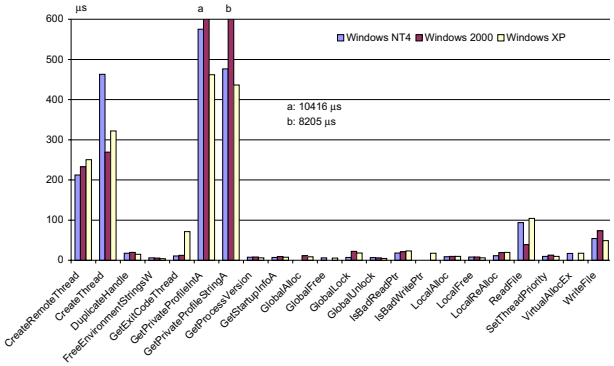

Figure 5: OS reaction time in case of SNS

3.2.3 OS Restart Time. Careful analysis of the collected data revealed a correlation between the system restart time

and the state of the workload. When the workload is completed, the mean restart time is very close to τres (obtained without fault injection), and when the workload is aborted or hangs, the restart time is 8% to 18% higher. Indeed, the number of experiments that led to workload abort/hang was respectively 101, 107 and 128 for Windows NT4, 2000 and XP. Even though Windows XP had induced more workload abort/hang outcomes, it still has the lowest system restart time as indicated in Table 7. The latter gives in rows 1 and 2 the restart times without faults, τres, and in presence of faults, Tres, and refines in the last two rows Tres according to the workload state, irrespective of the OS outcome.

Table 7: Restart time and workload state

	Wind. NT4	Wind. 2000	Wind. XP
τres	92 s	105 s	74 s
Tres	96 s	109 s	80 s
Tres after WL completion	95 s	106 s	76 s
Tres after WL abort/hang	102 s	123 s	90 s

4. Conclusion

In this paper we have briefly presented a dependability benchmark for OSs and an example of implementation prototype, then we have used the prototype to benchmark Windows NT4, 2000 and XP.

The benchmark addresses the user perspective. The OS is considered as a black box and the only required information is its description in terms of services and functions (system calls). We emphasize the OS robustness as regards application induced erroneous behavior.

The comparison of the three OSs showed that i) they are equivalent from the robustness point of view and that ii) Windows XP has the shortest reaction and restart times. Detailed information provided by the current benchmark prototype allowed refinement of the benchmark measures and confirmed the benchmark measure results. Sensitivity analyses with respect to the parameter corruption technique showed that, even though for each OS the robustness is slightly impacted by the technique used, the three OSs are impacted similarly.

Finally, the results obtained showed that using a reduced set of experiments (113) targeting only out-of-range data led to results similar to those obtained from the 552 initial experiments targeting additionally incorrect data and addresses. If this is confirmed for other OS families, this would divide the benchmark execution duration (that is proportional to the number of experiments) by almost 5, which is substantial.

References

[1] 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, 1996, pp. 314-323.

[2] A. Brown, "Availability Benchmarking of a Database System", EECS Computer Science Division, University of California at Berkley, 2002.

[3] J. Zhu, J. Mauro and I. Pramanick, "R3 - A Framwork for Availability Benchmarking", in *Int. Conf. on Dependable Systems and Networks (DSN 2003),* San Francisco, CA, USA, 2003, pp. B-86-87.

[4] K. Kanoun, J. Arlat, D. J. G. Costa, M. Dal Cin, P. Gil, J.-C. Laprie, H. Madeira and N. Suri, "DBench – Dependability Benchmarking", in *Supplement of the 2001 Int. Conf. on Dependable Systems and Networks (DSN-2001),* Göteborg, Sweden, 2001, pp. D.12-15.

[5] K. Kanoun, H. Madeira, Y. Crouzet, M. Dal Cin, F. Moreira and Ruiz J.-C, "DBench Dependability Benchmarks", LAAS-report no. 04-120, 2004.

[6] A. Kalakech, T. Jarboui, A. Arlat, Y. Crouzet and K. Kanoun, "Benchmarking Operating Systems Dependability: Windows as a Case Study", in *2004 Pacific Rim International Symposium on Dependable Computing (PRDC 2004),* Papeete, Polynesia, 2004, pp. 262-271.

[7] A. Mukherjee and D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Transactions of Software Engineering*, vol. 23, no. 6, pp. 366-378, 1997.

[8] 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, 1999, pp. 30-37.

[9] C. Shelton, P. Koopman and K. Devale, "Robustness Testing of the Microsoft Win32 API", in*. Int. Conference on Dependable Systems and Networks (DSN'2000),* New York, NY, USA, 2000, pp. 261-270.

[10] J. Durães and H. Madeira, "Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation", in *2002 Pacific Rim Int. Sym. on Dependable Computing,* Tsukuba City, Ibaraki, Japan, 2002, pp. 201-209.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler, "An Empirical Study of Operating Systems Errors", in *Proc. 18th ACM Symp. on Operating Systems Principles (SOSP-2001),* Banff, AL, Canada, 2001, pp. 73-88.

[12] A. Albinet, J. Arlat and J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel", in *Int. Conf. on Dependable Systems and Networks (DSN 2004),* Florence, Italy, 2004.

[13] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000, Third Edition*, 2000.

[14] TPC-C, *TPC Benchmark C, Standard Specification 5.1, available at http://www.tpc.org/tpcc/.* 2002.

[15] M. Vieira and H. Madeira, "Definition of Faultloads Based on Operator Faults for DBMS Recovery Benchmarking", in *2002 Pacific Rim International Symposium on Dependable Computing,* Tsukuba city, Ibaraki, Japan, 2002.

[16] K. Buchacker, M. Dal Cin, H. J. Höxer, R. Karch, V. Sieh and O. Tschäche, "Reproducible Dependability Benchmarking Experiments Based on Unambiguous Benchmark Setup Descriptions", in *Int. Conf. on Dependable Systems and Networks,* San Francisco, Ca, 2003, pp. 469-478.

[17] P. J. Koopman, J. Sung, C. Dingman, D. P. Siewiorek and T. Marz, "Comparing Operating Systems using Robustness Benchmarks", in *Proc. 16th Int. Symp. on Reliable Distributed Systems (SRDS-16),* Durham, NC, USA, 1997, pp. 72-79.

[18] G. Hunt and D. Brubaher, "Detours: Binary Interception of Win32 Functions", in *3rd USENIX Windows NT Symposium,* Seattle, Washington, USA, 1999, pp. 135-144.

[19] M. Vieira and H. Madeira, "A Dependability Benchmark for OLTP Application Environments", in *29th Int. Conference on Very Large Data Bases (VLDB 2003),* Berlin, Germany, 2003, pp. 742-753.

[20] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun and T. Marteau, "Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study", in*. 2002 Pacific Rim Int. Symposium on Dependable Computing (PRDC 2002),* Tsukuba, Japan, 2002, pp. 51-58.