Dependability Benchmarks for Operating Systems

Karama Kanoun, Yves Crouzet

▶ To cite this version:

Karama Kanoun, Yves Crouzet. Dependability Benchmarks for Operating Systems. International Journal of Performability Engineering, 2006, 2 (3), pp.275 - 287. ⟨hal-01979785⟩

HAL Id: hal-01979785
https://hal.science/hal-01979785v1

Submitted on 13 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

International Journal of Performability Engineering, Vol. 2, No. 3, July 2006, 275-287

Dependability Benchmarks for Operating Systems

KARAMA KANOUN and YVES CROUZET

LAAS-CNRS, 7, avenue du Colonel Roche, 31077, Toulouse Cedex 4 - FRANCE

Abstract: Dependability evaluation is playing an increasing role in system and software engineering together with performance evaluation. Performance benchmarks are widely used to evaluate system performance while dependability benchmarks are hardly emerging. A dependability benchmark for operating systems is intended to objectively characterize the operating system's behavior in the presence of faults, through dependability and performance-related measures, obtained by means of controlled experiments. This paper presents a dependability benchmark for general-purpose operating systems and its application to three versions of Windows operating system and four versions of Linux operating system. The benchmark measures are: operating system robustness (as regards possible erroneous inputs provided by the application software to the operating system via the application programming interface), operating system reaction and restart times in the presence of faults. The workload is JVM (Java Virtual Machine), a software layer, on top of the operating system allowing applications in Java language to be platform independent.

Key Words: *dependability, robustness, benchmark, operating systems, experimentation*

1. Introduction

Software dependability is usually evaluated based on data related to failures and corrections, observed on the software under development or during its operational life. However, when considering Off-The-Shelf software systems (which is the case of operating systems, OSs), most of the time no dependability data is available from their development. Only data collected during operation (if available) can be used to evaluate their dependability, which may be too late for selecting the right OS for building a new computer system based on an OS. In which case controlled experiments are of great help. The latter can either be carried out case-by-case (i,e., *ad hoc* way) or in a well-structured and standardized way, in order to characterize objectively the system behavior in the presence of faults. This is the aim of dependability benchmarks. Benchmarking the dependability of a system consists in evaluating dependability or performance-related measures, experimentally or based on experimentation and modeling, in a standard way. To be meaningful, a benchmark must satisfy a set of properties (e.g., representativeness, reproducibility, repeatability, portability, cost effectiveness). These properties must be taken into consideration from the earliest phases of the benchmark specification.

Our dependability benchmark is a robustness benchmark. Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. Robustness of OS can be viewed as its 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 address here the OS robustness as regards possible erroneous inputs provided by the application software to the OS via the Application Programming Interface (API). More explicitly, we consider corrupted parameters in system calls, shortly referred to as *faults*.

The benchmark presented in this paper is based on JVM (Java Virtual Machine), a software layer on top of the OS, allowing applications in Java language to be platform independent. It is applied to three Windows and four Linux OSs. The main concepts of the benchmark have been developed within the European project on Dependability Benchmarking, DBench [1].

The set of JVM dependability benchmarks is to the third set of OS benchmarks we have built up for Windows and Linux, based on the same high-level specification of the benchmark, using different workloads. The two previous ones used TPC-C Client performance benchmark for transactional systems [2] and PostMark, a file system performance benchmark [3]. Sensitivity analyses of the results with respect to the OS family benchmarked, the workload used and the faultload applied helped us to gain progressively confidence in the benchmark specification and in the results obtained.

The work reported in [4] is the most similar to ours, it addressed the "non-robustness" of the POSIX and Win32 APIs. Pioneer work on robustness benchmarking is published in [5]. Since then, a few studies have addressed OS dependability benchmarks, considering real time microkernels [6-8] or general purpose OSs [9, 10]. Robustness with respect to faults in device drivers is addressed in [11-13].

The remainder of the paper is organized as follows. Section 2 gives the specification of our OS benchmark. Section 3 is devoted to the benchmark implementation for Windows and Linux families. Section 4 presents benchmark results. Section 5 addresses benchmark properties and Section 6 concludes the paper.

2. Specification of the Benchmark

A dependability benchmark is specified through the definition of i) the benchmark target, ii) measures to be evaluated, iii) benchmark execution profile to be used to activate the operating system, iv) guidelines for conducting benchmark experiments and implementing the benchmark. The benchmark results are meaningful, useful and interpretable only if all the above items are supplied together with the results.

2.1. Benchmarking Target

An OS is a generic software layer providing basic services to the applications through the API, and communication with peripherals devices via device drivers. The *benchmark target* corresponds to the OS with the minimum set of device drivers necessary to run the OS under the benchmark execution profile. However, the benchmark target runs on a hardware platform whose characteristics impact the results. Thus, all benchmarks must be performed on the same hardware platform.

Although, in practice, the benchmark measures characterize the target system and the hardware platform, we state simply that the benchmark results characterize the OS.

Our benchmark addresses the user perspective, i.e., it is intended to be performed by (and to be useful for) someone who has no thorough knowledge about the OS and whose aim is to improve her/his knowledge about its behavior in the presence of faults. In practice, the user may well be the developer or the integrator of a system including the OS.

As a consequence, the OS is considered as a "black box". The only required information is its description in terms of system calls and in terms of services provided.

2.2. Benchmark Measures

The OS receives a corrupted system call. After execution of such a call, the OS is in one of the following states:

SEr (Error code): the OS generates an error code that is delivered to the application.
SXp (Exception): in the user mode, the OS processes the exception and notifies the application. However, for some critical situations, the OS aborts the application. In the kernel mode an exception is automatically followed by a *panic* state (e.g., blue screen for Windows and oops messages for Linux). Hence, the latter exceptions are included in the panic state and the term exception refers only to the first case of user mode exception.
SPc (panic): the OS is still "alive" but it is not servicing the application. In some cases, a soft reboot is sufficient to restart the system.
SHg (Hang): a hard reboot of the OS is required.
SNS (No Signaling): the OS does not detect the erroneous parameter and executes the erroneous system call. SNS is presumed when none of the previous situations (SEr, SXp, SPc, SHg) is observed.
Panic and *hang* situations (SPc, SHg) are actual states in which the OS can stay for a while. SEr and SXp characterize events. They are easily identified when the OS provides an error code or notifies an exception.
The benchmark measures include a robustness measure and two temporal measures.
OS Robustness (POS) is defined as the percentages of experiments leading to any of the states listed above. POS is thus a vector composed of 5 elements.
Reaction Time (Treac) corresponds to the average time necessary for the OS to respond to a system call in presence of faults, either by notifying an exception or by returning an error code or by executing the required instructions.
Restart Time (Tres) corresponds to the average time necessary for the OS to restart after the execution of the workload in the presence of faults. Although under nominal operation the OS restart time is almost deterministic, it may be impacted by the corrupted system call. The OS might need additional time to make the necessary checks and recovery actions, depending on the impact of the fault applied.
The OS *reaction time* and *restart time* are also evaluated by experimentation in absence of faults for comparison purposes. They are respectively denoted $\tau reac$ and τres.

2.3. Benchmark Execution Profile

For performance benchmarks, the benchmark execution profile is a workload that is as realistic and representative as possible for the system under benchmarking. For a dependability benchmark, the execution profile includes, in addition to the workload, a set of faults, referred to as the faultload.

In the current benchmark, the workload is JVM, solicited through a program allowing to display «*Hello World*» on the screen. This program activates 76 system calls for Windows family and 31 to 37 system calls for Linux Family.

The faultload consists of corrupted parameters of system calls. For Windows, system calls are provided to the OS through the Win32 environment subsystem. In Linux OSs, these system calls are provided to the OS via the POSIX API. During runtime, the system calls activated by the workload are intercepted, corrupted and re-inserted.

The parameter corruption technique relies on thorough analyses of system call parameters to define *selective substitutions* to be applied to these parameters (similarly to the one used in [14]). A parameter is either a data 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 is substituted by an *incorrect* (but existing) address (that could contain an incorrect or out-of-range data). We use a mix of these three techniques. More details can be found in [3].

2.4. Benchmark Conduct

Since disturbing the operating system may lead the OS to hang, a remote machine, referred to as the benchmark controller, is required to control the benchmark experiments, mainly in case of OS Hang or Panic states or workload hang or abort states (that cannot be reported by the machine hosting the benchmark target). Hence, we need at least two computers as shown in Figure 1. The *Target Machine* hosts the benchmarked OS and the workload, and ii) the *Benchmark Controller* is in charge of diagnosing and collecting part of benchmark data.

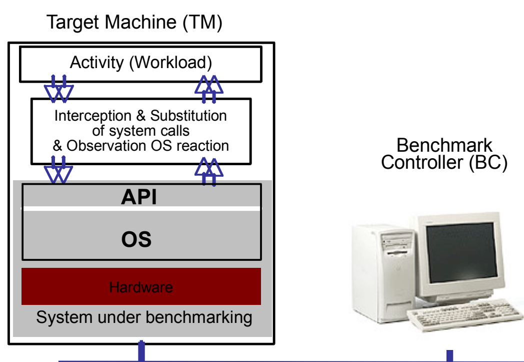

Fig. 1: Benchmark Environment

The two machines perform the following: i) restart of the system before each experiment and launch of the workload, ii) interception of system calls with parameters, ii) corruption of system call parameters, iii) re-insertion of corrupted system calls, vi) observation and collection of OS states. The experiment steps in case of workload completion are illustrated in Figure 2 and will be detailed in the next section. In case of workload non-completion state (i.e., the workload is in abort or hang state), the end of the experiment is governed by a watchdog timeout, fixed to 3 times the workload execution time without faults.

3. Benchmark Implementation

3.1. Prototype

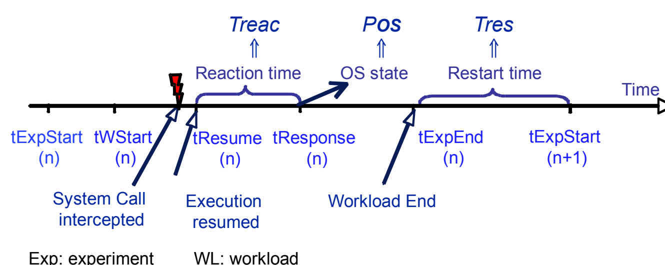

Fig. 2: Benchmark Execution Sequence in Case of Workload Completion

In order to obtain comparable results, all the experiments are run on the same target machine, composed of an Intel Pentium III Processor, 800 MHz, and a memory of 512 Mega Bytes. The hard disk is 18 Giga Bytes, ULTRA 160 SCSI. The benchmark controller in both prototypes for Windows and Linux is a Sun Microsystems workstation.

To intercept Win32 functions, we use the Detours tool [15], a library for intercepting arbitrary Win32 binary functions on X86 machines. We added three modules for i) substituting parameters of system calls by corrupted values ii) observing the reactions of the OS after execution of a corrupted system call, and iii) collecting the required measurements.

To intercept POSIX system calls, we used another interception tool, Strace [16] to which we added modules similar to those added to Detours.

3.2. Benchmark Preparation

Before each benchmark run (i. e., execution of the series of experiments related to a given OS), the target kernel is installed, and the interceptor is compiled for the current kernel (interceptors are kernel-dependent both for Windows and Linux). Once the benchmarking tool is compiled, it is used to identify the set of system calls activated by the workload. Parameters of these system calls are then analyzed and a database of corrupted values is built accordingly.

3.3. Benchmark Execution

At the beginning of each experiment, the target machine (TM) records the experiment start instant $t_{ExpStart}$ and sends it to the benchmark controller (BC) along with a notification of experiment start-up. The workload starts its execution. The Observer module records, in the experiment execution trace, the start-up instant of the workload, t_{WStart}, the activated system calls and their responses. This trace also collects the relevant data concerning states SEr, SXp and SNS. The recorded trace is sent to the BC at the beginning of the next experiment.

The parameter substitution module identifies the system call to be corrupted. The execution is then interrupted, a parameter value is substituted and the execution is resumed with the corrupted parameter value (t_{Resume} is saved in the experiment execution trace). The state of the OS is monitored so as to diagnose SEr, SXp, SNS. The

corresponding OS response time ($t_{Response}$) is recorded in the experiment execution trace. For each run, the OS reaction time after the experiment is calculated as the difference between $t_{Response}$ and t_{Resume}. At the end of the execution of the workload, the OS notifies the end of the experiment to the BC by sending an end signal along with the experiment end instant, t_{ExpEnd}. If the workload does not complete, then t_{ExpEnd} is governed by the value of a watchdog timer. If, at the end of the watchdog timer, the BC has not received the end signal from the OS, it then attempts to connect to the OS. If this connection is successful, and if the soft reboot is successful, then a workload abort or hang state is diagnosed. If the soft reboot is unsuccessful, then a panic state, SPc, is deduced and a hard reboot is required. Otherwise SHg is assumed.

At the end of a benchmark execution, all files containing raw results corresponding to all experiments are on the BC. A processing module extracts automatically the relevant information from these files (two specific modules are required for Windows and Linux families). The relevant information is then used to evaluate automatically the benchmark measures (the same module is used for Windows and Linux).

3.4. Benchmark Characteristics

For each system call activated by the workload, several parameters are corrupted leading to several experiments for the same system call. The number of system calls (activated by JVM under the program allowing to display «*Hello World*» on the screen) and the associated number of experiments for the OSs considered are indicated in Table 1.

Table 1: Number of System Calls and Experiments for each OS

	Windows family			Linux family			
	W- NT4	W- 2000	W- XP	L- 2.2.26	L- 2.4.5	L- 2.4.26	L- 2.6.6
# System Calls	76	76	76	37	32	32	31
# Experiments	1285	1294	1282	457	408	408	409

4. Benchmark Results

4.1. Measures

OSs robustness is given in Figure 3. It shows that all OSs of the same family are equivalent. It also shows that none of the catastrophic states (*Panic* or *Hang* OS states) occurred for all Windows and Linux OSs. Linux OSs notified more error codes (58-66%) than Windows (25%), while more exceptions were raised with Windows (22-23%) than with Linux (7-10%). More no-signaling cases have been observed for Windows (52-54%) than for Linux (27-36%).

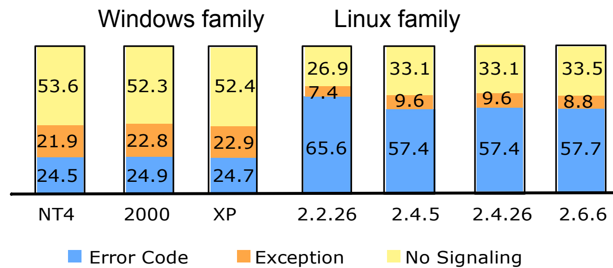

Fig. 3: OS Robustness (%)

These results are in conformance with our previous results, related to Windows using TPC-C Client [2] and to Windows and Linux using PostMark [3]. In [4] it was observed that on the one hand Windows 95, 98, 98SE and CE had a few catastrophic failures and on the other hand Windows NT, Windows 2000 and Linux are more robust and did not have any catastrophic failures as in our case.

The reaction times in the presence of faults (and without fault) are given in Figure 4. Note that for the Windows family, XP has the lowest reaction time, and for the Linux family, 2.6.6 has the lowest one. However, the reaction times of Windows NT and 2000 are very high. A detailed analysis showed that the large response time for Windows NT and 2000 are mainly due to system calls LoadLibraryA, LoadLibraryExA and LoadLibraryEXW. Not including these system calls when evaluating the average of the reaction time in the presence of faults leads respectively to 388μs, 182μs and 205μs for NT4, 2000 and XP (the associated average restart times without fault become respectively 191μs, 278μs and 298μs). For Linux the high values of the reaction times in presence of faults are also due to three system calls (execve, getdents64, nanosleep). Not including the reaction times associated to these system calls leads respectively to 88μs, 241μs, 227μs and 88μs for Linux 2.2.26, 2.4.5, 2.4.26 and 2.6.6.

The restart times are shown in Figure 5. The average restart time without faults, τres, is always lower than the benchmark restart time (with faults), Tres, but the difference is not significant. The standard deviation is very large for all OSs. Linux 2.2.26 and Windows XP have the lowest restart time (71 seconds, in the absence of fault) while Windows NT and 2000 restart times are around 90 seconds and those of Linux versions 2.4.5, 2.4.26 and 2.6.6 are around 80 seconds.

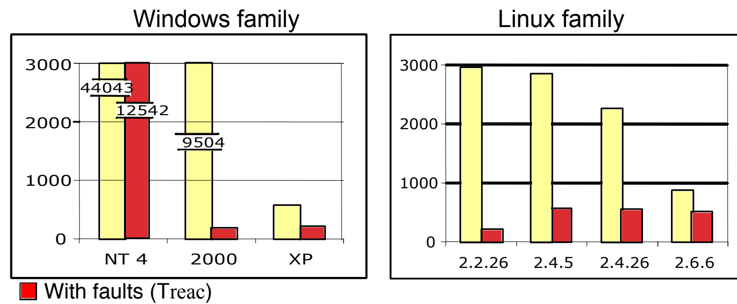

Fig. 4: OS Reaction Times (in μseconds)

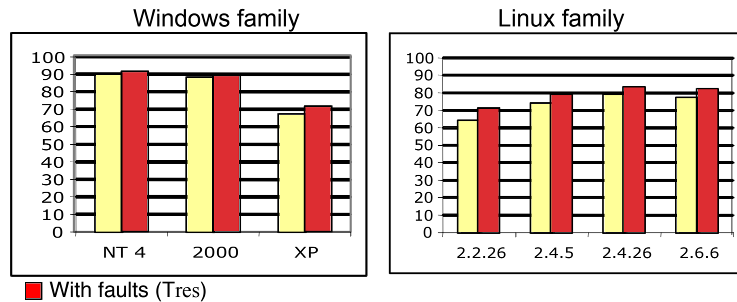

Fig. 5: OS Restart Times (in seconds)

4.2. Restart Time Refinement

It is worth to mention that the average restart times mask interesting phenomena. Detailed analyses show that all OSs of the same family have similar behavior and that the two families exhibit very different behaviors.

For Windows, there is a correlation between the restart time and the workload state at the end of the experiment. When the workload is completed, the restart time is almost the same as the average restart time without substitution. On the other hand, the restart time is statistically larger for all experiments with workload abort/hang. Moreover, statistically, the same system calls lead to workload abort/hang.

This is illustrated in Figure 6 in which the benchmark experiments are executed in the same order for the three Windows versions. Similar behaviors have been observed when using TPC-C [2] and PostMark workloads [3].

Linux restart time is not affected by the workload state. Detailed restart time analyses show high values appearing periodically. These values correspond to a check-disk performed by the Linux kernel every 26 restarts (which explains the important standard deviation on this measure). This is illustrated in Figure 7 for Linux 2.2.26, as an example. The same behavior has been observed when using the PostMark workload [3].

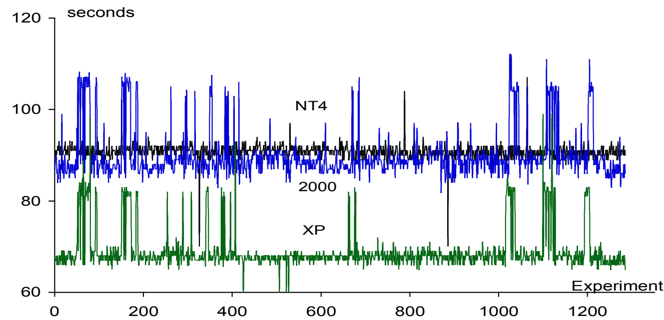

Fig. 6: Detailed Restart Time for Windows

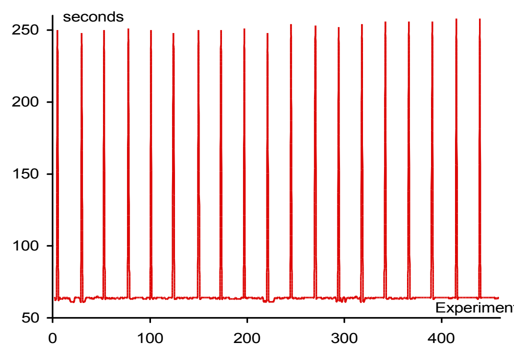

Fig. 7: Detailed Restart Time for Linux 2.2.26

5. Benchmark Properties

To be accepted and adopted by the scientific and industrial communities, a benchmark must satisfy a set of key properties, such as representativeness, repeatability, reproducibility, portability and cost-effectiveness. Representativeness concerns essentially the workload (that is without any doubt the most critical component of any dependability benchmark). All properties should be accounted for from the early phase of the benchmark specification as they directly impact the specification of all benchmark components. Some properties can be ensured by construction, some others have to be checked experimentally too.

In our previous work using TPC-C and PostMark as workloads, we have shown how the various properties have been taken into consideration what has been done to check some of them experimentally. In this section, we summarize the various properties and the kinds of verification carried out using JVM.

5.1 Faultload representativeness

It is very hard to guarantee that the faults used in our benchmark (corrupted values in system call parameters) are representative of all application software faults. Indeed, the OS is not expected to detect all application faults, but it is expected to avoid some

application faults that may lead to OS misbehavior. At least it is expected to detect system calls with obvious errors (such as out-of-range data or incorrect addresses). We have thus performed sensitivity analyses with respect to the parameter corruption technique.

The selective substitution technique used is composed of a mix of three corruption techniques as mentioned in Section 2.3: out-of-range data (OORD), incorrect data (ID) and incorrect addresses (IA). Let us denote the faultload used in our benchmarks by FL0. To analyze the impact of the faultload, we consider two subsets, including respectively i) IA and ODRD only (denoted FL1), and ii) ODRD only (denoted FL2). Taking Windows NT4 and Linux 2.2.26 as examples, moving from FL0 to FL2 the number of experiments decreases respectively from 1285 to 264 and from 457 to 119.

We ran the benchmarks of all OSs considered using successively FL0, FL1 and FL2. The results obtained confirm the equivalence between Linux family OSs as well as the equivalence between Windows family OSs, using the same faultload (FL0, FL1 or FL2). Indeed, for each OS, its robustness with respect to FL0, FL1 or FL2 is different but the robustness of all OSs of the same family with respect to each of the three faultloads is equivalent. The same results have been obtained using TPC-C Client and PostMark as workloads. This shows that using a mix of the three corruption techniques is meaningful.

5.2. Repeatability and Reproducibility

Repeatability is the property that guarantees *statistically equivalent results* when the benchmark is run more than once in the *same environment* (i.e., using the same system under benchmark and the same prototype). Our OS dependability benchmark is composed of a series of experiments. Each experiment is run after a system restart. The experiments are independent from each other and the order in which the experiments are run is not important at all. Hence, once the system calls to be corrupted are selected and the substitution values defined, the benchmark is fully repeatable. We have repeated all the benchmarks presented three times to check for repeatability.

Reproducibility is the property that guarantees that *another party* obtains statistically equivalent results when the benchmark is implemented from the *same specification* and is used to benchmark the same system under benchmarking. Reproducibility is strongly related to the amount of details given in the specification. The specification should be at the same time i) *general enough* to be applied to the class of systems addressed by the benchmark and ii) *specific enough* to be implemented without distorting the original specification. We managed to satisfy such a tradeoff. Unfortunately, we have not checked explicitly the reproducibility of the benchmark results by developing several prototypes by different people. On the other hand, the results seem to be independent from the faultload. This makes us confident about reproducibility.

5.3. Portability

Portability concerns essentially the faultload (i.e., its applicability to different OS families).

At the specification level, in order to ensure portability of the faultload, the system calls to be corrupted are not identified individually. We decided to corrupt all system calls of the workload. This is because OSs from different families do not necessarily comprise the very same system calls as they may have different APIs. However, most OSs feature comparable functional components.

At the implementation level, portability can only be ensured for OSs from the same family because different OS families have different API sets.

5.4. Cost

If a benchmark is very expensive, industry may not be ready to adopt it. Cost is expressed in terms of effort required to develop the benchmark, run it and obtain results. These steps require some effort that is, from our point of view, relatively affordable. In our case, most of the effort was spent in defining the concepts, characterizing the faultload and studying its representativeness.

The JVM benchmark benefited a lot from TPC-C and PostMark benchmarks as all benchmark components did exist and we had only to adapt them. The first step consisted in executing JVM for each OS to be benchmarked, to identify system calls activated. The second step was devoted to define, for each system call, the parameters to be corrupted and the exact substitution values, to prepare the database to be used in the Interception /substitution/ observation modules. This step took a couple of days for Linux family (activating 31-37 system calls depending on the version considered) and the double for Windows as it activates 76 system calls. Adaptation of the benchmark controller and of the Interception/substitution/observation modules required about one day for each family.

The benchmark duration ranges from one day for each Linux OS to less than three days for each Windows OS. More precisely, 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 less than 6 minutes without workload completion (including the watchdog timeout and the restart time). Thus, an experiment lasts less than 5 minutes for all OSs. The series of experiments of a benchmark is fully automated.

6. Conclusions

The dependability benchmark presented in this paper is the third benchmark we have developed for Windows and Linux, based on the same high-level specification of the benchmark but using different workloads. The results obtained are in conformance with those obtained with the two other workloads and increase our confidence in the benchmark specification and in the results obtained.

This benchmark and more generally the three benchmarks developed and applied show that all OSs of the same family are equivalent. They also show that none of the catastrophic states of the OS (*Panic* or *Hang*) occurred for any of the Windows and Linux OSs considered.

Linux OSs notified more error codes than Windows while more exceptions were raised with Windows than with Linux. More no-signaling cases have been observed for Windows than for Linux.

Concerning the OS reaction time measure, results show a great variation around the average due to a minority of system calls with large execution times that dodge the average. When these system calls are not considered, the reaction times of all the OSs of the same family become equivalent.

With respect to the restart time measure, Linux seems to be globally faster compared to Windows even though Windows XP and Linux 2.2.26 have the same restart times. Detailed analysis of the restart time showed i) a correlation between Windows restart time and the workload final state (in case of workload *hang* or *abort,* the restart time is higher

than in case of workload completion) and ii) that Linux performs a "check disk" after each 26 restarts after which the restart time is four times higher than the average.

We paid a particular attention to representativeness of faultload, and to the properties of repeatability, reproducibility, portability and cost effectiveness of the benchmark.

Acknowledgement

We would like to thank Jean Arlat who contributed to the first OS benchmark based on TPC-C Client. We are indebted to Ali Kalakech, Ana-Elena Rugina and Philippe Rumeau for their valuable contributions. They have implemented the series benchmarks based on TPC-C, PostMark and JVM, allowing to validate progressively the benchmark concepts, results and properties.

References

[1]. EC DBench project (IST-2000-25425), http://www.laas.fr/DBench, project final short report.

[2]. Kalakech A., K. Kanoun, Y. Crouzet and A. Arlat, *Benchmarking the Dependability of Windows NT, 2000 and XP*, Int. Conf. on Dependable Systems and Networks, Florence, Italy, pp. 681-686, 2004.

[3]. Kanoun K., Y. Crouzet, A. Kalakech, A. E. Rugina and P. Rumeau, *Benchmarking the Dependability of Windows and Linux using PostMark Workloads*, 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'2005), Chicago (USA), pp.11-20, 2005.

[4]. Shelton C. et al., *Robustness Testing of the Microsoft Win32 API*, Int. Conf. on Dependable Systems and Networks, New York, pp. 261-270, 2000.

[5]. Mukherjee A. and D. P. Siewiorek, *Measuring Software Dependability by Robustness Benchmarking*, IEEE Trans. of Software Engineering, Vol. 23 (6), pp. 366-378, 1997.

[6]. Chevochot P. and I. Puaut. *Experimental Evaluation of the Fail-Silent Behavior of a Distributed Real-Time Run-Time Support Built from COTS Components*, Proc. Int. Conference on Dependable Systems and Networks (DSN-2001), Göteborg, Sweden, 2001.

[7]. Arlat J. et al., *Dependability of COTS Microkernel-Based Systems*, IEEE Trans. on Computers, Vol.51 (2), pp. 138-163, 2002.

[8]. Gu W, Z. Kalbarczyk and R. K. Iyer, *Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors*, Int. Conf. on Dependable Systems and Networks, Florence, Italy, pp. 887-896, 2004.

[9]. Tsai T. K. et al., *An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems*, 26th Int. Symp. on Fault-Tolerant Computing, Sendai, Japan, pp. 314-323, 1996.

[10]. Koopman P. and J. DeVale, *Comparing the Robustness of POSIX Operating Systems*, 29th Int. Symp. on Fault-Tolerant Computing, Madison, pp. 30-37, 1999.

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

[12]. Chou A. et al., *An Empirical Study of Operating Systems Errors*, 18th ACM Symp. on Operating Systems Principles Banff, AL, Canada, pp. 73-88, 2001.

[13]. Albinet A., J. Arlat and J.-C. Fabre, *Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel*, Int. Conf. on Dependable Systems and Network, Florence, Italy, pp. 867-876, 2004.
[14]. Koopman P. et al., *Comparing Operating Systems using Robustness Benchmarks*, 16th Int. Symp. on Reliable Distributed Systems, Durham, USA, pp. 72-79, 1997.
[15]. Hunt G. and D. Brubaher, *Detours: Binary Interception of Win32 Functions*, 3rd USENIX Windows NT Symp., Seattle, Washington, USA, pp. 135-144, 1999.
[16]. McGrath R. and W. Akkerman, *Source Forge Strace Project*, *http://sourceforge.net /projects /strace*, 2004.

Karama Kanoun is currently Directeur de Recherche at LAAS-CNRS. She was Visiting Professor at the University of Illinois, Urbana Champaign, USA, for a semester, in 1998. Her current research interests include modeling, evaluation and benchmarking of computer system dependability. She has authored or co-authored more than 100 conference and journal papers, 5 books and 10 book chapters. She has been involved in more than 30 national research contracts and European projects. She has been a consultant for several French companies, the European Space Agency, Ansaldo Transporti and for the International Union of Telecommunications. She is Chairperson of the Special Interest Group on Dependability Benchmarking (SIGDeB) of the IFIP WG 10.4 and Chairperson of the French SEE Working Group "Design and Validation for Dependability".

Yves Crouzet received the Engineer degree from the Higher National School of Electronics, Electrical Engineering, Computer Science and Hydraulics, Toulouse, in 1975 and the "Docteur-Ingénieur" degree from the National Polytechnic Institute, Toulouse, in 1978. He is currently "Chargé de Recherche" at the National Center for Scientific Research (CNRS). Since 1975 he has been a member of the Dependable Computing and Fault-Tolerance group at LAAS-CNRS. During 1975-1982 he worked on the design and realization of self-checking VLSI circuits. Since 1982 his main research interests concern the experimental validation of dependable systems by fault-injection and the experimental validation of software testing methods by mutation analysis. He worked also on fault-tolerance of human errors.