Windows and Linux Robustness Benchmarks With Respect to Application Erroneous Behavior

Karama Kanoun, Yves Crouzet, Ali Kalakech, Ana-Elena Rugina

Windows and Linux Robustness Benchmarks With Respect to Application Erroneous Behavior

Karama Kanoun, Yves Crouzet, Ali Kalakech, and Ana-Elena Rugina
LAAS-CNRS, 7, Avenue Colonel Roche 31077 Toulouse Cedex 4, France

Abstract

This chapter presents the specification of dependability benchmarks for general-purpose operating systems with respect to application erroneous behavior, and shows examples of benchmark results obtained for various versions of Windows and Linux operating systems. The benchmark measures are: operating system robustness (as regards possible erroneous inputs provided by the application software to the operating system (OS) via the application programming interface), the OS reaction, and restart times in the presence of faults. Two workloads are used for implementing the benchmark: PostMark, a file system performance benchmark for operating systems, and the Java Virtual Machine (JVM) middleware, a software layer on top of the OS allowing applications in Java language to be platform independent.

1. Introduction

Software is playing an increasingly important role in our day-to-day life. In particular, operating systems (OSs) are more and more used even in critical application domains. Choosing the operating system that is best adapted to one's needs is becoming a necessity. For a long time, performance was the main selection criterion for most users and several performance benchmarks were developed and are widely used. However, an OS should not only have good performance but also a high dependability level. Dependability benchmarks emerged as a consequence. Their role is to provide useful information regarding the dependability of software systems [Tsai *et al.* 1996, Brown & Patterson 2000, Chevochot & Puaut 2001, Brown *et al.* 2002, Zhu *et al.* 2003, Zhu *et al.* 2003a]. This chapter is devoted to the specification, application and validation of two dependability benchmarks of OSs using two different workloads: PostMark, a file system performance benchmark, and JVM (Java Virtual Machine), a software layer on top of the OS allowing applications in Java language to be platform independent.

Benchmarking the dependability of a system consists of evaluating dependability or performance-related measures, experimentally or based on experimentation and modeling, in order to characterize objectively the system behavior in the presence of faults. Such an evaluation should allow non-ambiguous comparison of alternative solutions. Non-ambiguity, confidence in results and meaningfulness are ensured by a set of properties a benchmark should satisfy. For example, a benchmark must be representative, reproducible, repeatable, portable and cost effective. These properties should be taken into consideration from the earliest phases of the benchmark specification as they have a deep impact on almost all benchmark components. Verification of the benchmark key properties constitutes a large part of the benchmark validation.

Our dependability benchmark is a robustness benchmark. The robustness of an OS can be viewed as its capacity to resist/react to external faults, induced by the applications running on top of it, or originating from the hardware layer, or from device drivers. In this chapter we address 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. For the sake of conciseness, such erroneous inputs are referred to as *faults*.

The work reported in this chapter is a follow up of the European project on Dependability Benchmarking, DBench [DBench]. Our previously published work on OS dependability benchmarks was based on i) TPC-C Client performance benchmark for transactional systems [Kalakech *et al.* 2004b], ii) PostMark [Kanoun *et al.* 2005], and iii) JVM workload [Kanoun & Crouzet 2006].

The work reported in [Shelton *et al.* 2000] is the most similar to ours; it addressed the "non-robustness" of the POSIX and Win32 APIs (while we are interested in robust and non-robust behavior). Pioneer work on robustness benchmarking is published in [Mukherjee & Siewiorek 1997]. Since then, a few studies have addressed OS dependability benchmarks, considering real time microkernels [Chevochot & Puaut 2001, Arlat *et al.* 2002, Gu *et al.* 2004] or general purpose OSs [Tsai *et al.* 1996, Koopman & DeVale 1999]. Robustness with respect to faults in device drivers is addressed in [Chou *et al.* 2001, Durães & Madeira 2002, Albinet *et al.* 2004] and also in Chapter 13.

The remainder of the chapter is organized as follows. Section 2 gives the specification of the OS benchmarks. Section 3 presents benchmark implementation and results related to PostMark workload for Windows and Linux families. Section 4 presents benchmark implementation and results related to JVM workload. Section 5 refines

the benchmark results for PostMark and JVM. Section 6 outlines the main benchmark properties that are meaningful to OS benchmarks, and briefly shows what has been achieved to ensure and check them. Section 7 concludes the chapter.

2. Specification of the Benchmark

In order to provide dependability benchmark results that are meaningful, useful and interpretable, it is essential to define clearly the following benchmark components:

1) The benchmarking context.
2) The benchmark measures to be evaluated and the measurements to be performed on the system to provide the information required for obtaining the measures.
3) The benchmark execution profile to be used to exercise the operating system.
4) Guidelines for conducting benchmark experiments and implementing benchmark prototypes.

These components are presented hereafter.

2.1. Benchmarking Context

An OS can be seen as a generic software layer that manages all aspects of the underlying hardware. The OS provides i) basic services to the applications through the API, and ii) communication with peripheral devices via device drivers. From the viewpoint of dependability benchmarking, 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, for the benchmark target to be assessed, it is necessary to run it on top of a hardware platform and to use a set of libraries. Thus, the benchmark target along with the hardware platform and libraries form the *system under benchmarking*. Although, in practice, the benchmark measures characterize the system under benchmarking (e.g., the OS reaction and restart times are strongly dependent on the underlying hardware), for clarity purpose we will state that the benchmark results characterize the OS.

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 the 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 OS in terms of system calls (in addition of course to the description of the services provided by the OS).

2.2. Benchmark Measures

The benchmark measures include a robustness measure and two temporal measures.

After execution of a corrupted system call, the OS is in one of the states summarized in Table 1.

SEr	An *error code* is returned
SXp	An *exception* is raised, processed and notified
SPc	*Panic* state
SHg	*Hang* state
SNS	No-signaling state

Table 1: OS outcomes

SEr: corresponds to the case where the OS generates an error code that is delivered to the application.

SXp: corresponds to the case where the OS issues an exception. Two kinds of exceptions can be distinguished depending on whether it is issued during the application software execution (user mode) or during execution of the kernel software (kernel mode). In the user mode, the OS processes the exception and notifies the application (the application may or may not take into account explicitly this information). However, for some critical situations, the OS aborts the application. An exception in the kernel mode is automatically followed by a *panic* state (e.g., blue screen for Windows and oops messages for Linux). Hence, hereafter, the latter exceptions are included in the panic state and the term exception refers only to user mode exceptions.

SPc: In the panic state, 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: In this state, a hard reboot of the OS is required.

SNS: In the no-signaling state, the OS does not detect the presence of the corrupted parameter. As a consequence, it accepts the erroneous system call and executes it. It may thus abort, hang or complete its execution. However, the response might be erroneous or correct. For some system calls, the application may not require any explicit response, so it simply continues execution after sending the system call. SNS is presumed when none of the previous outcomes (SEr, SXp, SPc, SHg) is observed[1].

Panic and *hang* outcomes are actual states in which the OS can stay for a while. They characterize the OS's non-robustness. Conversely, SEr and SXp characterize only events. They are easily identified when the OS provides an error code or notifies an exception. These events characterize the OS's robustness.

OS Robustness (POS) is defined as the percentages of experiments leading to any of the outcomes listed in Table 1. 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 the presence of faults, either by notifying an exception or by returning an error code or by executing the system call.

Restart Time (Tres) corresponds to the average time necessary for the OS to restart after the execution of the workload in the presence of one fault in one of its system calls. 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 observed in absence of faults for comparison purpose. They are respectively denoted $\tau reac$ and τres.

---

1 In our work we do not address "hindering" errors (e.g., incorrect error codes being returned) as in [Shelton et al. 2000]. This is extremely difficult to deal with, and it is almost impossible to automate their identification.

Note that our set of measures is different from the one used in [Shelton et al. 2000]. Shelton et al. only take into account non-robust behavior of the OS. They use the CRASH scale to measure the OS non-robustness (**C**atastrophic, **R**estart, **A**bort, **S**ilent and **H**indering failures). Our aim is to distinguish as clearly as possible the proportions of robust versus non-robust behavior of the OS. Also, we completed the set of measures by adding to it the two temporal measures (*Texec* and *Tres*) presented above.

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, corrupted parameters in system calls. The set of corrupted parameters is referred to as the faultload.

The benchmark is defined so that the workload could be any performance benchmark workload (and, more generally, any user specific application) intended to run on top of the target OS. In [Kalakech *et al.* 2004b] we have used the workload of TPC-C Client [TPC-C 2002], and in this work we use two workloads PostMark [Katcher 1997] and JVM [Lindholm & Yellin 1999]. The two workloads will be discussed further in Sections 3 and 4.

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

We use a parameter corruption technique relying on thorough analysis of system call parameters to define *selective substitutions* to be applied to these parameters (similar to the one used in [Koopman *et al.* 1997]). 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 (containing usually an incorrect or out-of-range data). We use a mix of these three corruption techniques[2]. Note that non-existing addresses are always detected. Hence they are not considered as interesting substitution values.

To reduce the number of experiments, the parameter data types are grouped into classes. A set of substitution values is defined for each class. They depend on the definition of the class. Some values require a *pre* and a *post* processing such as the creation and the destruction of temporary files. For example, for Windows, we group the data types into 13 classes. Among these classes, 9 are pointer classes. Apart from *pvoid* (pointer which points to anything), all other pointers point to a particular data type. Substitution values for these pointers are combination of pointer substitution values and the corresponding data type substitution values. Similarly, for Linux, we group the data types into 13 classes among which 5 are pointer classes. We use the same substitution values for basic data types (i.e., integer) both for Windows and Linux. Nevertheless, some data types are system-dependent. Consequently, they have specific substitution values. In Linux, for example, we define a class corresponding to the type mode. A mode is an integer with a particular meaning: read/write modes or permission flags. As the validity domain of this data type can be identified precisely, pertinent substitution values are defined for it. Table 2 reviews the substitution values associated with the basic data type classes.

---

2 A discussion about the adequacy of this choice is given in Section 6.1.3.

Data type class	Substitution values					
Pvoid	NULL	0xFFFFFFFF	1	0xFFFF	-1	Random
Integer	0	1	MAX INT	MIN INT	0.5	
Unsigned integer	0	1	0xFFFFFFFF	-1	0.5	
Boolean	0	0xFF (Max)	1	-1	0.5	
String	Empty	Large (> 200)	Far (+ 1000)			

Table 2: Parameter substitution values

2.4. Benchmark Conduct and Implementation

Since perturbing the operating system may lead the OS to hang, a remote machine, referred to as the benchmark controller, is required to reliably 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). 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 in charge of diagnosing and collecting part or all benchmark data. The two machines perform the following functions: 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, iv) observation and collection of OS outcomes.

The experiment steps in case of workload completion are illustrated in Figure 1. In case of workload non-completion state (i.e., the workload is in abort or hang state), the end of the experiment is provided by a watchdog timeout as illustrated in Figure 2. The timeout duration is fixed to a value that is three times greater than the largest workload execution time without faults.

To intercept Win32 functions, we use the Detours tool [Hunt & Brubaher 1999], a library for intercepting Win32 binary functions on X86 machines. The part of Detours in charge of system call interception is composed of 30 Kilo-lines of code (KLOC). The modifications we carried out on this tool concern i) the replacement of system call parameters by corrupted values (this module is 3 KLOC) and ii) the addition of modules to observe the reactions of the OS after parameter corruption, and to collect the required measurements (this module is 15 KLOC). To intercept *POSIX* system calls, we used another interception tool, Strace [McGrath & Akkerman 2004]. Strace is composed of 26 KLOC. Also, we added two modules to this tool to allow i) substitution of the parameters and ii) observation of Linux behavior after parameter corruption (these modules correspond to 4 KLOC together). The reaction time is counted from the time the corrupted system call is re-inserted. Hence the time to intercept and substitute system calls is not included in the system reaction time, as shown in Figures 1 and 2.

Figure 3 summarizes the various components of the benchmark environment. All the experiments have been run on the same target machine, composed of an Intel Pentium III Processor, 800 MHz, and a memory of 512 Megabytes. The hard disk is 18 Gigabytes, ULTRA 160 SCSI. The benchmark controller in both prototypes for Windows and Linux is a Sun Microsystems workstation.

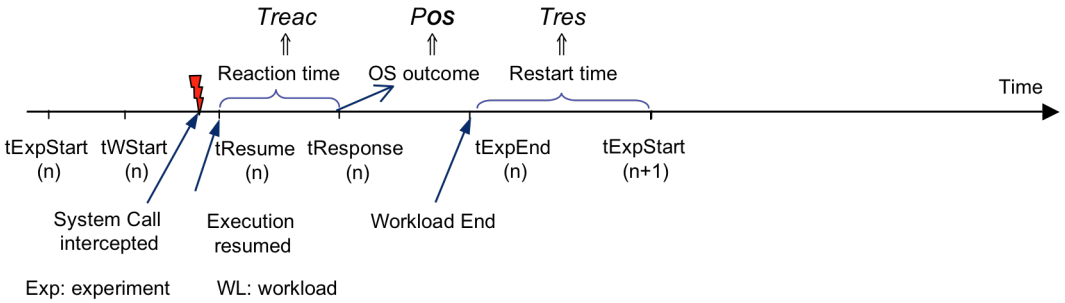

Figure 1: Benchmark execution sequence in case of workload completion

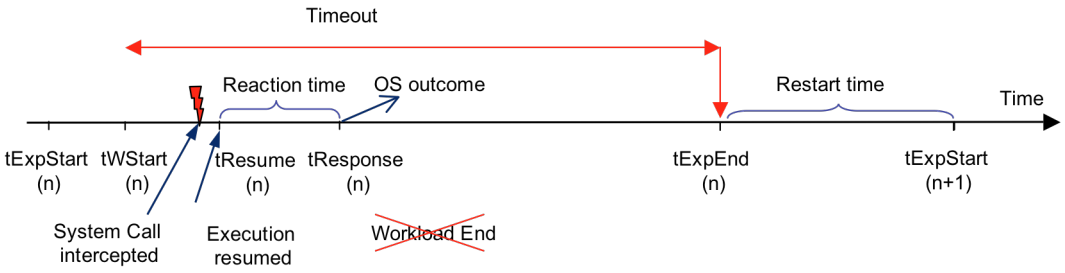

Figure 2: Benchmark execution sequence in case of workload abort or hang

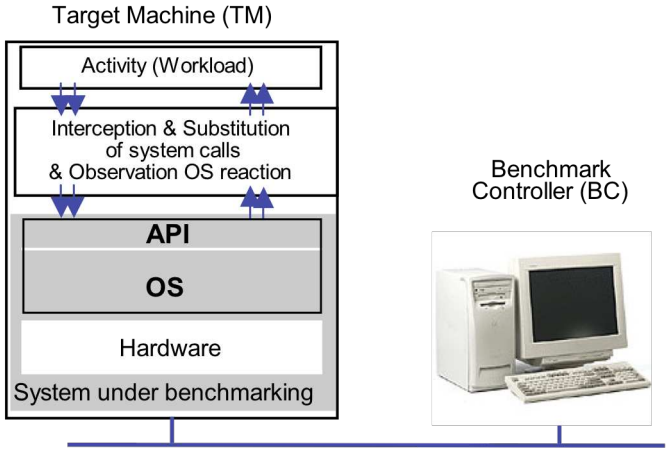

Figure 3. Benchmark environment

Before each benchmark run (i.e., before 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 because they depend on kernel headers that are different from one version to another). Once the benchmarking tool is compiled, it is used to identify the set of system calls activated by the workload. All parameters of all these system calls are then analyzed and placed into the corresponding class. A database of substitution values is then generated accordingly.

Following the benchmark execution sequence presented in Figures 1 and 2, 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, 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 checks whether the current system call has parameters. If it is not the case, the execution is simply resumed; otherwise, the execution is 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, and SNS. The corresponding OS response time ($t_{Response}$) is recorded in the experiment execution trace. For each run, the OS reaction time is calculated as the difference between $t_{Response}$ and t_{Resume}.

At the end of the execution of the workload, the OS notifies the BC of the end of the experiment by sending an end signal along with the experiment end instant, t_{ExpEnd} , and then it restarts so that the current experiment does not have any effects on the following experiment. If the workload does not complete, then t_{ExpEnd} is governed by the value of a watchdog timer. The BC collects the SHg state and the workload abort/hang states. It is in charge of restarting the system in such cases. When no faultload is applied, the average time necessary for the OS to execute PostMark or JVM is less than 1 minute for Windows and for Linux. We considered that three times the normal execution time is enough to conclude on the experiment's result. Thus, we have fixed the watchdog timer to 3 minutes. If, at the end of this watchdog timer, the BC has not received the end signal from the OS, it then attempts to ping the OS. If the OS responds (ping successful), the BC attempts to connect to it. If the connection is successful, then a workload abort or hang state is diagnosed. If the connection is unsuccessful, then a panic state, SPc, is deduced. Otherwise, SHg is assumed. If workload abort / hang or SPc or SHg are observed, $t_{Response}$ does not take any value for the current experiment. Thus, the measure Treac is not skewed by the watchdog timer value.

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. PostMark Dependability Benchmark Implementation and Results

PostMark creates a large pool of continually changing files and measures the transaction rates for a workload emulating Internet applications such as e-mail or netnews. It generates an initial pool of random text files ranging in size from a configurable low bound to a configurable high bound. The file pool is of configurable size and can be located on any accessible file system. The workload of this benchmark, referred to as PostMark for simplicity, is responsible for realizing a number of transactions. Each transaction consists of a pair of smaller transactions: i) create file or delete file and ii) read file or append file. PostMark is written in the C language. From a practical point of view, PostMark needs to be compiled separately for each OS.

Six versions of Windows OSs are targeted: Windows NT4 Workstation with SP6, Windows 2000 Professional with SP4, Windows XP Professional with SP1, Windows NT4 Server with SP6, Windows 2000 Server with SP4 and Windows 2003 Server. In the rest of this chapter, Windows 2000 Professional and Windows NT4

Workstation will be referred to as Windows 2000 and Windows NT4 respectively. Four Linux OSs (Debian distribution) are targeted: Linux 2.2.26, Linux 2.4.5, Linux 2.4.26 and Linux 2.6.6. Each of them is a revision of one of the stable versions of Linux (2.2, 2.4, 2.6). Table 3 summarizes the number of system calls targeted by the benchmark experiments carried out along with the number of corresponding parameters and the number of experiments for each OS.

	Windows family						Linux family			
	W- NT4	W- 2000	W- XP	W- NT4S	W- 2000S	W- 2003S	L- 2.2.26	L- 2.4.5	L- 2.4.26	L- 2.6.6
# System Calls	25	27	26	25	27	27	16	16	16	17
# Parameters	53	64	64	53	64	64	38	38	38	44
# Experiments	418	433	424	418	433	433	206	206	206	228

Table 3: Number of system calls, corrupted parameters and experiments for each OS, using PostMark

OS robustness is given in Figure 4. It shows that all OSs of the same family are equivalent, which is in conformance with our previous results, related to Windows using TPC-C Client [Kalakech *et al.* 2004b]. It also shows that none of the catastrophic outcomes (*Panic* or *Hang* OS states) occurred for all Windows and Linux OSs. Linux OSs returned more error codes (59-67%) than Windows (23-27%), while more exceptions were raised with Windows (17-22%) than with Linux (8-10%). More no-signaling cases have been observed for Windows (55-56%) than for Linux (25-32%). In [Shelton *et al.* 2000] 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.

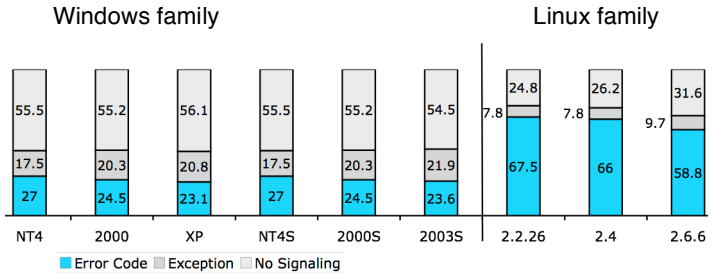

Figure 4: OS Robustness (%), using PostMark

The reaction time is given in Figure 5. Globally, Windows OSs have shorter response times than Linux OSs. The standard deviation is significantly larger than the average for all OSs. Except for the two revisions of Linux 2.4, τreac is always larger than Treac, the reaction time in the presence of faults. This can be explained by the fact that after parameter corruption, the OS detects the anomaly in almost 45% of cases for Windows and 75% of cases for Linux, and stops system call execution, returns an error code or notifies an exception.

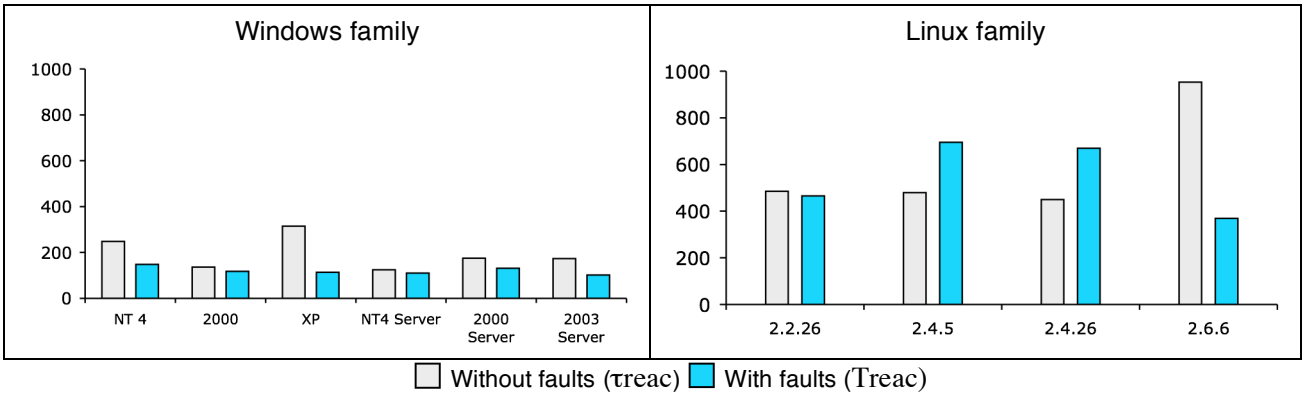

Figure 5: Reaction time (in micro seconds), using Postmark

Note that for the Windows family, Windows XP has the lowest reaction time in the presence of faults and for the Linux family, Linux 2.6.6 has the lowest reaction time. For Linux 2.6.6, we notice that τreac is almost two times larger than for the other revisions. A detailed analysis of the results showed that this is due to one system call, execve, for which the execution time is 15000 µs for Linux 2.6.6 and 6000 µs for other versions. These values may be considered as outliers. Removing them and re-evaluating the average lead to more representative results. Another measure of interest in this case could be the median, since it is not sensitive to extreme values.

The restart times are shown in Figure 6. The average restart time without faults, τres, is always lower than the average restart time with faults (Tres), but the difference is not significant. Linux seems to be globally faster (71-83s) than Windows (74-112s). However, if we consider only OS versions introduced in the market after 2001, the other OSs rank as follows: Linux 2.2.26 (71s), Windows XP (74s), Windows 2003 server (77s), Linux 2.4.5 (79s), Linux 2.6.6 (82s), Linux 2.4.26 (83s).

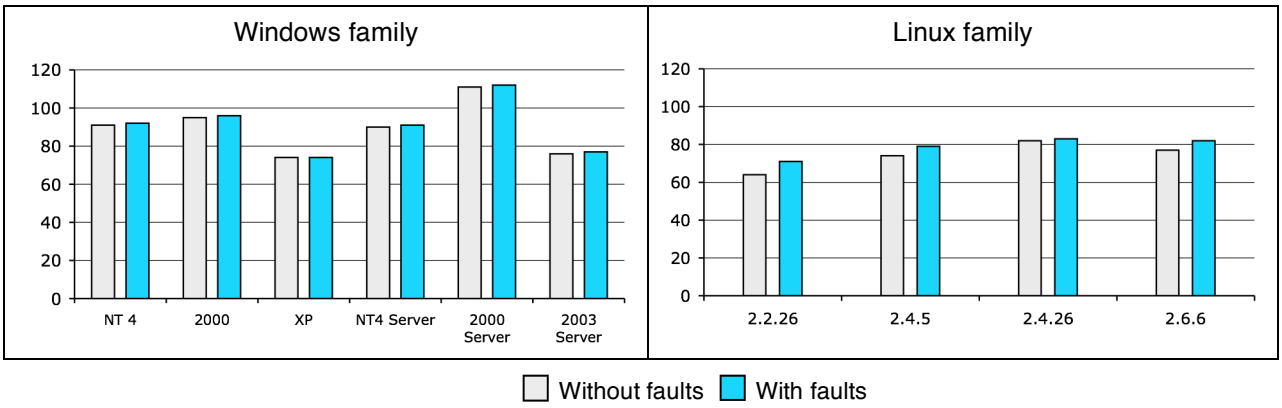

Figure 6: Restart time (in seconds), using Postmark

Concerning the Linux family, we note that the restart time increases with new versions or revisions, except for Linux 2.6.6. This progression is due to the increasing size of kernels with the version evolution. The exception of Linux 2.6.6 is justified by the fact that the Linux kernel was restructured in its version 2.6.

4. JVM Dependability Benchmark Implementation and Results

Java Virtual Machine (*JVM)* is a software layer between the OS and Java applications, allowing applications in Java language to be platform independent. The specifications of the virtual machine [Lindholm & Yellin 1999] are independent from the hardware platform but each platform requires a specific implementation. The benchmark based on JVM has been applied to three Windows versions (NT, 2000 and XP) and to the four Linux versions considered in the previous section. In this benchmark, JVM is solicited through a small program allowing to activate 76 system calls with parameters for the Windows family and 31 to 37 system calls with parameters for the Linux Family, as indicated in Table 4. These system calls are intercepted and corrupted using the corruption technique presented in Section 2.3, which leads to the number of experiments indicated in the last line 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
# Parameters	216	214	213	86	77	77	77
# Experiments	1285	1294	1282	457	408	408	409

Table 4: Number of system calls, corrupted parameters and experiments for each OS, using JVM

Robustness is given in Figure 7. As for the PostMark workload, no *Hang* or *Panic* states have been observed. It can be noticed that the three Windows versions are roughly equivalent, as well as the Linux versions.

Comparison with Figure 4 shows that the robustness of each family is the same using PostMark or JVM workloads (within 5% discrepancy). Furthermore, we have observed the same robustness for Windows versions using TPC-C as workload in our previous work [Kalakech *et al.* 2004a]. The three workloads solicit different numbers of systems calls and only some of them are the same. Nevertheless they lead to the same robustness.

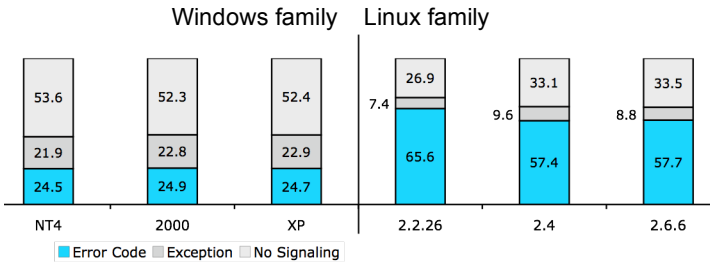

Figure 7: Robustness measures with JVM

The reaction times in the presence of faults (and without fault) are given in Figure 8. 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. For Linux, the extremely high values of the reaction times without faults are due to two system calls (sched_getparam and sched_getscheduler). Their

execution times are significantly larger without fault than in the presence of faults. A detailed analysis of the results showed that for these two system calls, most of the corruption experiments ended with an error code (SEr). Thus, we assume that the system calls were abandoned after an early anomaly detection by the OS. Also for Linux, the reaction times in the presence of faults are relatively high. This is due to three system calls (execve, getdents64 and nanosleep). Not including the reaction times associated with these system calls leads respectively to a Treac of 88µs, 241µs, 227µs and 88µs for the 2.2.26, 2.4.5, 2.4.26 and 2.6.6 versions.

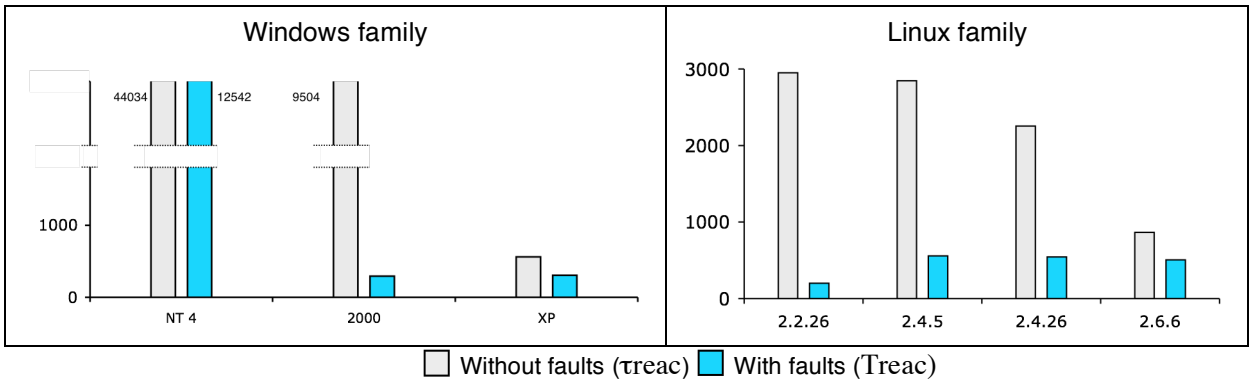

Figure 8: OS reaction time (in micro seconds), using JVM

The restart times are given in Figure 9. As for the PostMark workload, the average restart time without faults, τres, is always lower than the benchmark restart time (in the presence of 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 faults) 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.

It is interesting to note that the order of the OSs considered is the same for PostMark and for JVM, except for Windows NT and 2000 (which have the highest restart times). Indeed, the only change concerns Windows 2000 whose restart time is slightly decreased for JVM, making it better than Windows NT.

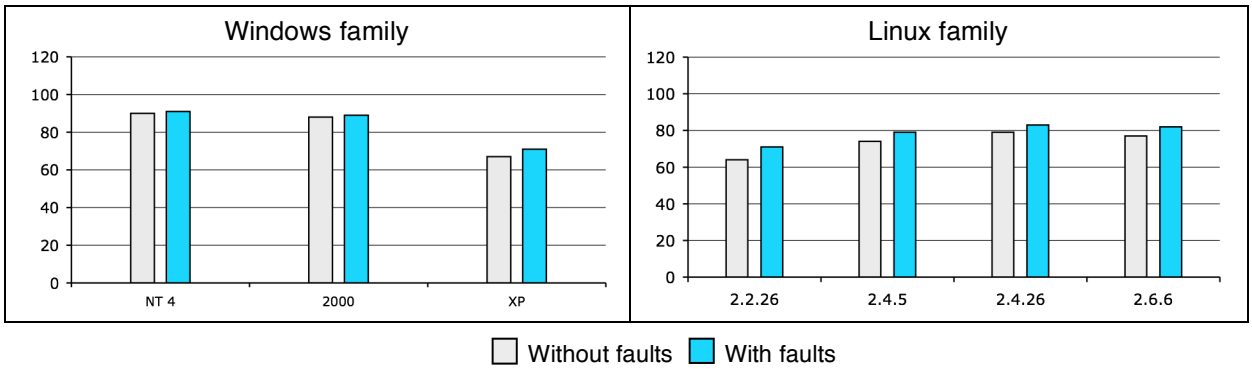

Figure 9: OS restart time (in seconds), using JVM

5. Results Refinement

The benchmark temporal measures are refined to provide more insights into those presented in Sections 3 and 4. We first consider PostMark, then JVM. For each of them, we mainly detail the temporal measures.

5.1. PostMark

5.1.1 Reaction time

Table 5 presents the detailed reaction times with respect to OS outcomes after execution of corrupted system calls (Error Code, Exception and No Signaling). Thus, three average times are added to detail Treac: TSEr, TSXp (the times necessary to return respectively an error code or an exception) and TSNS (the execution time of the corrupted system call, in case of no-signaling state).

***Windows* Family**

	NT4		2000		XP	
	Avg.	S. Dev.	Avg.	S. Dev.	Avg.	S. Dev.
Treac	148 µs	219 µs	118 µs	289 µs	114 µs	218 µs
TSEr	45 µs	107 µs	34 µs	61 µs	45 µs	118 µs
TSXp	40 µs	15 µs	37 µs	15 µs	50 µs	96 µs
TSNS	234 µs	437 µs	186 µs	375 µs	168 µs	265 µs
	NT4 Server		**2000 Server**		**2003 Server**	
Treac	110 µs	221 µs	131 µs	289 µs	102 µs	198 µs
TSEr	41 µs	66 µs	29 µs	33 µs	25 µs	61 µs
TSXp	35 µs	15 µs	37 µs	15 µs	48 µs	20 µs
TSNS	166 µs	280 µs	210 µs	396 µs	156 µs	252 µs

***Linux* Family**

	2.2.26		2.4.5		2.4.26		2.6.6	
Treac	167 µs	300 µs	466 µs	2276 µs	425 µs	2055 µs	93 µs	12 µs
TSEr	208 µs	361 µs	92 µs	105 µs	84 µs	6 µs	91 µs	10 µs
TSXp	88 µs	5 µs	91 µs	8 µs	91 µs	8 µs	106 µs	13 µs
TSNS	85 µs	5 µs	1545 µs	4332 µs	1405 µs	3912 µs	91 µs	11 µs

Table 5: Detailed reaction time, using PostMark

For the Windows family, it can be seen that for versions 2000, 2000 Server, XP and 2003 Server, returning an error code takes less time than raising an exception. This can be explained by the fact that when returning an error code, tests are carried out on the parameter values at the beginning of the system call code and the system call is abandoned, while the exceptions are raised from a lower level of the system under benchmarking. Nevertheless, in the cases of Windows NT4 and NT4 Server, TSEr is higher than TSXp. The cause of this anomaly lies in the long time necessary to GetCPInfo system call to return an error code when its first parameter is corrupted.

Concerning the Linux family, the averages presented in this table do not take into account execve system call execution time. Cells in grey correspond to high values of the standard deviation and are commented hereafter. We notice the high values of TSNS corresponding to the two revisions of version 2.4, compared to the two other versions. The very high standard deviation suggests a large variation around the average, which is confirmed in

Figure 10 that gives the OS reaction time for all system calls leading to the no-signaling state for all Linux OSs. We can clearly see that for Linux 2.4 the average time necessary for executing mkdir is more than 10 times larger than for all other system calls. We have noticed that mkdir has an extremely long execution time when its second parameter (which corresponds to the permissions to apply on the newly created folder) is corrupted.

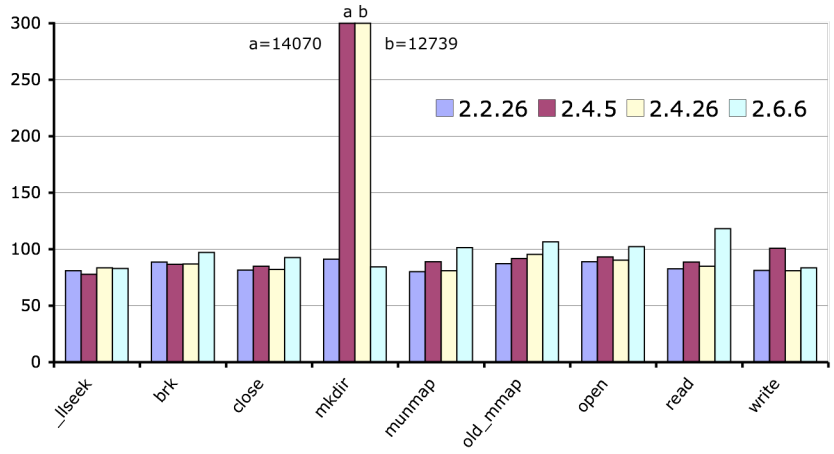

Figure 10: Linux reaction time in case of SNS (in micro seconds), using PostMark

Also, a very large average time to return an error code is observed for Linux 2.2.26, with a high standard deviation. Figure 11 details the times necessary to return error codes for Linux system calls. It is clear that these times are very similar except for unlink system call in Linux 2.2.26, which explains the high TSEr of Linux 2.2.26 compared to the other versions. After discarding the exceptional values corresponding to execve, mkdir and unlink system calls, the average reaction times Treac of the four targeted Linux OSs become very close. The largest difference is of 8µs. Also, the average reaction times with respect to OS outcomes after execution of corrupted system calls (TSEr, TSXp, TSNS) become very close. The largest difference is of 18µs. Furthermore, τreac and Treac become very close.

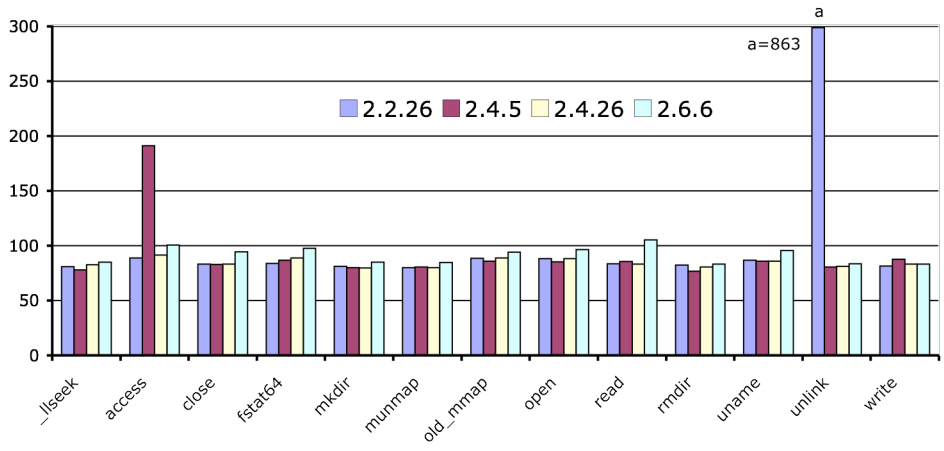

Figure 11: Linux reaction time in case of SER (in micro seconds), using PostMark

5.1.2 Restart time

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 state of the workload at the end of the experiment. When the workload is completed, the average restart time is statistically equal to the restart time without parameter substitution. On the other hand, the restart time is larger and statistically equal for all experiments with workload abort/hang. This is illustrated in Figure 12 in case of Windows NT, 2000 and NT. For example, the average restart time in case of workload completion is 73 s and 80 s in case of workload abort/hang, for Windows XP.

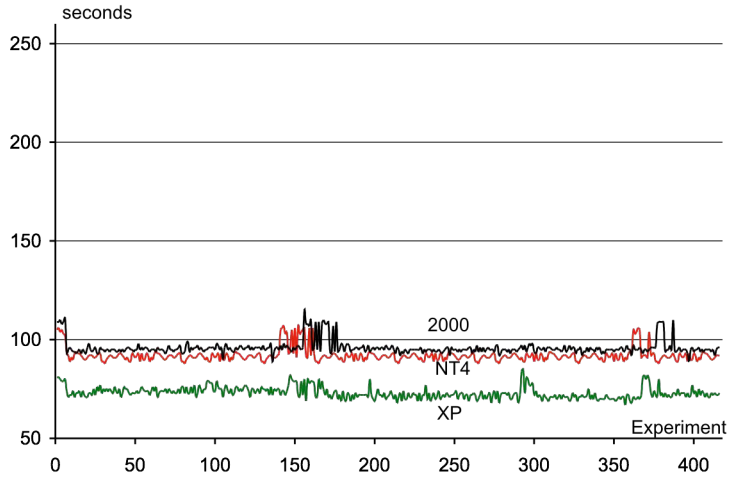

Figure 12: Detailed Windows restart time, using PostMark

Linux restart time is not affected by the workload final state. Detailing Linux restart times shows high values appearing periodically. These values correspond to a "check-disk" performed by the Linux kernel every 26 Target Machine restarts. This is illustrated for Linux 2.2.26 and 2.6.6 in Figure 13, and induces an important standard deviation on this measure. Also, it is interesting to note that the check-disk duration decreases with the version evolution, while the regular restart time increases. It seems natural for the time needed to complete a check-disk to decrease while the Linux kernel evolves. The increase of the regular restart time may be due to the increasing size of Linux kernels.

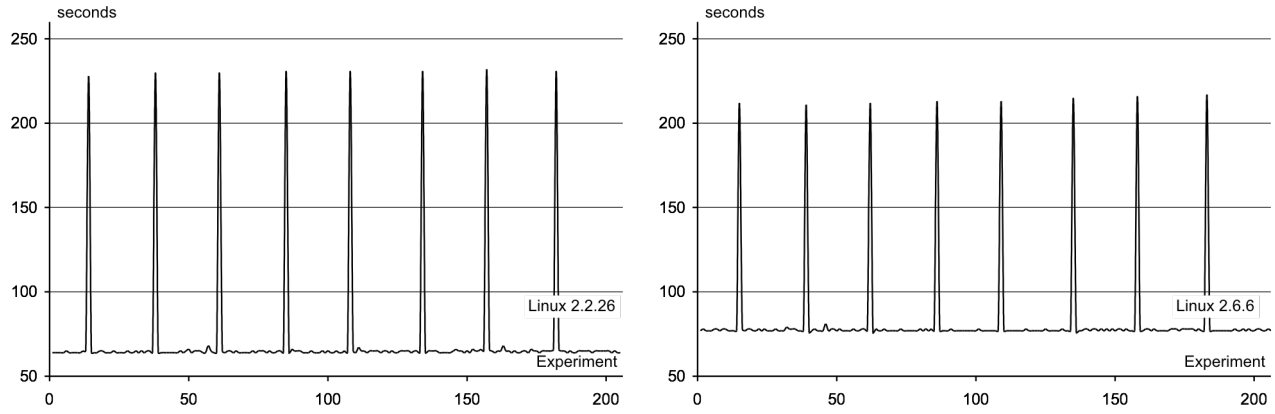

Figure 13: Detailed Linux restart time, using PostMark

5.2. JVM

5.2.1 Reaction time

Similarly to Table 5, Table 6 presents the detailed reaction times with respect to OS outcomes after execution of corrupted system calls (Error Code, Exception and No Signaling).

For Windows family, the averages presented in this table do not take into account LoadLibraryA, LoadLibraryExA and LoadLibraryExW system call execution times. As in the case of the use of the PostMark workload, we notice that, for Windows 2000 and Windows XP, T_{SEr} is higher than T_{SXp}, which is higher than T_{SNS}. The standard deviations for these measures are rather small. On the other hand, for Windows NT4 T_{SEr} is lower than T_{SXp}. Figure 14 details the times necessary to raise exceptions for Windows system calls. It is clear that these times are similar except for the system call FindNextFileW. The execution time of this system call is greater than 13400 µs for Windows NT4 while for the other Windows OSs it is smaller than 75 µs. It is noteworthy that the execution time of this system call is also the cause for the large values for T_{SE} and T_{SXp} in the case of NT4.

***Windows* Family**

	NT4		2000		XP	
	Avg.	S. Dev.	Avg.	S. Dev.	Avg.	S. Dev.
Treac	388 µs	3142 µs	190 µs	451 µs	214 µs	483 µs
TSEr	298 µs	3006 µs	47 µs	110 µs	51 µs	103 µs
TSXp	424 µs	3862 µs	84 µs	168 µs	98 µs	209 µs
TSNS	417 µs	2858 µs	307 µs	588 µs	344 µs	625 µs

***Linux* Family**

	2.2.26		2.4.5		2.4.26		2.6.6	
Treac	88 µs	85 µs	241 µs	1479 µs	227 µs	1438 µs	88 µs	26 µs
TSEr	90 µs	101 µs	79 µs	6 µs	84 µs	30 µs	86 µs	8 µs
TSXp	87 µs	7 µs	85 µs	8 µs	87 µs	8 µs	98 µs	15 µs
TSNS	84 µs	6 µs	572 µs	2545 µs	523 µs	2545 µs	89 µs	43 µs

Table 6: Detailed reaction time, using JVM

Concerning the Linux family, the averages presented in this table do not take into account execve, getdents64 and nanosleep system call execution times. As in the case of the use of the PostMark workload, we notice the high values of T_{SNS} corresponding to the two revisions of version 2.4, compared to the two other versions. The very high standard deviation suggests a large variation around the average, which is confirmed in Figure 15 that gives the OS reaction time for all system calls leading to the no-signaling state for all Linux OSs. As in the case of the use of PostMark workload, for Linux 2.4 the average time necessary for executing mkdir is more than 10 times larger than for all other system calls.

After discarding the exceptional values corresponding to execve, getdents64, nanosleep and mkdir system calls, the average reaction times T_{reac} of the four targeted Linux OSs become very close. The largest difference is of 15µs. Also, the average reaction times with respect to OS outcomes after execution of corrupted system calls (T_{SEr}, T_{SXp}, T_{SNS}) become very close. The largest difference is of 28µs. Furthermore, τ_{reac} and T_{reac} become very close (if we discard the executions times for sched_getparam and sched_getscheduler system calls).

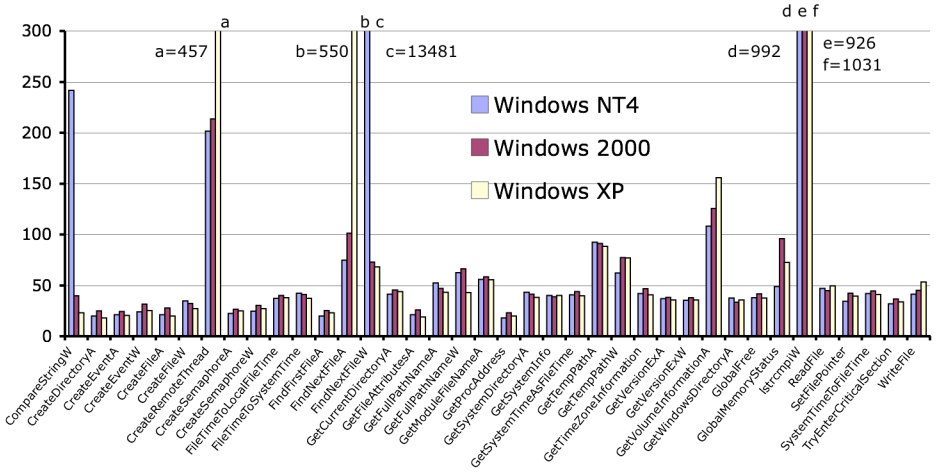

Figure 14: Windows reaction time in case of SXp (in micro seconds), using JVM

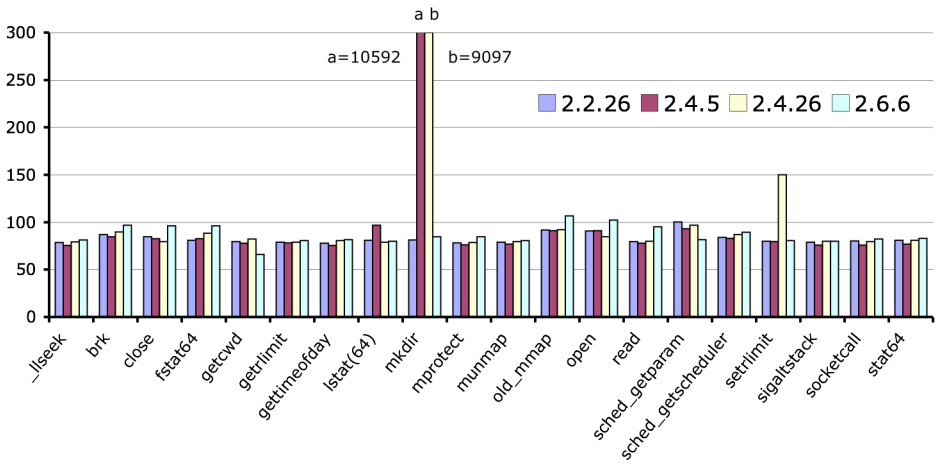

Figure 15: Linux reaction time in case of SNS (in micro seconds), using JVM

5.2.2 Restart time

As in the case of the PostMark workload, for Windows, there is a correlation between the restart time and the state of the workload at the end of the experiment. When the workload is completed, the average restart time is statistically equal to the restart time without parameter substitution. On the other hand, the restart time is larger and statistically equal for all experiments with workload abort/hang. This is illustrated in Figure 16. For example, for Windows XP, the average restart time in case of workload completion is 68 s and 82 s in case of workload abort/hang.

As in the case of the PostMark workload, Linux restart time is not affected by the workload final state. Detailing the restart times shows high values appearing periodically, due to a "check-disk" performed by the kernel every 26 Target Machine restarts. This is illustrated for Linux 2.2.26 and 2.6.6 in Figure 17. Also, it is noteworthy that the check-disk duration decreases with the version evolution, while the regular restart time increases.

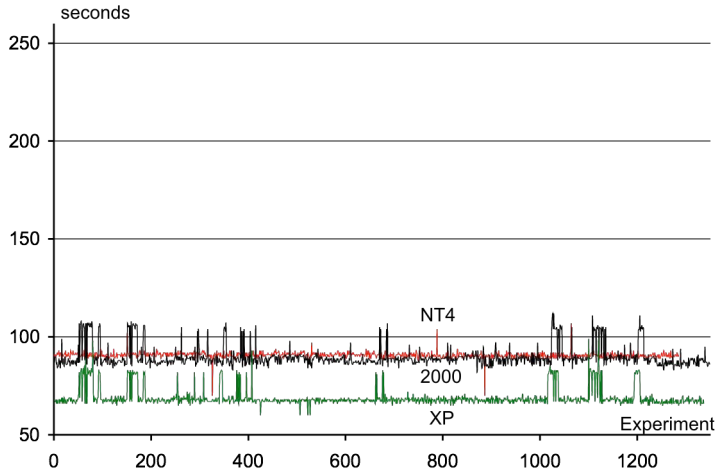

Figure 16: Detailed Windows restart time, using JVM

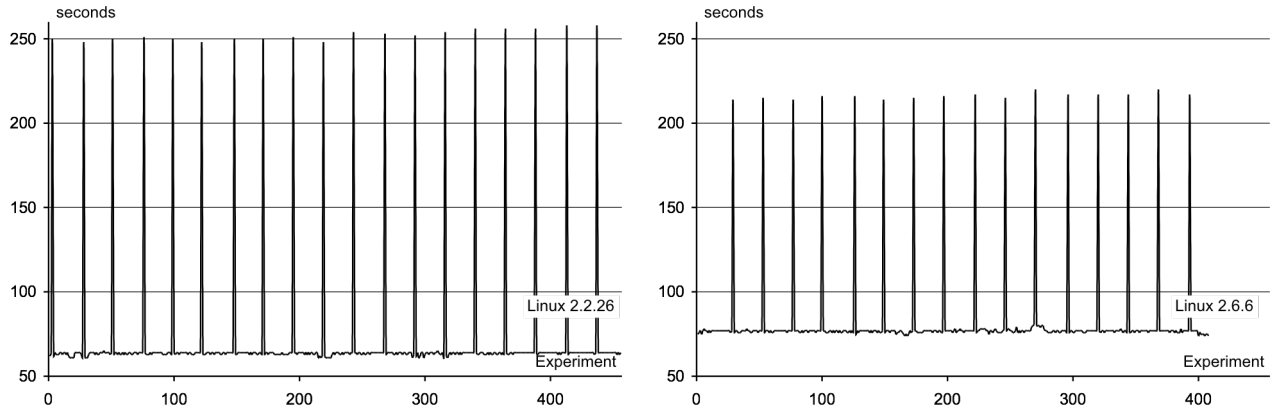

Figure 17: Detailed Linux restart time, using JVM

5.3. PostMark and JVM

OS robustness of all OSs with respect to PostMark and JVM workloads is synthesized in Table 7. It shows that none of the catastrophic outcomes (*Panic* or *Hang* OS states) occurred for all Windows and Linux OSs. It also shows that Linux OSs reported more error codes (57%-67%) than Windows (23%-27%), while more exceptions were raised with Windows (17%-25%) than with Linux (7%-10%). More no-signaling cases have been observed for Windows (52%-56%) than for Linux (25%-33%). For Linux family, Linux 2.2.26 seems to have the most robust behavior (the smallest proportion of no-signaling) while for Windows family, the differences between OSs are too small to differentiate them clearly.

	PostMark			JVM		
	SEr	SXp	SNS	SEr	SXp	SNS
Linux 2.2.26	67.5%	7.8%	24.8%	65.6%	7.4%	26.9%
Linux 2.4.x	66.0%	7.8%	26.2%	57.1%	9.8%	33.1%
Linux 2.6.6	58.8%	9.7%	31.6%	57.7%	8.8%	33.5%
Windows NT4	27.0%	17.5%	55.5%	24.5%	21.8%	53.6%
Windows 2000	24.5%	20.3%	55.2%	24.8%	22.8%	52.3%
Windows XP	23.1%	20.7%	56.1%	24.7%	22.8%	52.4%

Table 7: Linux and Windows robustness using PostMark and JVM

The reaction times in the presence of faults are globally smaller for Windows family OSs than for Linux family OSs (if we exclude a few systems calls for which the reaction time is exceptionally very long). We have noticed that the execution times of a minority of system calls can have important consequences on the mean reaction time values. For instance, the execution time of mkdir system call for Linux 2.4 biases the Treac measure. High standard deviations on this measure are due to a minority of system calls with very large reaction time compared to the majority. If we discard these exceptional values, Treac and τreac become very close. Moreover, Treac for Windows and Linux families become close.

The restart times in the presence of faults (Tres) are always higher than the restart times without faults (τres). The lowest Tres was observed for Linux 2.2.26 and Windows XP (71 s). For the Windows family, the restart

time is higher in case of workload abort/hang both when using PostMark and JVM as workloads. For Linux family, the standard deviation on this measure is high because of check-disks performed every 26 Target Machine restarts. We have noticed that check-disks take a longer time to perform when using JVM as workload than when using PostMark. The regular restart time is the same both when using PostMark and JVM.

The ordering of the OSs with respect to the restart time is the same for Postmark and JVM except for Windows 2000, for which the restart time is better than Windows NT using JVM while it is worse when using PostMark.

6. Benchmark Properties

In order to gain confidence in dependability benchmark results, one has to check that the key properties are fulfilled. These properties are addressed successively in the rest of this section. We first define the property then we show what has been achieved to satisfy and check it.

6.1. Representativeness

Representativeness concerns the benchmark measures, the workload and the faultload.

6.1.1 Measures

The measures evaluated provide information on the OS state and temporal behavior after execution of corrupted system calls. We emphasize that these measures are of interest to a system developer (or integrator) for selecting the most appropriate OS for his/her own application. Of course other measures would help.

6.1.2 Workload

We have selected two benchmark workloads whose characteristics, in terms of system calls activated, are detailed hereafter. Nevertheless, the selection of any other workload does not affect the concepts and specification of our benchmark.

The PostMark workload activates system calls belonging to functional components: file management, thread management, memory management and system information. Most of system calls belong to the file management functional component (62% for Linux, 48% for Windows). However, a significant amount of system calls belong to the thread management (12% for Linux, 32% for Windows) and to the memory management (19% for Linux, 8% for Windows) functional components. In total, 93% for Linux and 88% system calls for Windows are in these functional components. The PostMark workload is representative if the OS is used as a file server.

Let us stress that we chose the workload to be the JVM and not a particular Java application, to focus on system calls activated by the JVM, regardless of the Java application running on top of it. The JVM is solicited through a very small program. The JVM workload activates system calls fairly distributed with respect to functional components (file management, thread management, memory management, user interface, debugging and handling, inter-process communication). Similarly to the PostMark workload, most of the activated system calls (88% for Linux, 73% for Windows) belong to the file management (40% for Linux, 26% for Windows), the thread management (24% for Linux, 36% for Windows), and to the memory management (24% for Linux, 11% for Windows) functional components.

6.1.3 Faultload

The faultload is without any doubt the most critical component of the OS benchmark and more generally of any dependability benchmark. Faultload representativeness concerns i) the parameter corruption technique used and ii) the set of corrupted parameters.

Parameter corruption technique

In our previous work [Jarboui *et al.* 2002], performed for Linux, we have used two techniques for system call parameter corruption: the *systematic bit-flip* technique consisting in flipping systematically all bits of the target parameters (i.e., flipping the 32 bits of each considered parameter) and the *selective substitution technique* described in Section 2. This work showed the equivalence of the errors induced by the two techniques. In [Kalakech *et al.* 2004b] we obtained the same robustness for Windows 2000 using the systematic bit-flip technique and the selective substitution technique.

The application of the bit-flip technique requires much more experimentation time compared to the application of selective substitution technique. Indeed, in the latter case, the set of values to be substituted is simply determined by the data type of the parameter (see Section 2), which leads to a more focused set of experiments. We have thus preferred the selective substitution technique for pragmatic reasons: it allows derivation of results that are similar to those obtained using the well-known and accepted bit-flip fault injection technique, with much fewer experiments. Our benchmark is based on selective substitutions of system call parameters to be corrupted.

Parameters to be corrupted

The selective substitution technique used is composed of a mix of three corruption techniques as mentioned in Section 2: 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 OORD only (denoted *FL1*), and ii) OORD only (denoted *FL2*). For each workload (PostMark and JVM), 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). Note that 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 the same faultload is equivalent. The same results have been obtained in [Kalakech *et al.* 2004b], using TPC-C Client as workload.

The number of substitutions (hence the number of experiments) decreases significantly when considering FL1 and F2. By way of examples, Table 8 gives the number of experiments for Windows NT4 and Linux 2.4 for PostMark and Figure 18 shows the robustness of Windows NT4, 2000 and XP with respect to FL1 and FL2, for PostMark. (robustness with respect to FL0 is given in Figure 3).

	ID	IA	OORD	# experiments, PostMark Windows NT4	# experiments, PostMark (Linux 2.4)
FL0	x	x	x	418	206
FL1		x	x	331	135
FL2			x	77	55

Table 8: Faultloads considered

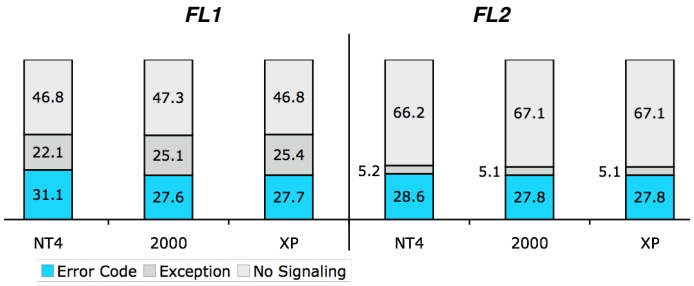

Figure 18: OS Robustness using FL1 and FL2 (%), using PostMark

Further validation concerning selective substitution

For each parameter type class, we performed a sensitivity analysis regarding specific values of parameter substitution. This analysis revealed that different random values chosen to substitute for the original parameter lead to the same outcome of benchmark experiments. Hence the benchmark results are not sensitive to the specific random values given to the corrupted parameters as substitution values.

Moreover, we checked the representativeness of incorrect data faults. One could argue that the OS is not assumed to detect this kind of fault as the substitution values are inside the validity domain of the parameter type. The analysis of the execution traces corresponding to experiments with incorrect data substitution that led to notification of error codes in the case of Linux, revealed that 88.6% of the faults correspond to out-of-range data in the very particular context of the workload execution. Consequently, the notification of error codes was a normal outcome in these cases. Incorrect data are thus very useful: they can provide a practical way for generating out-of-range data in the execution context. Note that an enormous effort would be needed to analyze all execution contexts for all system calls to define pertinent substitution values for each execution context.

6.2. Repeatability and Reproducibility

The benchmarking of a given system can be based either on an existing benchmark implementation (an existing prototype) or on an existing specification only. Repeatability concerns the benchmark prototype while reproducibility is related to the benchmark specification.

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). This property is central to benchmarking. Our OS dependability benchmark is composed of a series of experiments. Each experiment is run after 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. 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 technique used to corrupt system call parameters. This makes us confident about reproducibility. However, more verification is still required.

6.3. 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 workloads. 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.

Let us consider the case of PostMark as an example, for which the first prototype developed was for Windows 2000. It proved to be portable without modification for Windows 2000 Server and Windows 2003 Server (PostMark activates the same 27 system calls with parameters), and with minor adaptations for the others. One system call (FreeEnvironmentStringA) is not activated under Windows NT4, NT4 Server and XP and another system call (LockResource) is not activated under NT4 and NT4 Server. In these cases, the system calls that are not activated are dropped from the substitution values database.

For Linux, the prototype revealed to be portable across all OSs except the interceptor Strace, which is kernel-dependent. Consequently, we used one version of Strace for Linux 2.2 and 2.4 and another version for Linux 2.6. Also, PostMark activates the same system calls for Linux 2.2.26 and 2.4 while it activates a supplementary system call (mmap2) for Linux 2.6.6. Consequently, we added this system call to the set of activated system calls and an entry in the substitution values database.

6.4. Cost

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 implementation of the benchmark itself was not too time consuming.

Let's first consider PostMark, then JVM (which benefited a lot from the PostMark benchmarks as all benchmark components did exist and we had only to adapt them).

For PostMark, the benchmark implementation and running took us less than one month for each OS family, spread as follows:

- The installation of PostMark took one day both for Windows and Linux.
- The implementation of the different components of the controller took about two weeks for each OS family, including the customization of the respective interceptors (Detours and Strace).
- The implementation of the faultload took one week for each OS family, during which we have i) defined the set of substitution values related to each data type and ii) created the database of

substitution values. Both databases are portable on OSs belonging to their family (one database for Windows family and one database for Linux family). However, small adaptations were necessary (see Section 6.3).

- The benchmark execution time for each OS is less than two days.

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, on average, an experiment lasts less than 5 minutes. The series of experiments of a benchmark is fully automated. Hence, the benchmark execution duration ranges from one day for Linux to less than two days for Windows (25-27 system calls are activated by PostMark on Windows, while only 16-17 system calls are activated on Linux).

For JVM, 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 because 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.

7. Conclusion

We have presented the specification of a dependability benchmark for OSs with respect to erroneous parameters in system calls, along with prototypes for two families of OSs, Windows and Linux, and for two workloads. These prototypes allowed us to obtain the benchmark measures defined in the specification. We stress that the measures obtained for the different OSs are comparable as i) the same workloads (PostMark and JVM) were used to activate all OSs, ii) the faultload corresponds to similar selective substitution techniques applied to all system calls activated by the workload and iii) the benchmark conduct was the same for all OSs.

Concerning the robustness measure, the benchmark results show that all OSs of the same family are nearly 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 reported 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, results show that globally Linux reaction time, related to system calls activated by the workload, is longer than Windows reaction time. Refinement of this measure revealed a great variation around the average and that a minority of system calls with large execution times skewed 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, Windows XP and Linux 2.2.26 have the shortest restart times in the presence of faults (71 s). Detailed analysis showed i) a correlation between Windows restart time and the workload final state (in case of workload *hang* or *abort,* the restart time is 10 % higher than in case of workload completion) and ii) that Linux performs a check-disk after each 26 restarts. A restart with a check-disk is three to four times longer than the average.

We validated our benchmark paying 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 all the DBench colleagues who, through the numerous discussions all over the project, helped us in defining the OS benchmark as it is in this paper. In particular, we are grateful to Jean Arlat who contributed to the OS benchmark based on TPC-C Client.

References

[Albinet *et al.* 2004] 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,* (Florence, Italy), pp. 867-876, 2004.

[Arlat *et al.* 2002] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 138-163, February 2002.

[Brown *et al.* 2002] A. Brown, L. C. Chung and D. A. Patterson, "Including the Human Factor in Dependability Benchmarks", in *Proc. of the 2002 DSN Workshop on Dependability Benchmarking,* (Washington, DC, USA), 2002.

[Brown & Patterson 2000] A. Brown and D. A. Patterson, "Towards Availability Benchmarks: A Cases Study of Software RAID Systems", in *Proc. 2000 USENIX Annual Technical Conference,* (San Diego, CA, USA), USENIX Association, 2000.

[Chevochot & Puaut 2001] P. Chevochot and I. Puaut, "Experimental Evaluation of the Fail-Silent Behavior of a Distributed Real-Time Run-Time Support Built from COTS Components", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), pp. 304-313, IEEE CS Press, 2001.

[Chou *et al.* 2001] 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), pp. 73-88, ACM Press, 2001.

[DBench] http://www.laas.fr/DBench, Project Reports section, project short final report.

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

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

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

[Jarboui *et al.* 2002] 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 *Proc. 2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC-2002),* (Tsukuba City, Ibaraki, Japan), pp. 51-58, IEEE CS Press, 2002.

[Kalakech *et al.* 2004a] A. Kalakech, T. Jarboui, J. Arlat, Y. Crouzet and K. Kanoun, "Benchmarking Operating System Dependability: Windows 2000 as a case study", in *Proc. 2004 Pacific Rim Int. Symp. on Dependable Computing (PRDC-2004),* (Papeete, Tahiti), pp. 261-270, 2004.

[Kalakech *et al.* 2004b] A. Kalakech, K. Kanoun, Y. Crouzet and J. Arlat, "Benchmarking the Dependability of Windows NT, 2000 and XP", in *Proc. Int. Conf. on Dependable Systems and Networks (DSN-2004),* (Florence, Italy), pp. 681-686, 2004.

[Kanoun *et al.* 2005] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina and P. Rumeau, "Benchmarking the Dependability of Windows and Linux using Postmark workloads", in *Proc. 16th Int. Symposium on Software Reliability Engineering (ISSRE-2006),* (Chicago, USA), 2005.

[Kanoun & Crouzet 2006] K. Kanoun and Y. Crouret, "Dependability Benchmarking for Operating Systems", in *International Journal of Performance Engineering,* vol. 2, no. 3, pp. 275-287, 2006.

[Katcher 1997] J. Katcher, *PostMark: A New File System Benchmark*, Network Appliance, www.netapp.com/tech_library/3022.html, N°3022, 1997.

[Koopman & DeVale 1999] P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems", in *Proc. 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29),* (Madison, WI, USA), pp. 30-37, IEEE CS Press, 1999.

[Koopman *et al.* 1997] 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), pp. 72-79, IEEE CS Press, 1997.

[Lindholm & Yellin 1999] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* Addison-Wesley Professional, 1999.

[McGrath & Akkerman 2004] R. McGrath and W. Akkerman, *Source Forge Strace Project*, http://sourceforge.net/projects/strace/, 2004.

[Mukherjee & Siewiorek 1997] A. Mukherjee and D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Transactions of Software Engineering*, vol. 23 no. 6, pp. 366-376, 1997.

[Shelton *et al.* 2000] C. Shelton, P. Koopman and K. D. Vale, "Robustness Testing of the Microsoft Win32 API", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp. 261-270, IEEE CS Press, 2000.

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

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

[Zhu et al. 2003] Ji J. Zhu, J. Mauro, and I. Pramanick, "Robustness Benchmarking for Hardware Maintenance Events", in Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003), pp. 115-122, San Francisco, CA, USA, IEEE CS Press, 2003.

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

[IEEE 1990] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12–1990, IEEE Computer Soc., Dec. 10, 1990.