Chapter 2

Dependability Benchmark for General Purpose Operating Systems

Abstract

This chapter presents a dependability benchmark for general-purpose operating systems (OSs). The benchmark is defined through the specifications of its main components. The specifications are implemented in the form of a benchmark prototype. The important novelty, as regards OS dependability benchmarking, is threefold. First, it lies on a comprehensive and structured set of measures: outcomes are considered both at the OS level and at the application level. Second, these measures include not only robustness measures (e.g., the distribution of the observed outcomes for the OS and the application), but also related temporal measures in the presence of faults (e.g., OS reaction time and restart time). Finally, we are using a realistic workload (namely, TPC-C client), instead of a synthetic workload.

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

2.1. Introduction

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

The aim of our OS dependability benchmark is to provide results that objectively i) characterize qualitatively and quantitatively the OS behaviour in the presence of faults and ii) evaluate performance-related measures in the presence of faults. These results can help a system integrator in selecting the most appropriate OS, based on the benchmark measures evaluated, in complement to other criteria (e.g., performance, maintenance, etc.). The benchmark is entirely based on experimentation on the OS.

Several relevant attempts were already proposed to help characterize the failure modes and robustness of software executives. A comprehensive analysis of the issues linking robustness and dependability can be found in [Mukherjee and Siewiorek 1997]. The executives targeted in these studies encompass real time microkernels [Chevochot and Puaut 2001; Arlat et al. 2002], general purpose OSs [Tsai et al. 1996; Koopman and DeVale 1999], as well as CORBA middleware implementations [Pan et al. 2001; Marsden et al. 2002]. Results concerning the robustness with respect to faults in device drivers can be found in [Durães and Madeira 2002] and in [Albinet et al 2004]. The work reported in [Shelton et al. 2000] specifically addressed the robustness of the Win32 API as is the case for the proposed benchmark prototype.

The important novelty, as regards OS dependability benchmarking, that is provided by the work reported here is threefold. First, it lies on a comprehensive and structured set of measures: outcomes are considered both at the OS level and at the application level. Second, these measures include not only robustness measures (e.g., the distribution of the observed outcomes for the OS and the application), but also related temporal measures in the presence of faults (e.g., OS reaction time and restart time). Finally, we are using a realistic workload (namely, TPC-C client), instead of a synthetic workload.

The remainder of the chapter is organized as follows. Section 2.2 gives an overview of the benchmark. Sections 2.3 and 2.4 specify respectively the benchmark measures and the experimental dimensions. Section 2.5 describes a particular prototype for Windows family. Section 2.6 presents some results related to the comparison of three OSs namely Windows NT4, Windows 2000 and Windows XP, obtained using this prototype. Section 2.7 is devoted to the validation of the benchmark while Section 2.8 concludes the chapter. Annex 2-A provides sensitivity analysis related to the faultload selection and Annex 2-B shows how the benchmark measures can be refined, based on complementary measures.

2.2. Dependability Benchmark Overview

An OS can be viewed as a generic software layer that virtualises and manages all aspects of the underlying hardware. The OS provides i) basic services to the applications, through an Application Programming Interface, API, (e.g., Win32 for Windows OSs), and ii) communication with peripherals devices via device drivers.

For an OS dependability benchmark, the Benchmark Target (BT) corresponds to the OS. However, for the BT to be assessed, it is necessary to run it on top of a hardware platform and to use a set of libraries such as device drivers. Thus, the BT along with the hardware platform and libraries necessary to run it under the benchmark execution profile form the System Under Benchmarking (SUB). Although, in practice, the benchmark results obtained characterise the SUB (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 characterise the OS.

The OS dependability benchmark presented in this chapter is a robustness benchmark [Kalakech et al. 2004]. Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. Robustness can thus be viewed as an indication on the OS capacity to resist/react to faults induced by the applications running on top of it, or originating from the hardware layer or from device drivers. We put emphasis on the OS robustness as regards application erroneous behaviours, more precisely, with respect to possible erroneous system calls provided by the application software to the OS via the API. These erroneous system calls result from corrupted parameters, including invalid parameters, in system calls. It is worth mentioning that for sake of conciseness such erroneous system calls are shortly referred to as *faults* in this chapter.

The proposed 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. The benchmark results are aimed at significantly improving her/his knowledge about its behaviour in presence of faults and comparing alternative OSs. In practice, the user may well be the developer (or the integrator) of a system including the target OS as a COTS component. What is important is that the OS is considered as a “black box” and accordingly, 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).

We specify a full set of benchmark measures. Some of them characterize the OS behaviour in the strict sense, while some others provide information on the impact of the fault as perceived by the executed workload. As users may be only interested in characterizing the OS behaviour in the strict sense, the benchmark implementation should make it possible to simply ignore some measures and thus not to record the associated measurements.

The benchmark is defined so that the workload executed on the OS could be any performance benchmark workload (and, more generally, any user tailored application) intended to run on

top of the benchmarked OS. The specification and the prototype example we have developed use TPC-C client [TPC-C 2002] as a privileged workload.

The specifications of the benchmark must define clearly the following items:

1) The benchmark measures to be evaluated.

2) The benchmark experimental dimensions (i.e., the workload, faultload and the measurements to be performed on the system).

3) The set-up and related implementation issues required for developing and running a benchmark prototype.

It is clear that these items are not totally independent from each other. However, for practical reasons, in the subsequent sections, we address them separately: Sections 2.3 and 2.4 will cover the specifications of the measures and experimental dimensions, while the benchmark prototype implemented according to these specifications will be described in Section 2.5.

2.3. Benchmark Measures

Erroneous system calls are provided to the OS through the API. Their consequences on the OS behaviour are deduced from the OS outcomes that are complemented by the workload outcomes. Measures defined from the OS outcomes characterise the OS behaviour in the strict sense, while the measures at the workload level provide information on the impact of the OS behaviour on the state of the application. Measures defined from both outcomes allow for a comprehensive description of the consequences of erroneous system calls.

We will first define the OS benchmark measures then address measures related to the combined behaviour of the OS and the workload.

For each of them, two classes of measures are considered: i) robustness measures, concerning the qualitative and quantitative behaviour in the presence of faults and ii) temporal measures in the presence of faults standing for OS reaction time, system restart time and workload execution time.

2.3.1. OS Measures

We first define the states of the benchmarked OS after execution of the corrupted system call. Then, we define the related robustness measure before describing the temporal measures.

After execution of a corrupted input, the OS is in one of the states defined in Table 2.1.

- **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 application is automatically aborted by the OS. 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 the hang state, a hard reboot of the OS is required.
- **SNS**: In the no-signalling state, the OS does not detect the presence of the erroneous parameter. As a consequence, it accepts the erroneous system call and executes it. It may thus abort, hang or respond to the application whenever such a response is expected from it. However, the response might be erroneous or correct. For some system calls, the application may not require any explicit response, so it simply resumes execution after issuing the system call.

  SNS can be characterized by the fact that none of the previous outcomes (SEr, SXp, SPc, SHg) is observed.

Table 2.1: OS outcomes

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

Remarks

- *Panic* and *hang* outcomes are actual states, as they can last for a while. Conversely, SEr and SXp characterize only events. They are easily identified when an error code or a user exception notification are provided by the OS.
- It could happen that several error codes or/and exceptions are generated successively during a single run of the workload. Accordingly several options can be considered for categorizing these runs [Rodríguez et al. 2002]. In the proposed benchmark, the outcomes are categorized according to the first event. If needed, further refinement of the outcomes can be made according to the actual error or exception reported.

2.3.1.1. OS Robustness

A benchmark campaign is composed of a series of independent runs (referred to as experiments). Each run consists in executing the workload with a corrupted system call. The

robustness measure, **POS**, is defined as the percentages of experiments leading to any of the outcomes "**S***" in Table 2.1.

The OS robustness, **POS**, is thus a vector composed of 5 elements.

2.3.1.2. OS Reaction Time

This measure corresponds to the time for the OS to react to a system call in presence of faults, either by notifying an exception or by issuing an error code or by executing the required instructions (the result could be correct or not)..

Also, the response time in the presence of faults can be evaluated with respect to each outcome of Table 2.1. Let these times be denoted respectively t_{SEr}, t_{SXp} and t_{SNS}. They are counted from the instant where the modified system call is provided to the OS, up to the issue of the corresponding event by the OS, i) an error code return, ii) an exception notification, and iii) a response to a system call.

Let $\mathbf{t_{exec}}$ be the reaction time in the presence of fault and $\boldsymbol{\tau_{exec}}$ the average reaction time in absence of faults.

2.3.1.3. OS Restart Time

The duration of a restart is a very important measure for the designers of critical applications because during this time the system is unavailable. Although under nominal operation, the OS restart time is almost deterministic, it may significantly 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 being applied.

Let $\mathbf{t_{res}}$ be the restart time in the presence of fault and $\boldsymbol{\tau_{res}}$ the average restart time in absence of faults.

2.3.2. Workload Measures and Comprehensive Combined Measures

Observation of the workload final state helps identifying the impact of the OS on the workload. We first define the states in which the workload could be after execution of the corrupted system call, and then define the OS and workload combined states. After that, we define workload benchmark measures.

The workload is characterized by one of the following outcomes: i) the workload completes with correct results, ii) it completes with erroneous results, iii) the workload is aborted, or iv) the workload hangs. These outcomes are summarized in Table 2.2. Let WC refer to the case where the workload is in a completion state ($WC = WCC \cup WEC$), being it correct or erroneous.

Clearly, the workload could end up in any of the four possible states of Table 2.2 irrespective of the outcomes SEr, SXp or SNS that can characterize the state of the OS. Conversely,

whenever the outcome SPc (*Panic*) is observed, this can only lead the workload to abort or hang, while an OS *Hang* leads necessarily the workload to hang.

Table 2.3 presents the possible comprehensive states combining OS and workload states denoted (S* – W*). The most critical situation corresponds to the cases when the OS is in a No Signalling state while the workload is an erroneous completion state, SNS – WEC.

Table 2.2: Possible workload outcomes

WCC	Correct completion
WEC	Erroneous completion
WAb	Abort
WHg	Hang

Table 2.3: Possible combined outcomes

SUB (OS) → / ↓ Workload	Error code	Exception	Panic	Hang	No signalling
Correct completion	SEr – WCC	SXp – WCC	—	—	SNS – WCC
Erroneous completion	SEr – WEC	SXp – WEC	—	—	SNS – WEC
Abort	SEr – WAb	SXp – WAb	SPc – WAb	—	SNS – WAb
Hang	SEr – WHg	SXp – WHg	SPc – WHg	SHg – WHg	SNS – WHg

The OS robustness measure, **POS**, can be refined using the distribution of the percentages of the workload outcomes when the OS is in state SNS. Let **PSNS** denote the associated dimension-4 vector. For sake of simplicity, PSNS is referred to as the *workload robustness*.

The workload temporal measure of interest is the *duration of workload completion*, irrespective of the state of the SUB. Let $\mathbf{t_{WC}}$ denote the time required for the workload to reach state WC in the presence of fault and $\boldsymbol{\tau_{WC}}$ the workload average execution time in absence of faults.

2.3.3. Summary of Measures

Table 2.4 summarises the two robustness measures associated respectively to the OS being considered alone, and the OS combined with the workload.

The temporal measures are evaluated as average times over all experiments categorized by a specific outcome. However, standard deviation, maximum and minimum values are also of prime interest.

Let T_{exec}, T_{SEr}, T_{SXp}, T_{SNS}, T_{res} and T_{WC} denote respectively the average times for the previously identified times.

Table 2.5 recapitulates these temporal measures.

Table 2.4: Robustness measures

Measure	Definition
POS	OS robustness Behaviour of the OS — vector of 5 elements
PSNS	Workload robustness Behaviour of the workload when the OS is in SNS— vector of 4 elements

Table 2.5: Temporal benchmark measures

Reaction time

τ_{exec}	Time for the OS to execute a system call, in the absence of faults
T_{exec}	Time for the OS to execute a system call, in the presence of faults
T_{SEr}	Time for the OS to return an error code in the presence of faults
T_{SXp}	Time for the OS to notify an exception in the presence of faults
T_{SNS}	System call execution time when the OS is in state SNS, in the presence of faults

Restart time

τ_{res}	Duration of OS restart in the absence of faults
T_{res}	Duration of OS restart in the presence of faults

Workload execution time

τ_{WC}	Duration of the workload execution in the absence of faults
T_{WC}	Duration of the workload execution in the presence of faults

2.3.4. Basic and Complementary Measures

Performance benchmarks usually evaluate only one or two measures. It can be considered that our benchmark evaluates two sets of measures:

- The first set is composed of POS, T_{exec} and T_{res}.
- The second set corresponds to the other measures of Tables 2.4 and 2.5.

From our point of view, the first set constitutes the *basic set of measures* for characterizing the OS in the strict sense. The second set includes measures that are not mandatory and can be considered as *complementary measures*. Complementary measures concern either the OS (T_{SEr}, T_{SXp}, T_{SNS}), they can be used to provide more refined information on its behaviour, or the workload measures (PSNS,T_{WC}).

Specific benchmark implementations may simply ignore measures that are of less interest (from the basic or complementary set) and the associated information does not need to be

recorded. However, it is worth to mention that, except for PSNS that requires additional analyses, all the other measures in Tables 2.2 and 2.3 do not require any specific analysis and effort to be implemented as we will show in the next section.

2.4. Experimental Dimensions

In the case of performance benchmarks, the benchmark execution profile is simply a workload that is as realistic and representative as possible for the system under test. For our general-purpose OS dependability benchmark, the execution profile includes in addition corrupted parameters in system calls. The set of corrupted parameters is referred to as the faultload. We aim at modifying the parameters of the system calls activated by the workload to simulate the erroneous parameter values that the application processes could communicate to the OS.

From a practical point of view, the faultload can be either integrated within the workload (i.e., the faults are embedded in the program being executed) or provided in a separate module. For enhanced flexibility, we made the latter choice: the workload and the faultload are implemented separately, which allows for a better portability of the faultload. As a consequence: i) the same faultload can be applied with different workloads, and ii) any available performance benchmark workload can be used.

In our benchmark, we use the workload of TPC-C client, but we do not use the performance measures specified by TPC-C as they are far from being suitable to characterize the behaviour of an OS.

In the sequel, we concentrate on the specification of the faultload and more precisely on the technique for corrupting the system call parameters, and the selection of the set of system calls to be corrupted. Then we address the measurements to be performed on the system during the experiments in order to derive the benchmark measures (basic and complementary measures).

2.4.1. Parameter Corruption Technique

We use a parameter corruption technique similar to the one used in [Koopman et al. 1997], relying on a thorough analysis of system call parameters to define *selective substitutions* to be applied to these parameters. 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 can be 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.

Annex 2-A provides some comparative results intended to explain and justify our choice of the retained parameter corruption technique.

To reduce the number of experiments, the parameter data types are grouped into classes. A set of values is defined for each class. They depend on the definition of the class. For example, for Windows**,** we have grouped 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. Table 2.6 reviews the substitution values associated with the most used data type classes.

Table 2.6: Parameter substitution values per data type class

Data type class	Substitution values				
pvoid	NULL	0xFFFFFFFF	1	0xFFFF	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)		

2.4.2. System Calls to Be Corrupted

Ideally, and without any time limitation, all system calls used in the workload with parameters should be corrupted. For small workloads this might be possible. However, for workloads such TPC-C client, more than 100 system calls are involved, with several occurrences in the program, this would require several weeks of experimentation. In addition, all system calls are not necessarily interesting to be corrupted. For practical reasons, one has to target a subset of system calls. This selection depends on the nature of system calls that are worth to corrupt and the accepted benchmark experimentation duration. The benchmark duration depends on the duration of each experiment and on the number of experiments to be performed. The latter depends on i) the number of system calls to corrupt, ii) the number of parameters to corrupt in a system call and iii) the number of substitution values associated to each parameter. From our experience on Linux and Windows, an experiment lasts on average less than 5 minutes. Using a fully automated benchmark set-up, approximately 1400 experiments can be run in 5 days. This leads to consider 40 to 60 system calls to be corrupted (depending on the number of parameter substitutions) for a 5-day fully automated benchmark execution.

The first step in choosing system calls whose parameters will be corrupted consists in identifying all system calls used in the workload as well as their occurrences. In order to insure *portability* of the faultload, our recommendation is to use the criticality of OS functional components as a selection criterion. Selection of system calls associated to specific functional components facilitates comparison between OSs that are not from the same family (e.g, with distinct APIs). Indeed, even though OSs do not necessarily comprise exactly the same system calls, most OSs feature comparable functional components.

Our benchmark targets the following functional components that we have identified as the most critical for a general-purpose OS: *Processes and Threads*, *File Input/Output*, *Memory Management* and *Configuration Manager*.

2.4.3. Measurements

The number of substitutions determines the number of experiments. The experiments are achieved independently and the system is restarted after each experiment. To assess the benchmark measures defined in Tables 2.4 and 2.5, several measurements have to be performed on the system during each experiment, related to the OS and to the workload states and temporal behaviour.

At the SUB level, after each experiment run:

- The state of the OS is to be recorded. In states SEr, SXp and SNS, the OS is alive and delivers explicit information to the workload. These states can be recorded easily, along the time required to the OS to provide the information to the workload (respectively t_{SEr}, t_{SXp} and t_{SNS}). The panic and hang states can only be reliably diagnosed and recorded by a remote machine (refered to as the Benchmark Controller — see Section 2.5). The latter is part of the Benchmark Management system.
- The system restart duration time t_{res} is measured.

Hence for each experiment, the OS state and restart duration are to be recorded. In addition, states SEr, SXp and SNS, the associated times can optionally be recorded to evaluate more refined measures.

After running the whole set of experiments, POS (the dimension-5 OS robustness vector) and T_{res} (the average restart time) can be evaluated, as well as the average OS reaction times T_{exec}, T_{SEr}, T_{SXp} and T_{SNS}.

At the workload level, after each experiment run, the workload state is recorded as well as the workload completion time. After running the whole set of experiments, the workload robustness P_{SNS} (the dimension-4 vector for combined benchmark measures), and the (average) workload execution time T_{WC} are evaluated.

2.5. Benchmark Prototype

In this section, we present the dependability benchmark prototype that was implemented for Windows family.

First, we describe the system under benchmark and the execution profile defined. Then, we detail how the operating system benchmark specifications presented in the previous sections was implemented.

2.5.1. Systems Under Benchmark

The considered SUB is made up of the operating system running on x86 hardware. In the architecture of Windows, the application processes call OS services through one or more environment subsystem Dynamic Link Libraries (DLLs). The role of an environment subsystem is to expose a subset of the base Windows 2000 executive system services to application programs. The Win32 environment subsystem DLLs (such as kernel32.dll,

Advapi32.dll, User32.dll, and Gdi.dll) implements the Win32 API functions. Although Windows 2000 was designed to support multiple programming interfaces, Win32 is its primary and preferred interface. Moreover, Windows 2000 cannot run without the Win32 environment subsystem [Solomon and Russinovich 2000]. Consequently, this is the considered interface on which the faultload is applied.

2.5.2. Execution Profile

The TPC-C client implementation used in the current benchmark set-up is the same as the one used by other DBench partners (see e.g., [Vieira and Madeira 2003] and Chapter 5). The workload of TPC-C client activates 132 system calls (with parameters). The considered functional components (Processes and Threads, File Input/Output, Memory Management and Configuration Manager*)* use 28 system calls, for which 75 parameters have been corrupted leading to run 552 experiments using the benchmark experimental set-up presented hereafter. Annex 2-A shows that for the three OSs considered, the selected system calls lead to the same benchmark comparison result as when considering all the 132 system calls of TPC-C client.

2.5.3. Experimental Set-up and Benchmark Conduct

As it was already pointed out, since perturbing the operating system may lead the OS to hang, a remote machine is required to reliably control the benchmark experiments. Accordingly, for running an OS dependability benchmark we need at least two computers: i) the Target Machine (TM) for hosting the benchmarked OS and the workload, and ii) the Benchmark Controller (BC) that is primarily in charge of diagnosing and collecting data in case of a hang or an abort (see Tables 2.1 and 2.2). Furthermore, as we are using a TPC-C client as workload in the TM, the Data Base Management System (DBMS) is needed to process the TPC-C client requests. We used a third machine with Oracle DBMS for that purpose.

Figure 2.1 illustrates the various components that characterize the proposed OS dependability benchmark prototype. These components are executed on the TM that is running the benchmarked OS, and on the remote BC machine. The two machines are connected via an Ethernet network.

The Benchmark Management System referred to in Chapter 1 is therefore composed of the benchmark controller, the interceptor and the Data Base Management System (DBMS).

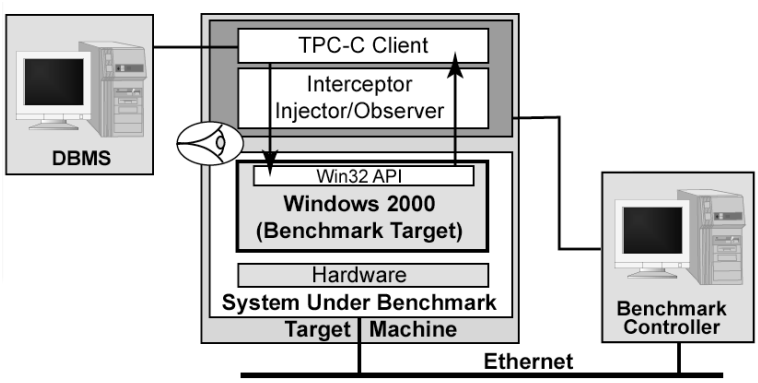

Figure 2.1. Experimental set-up

To intercept the Win32 functions (i.e., system calls), we have modified the "Detours" tool [Hunt and Brubaher 1999], a library for intercepting arbitrary Win32 system calls on x86 machines. This modification was made to facilitate replacement of system call parameters by substitution values. Also, we have added several modules in the library to observe the reactions of the OS after parameter substitution, and to retrieve the required measurements.

At the beginning of each experiment, the target machine records the experiment start instant $t_{ExpStart}$ and sends it to the BC along with a notification of experiment start-up. The workload starts its execution. The *Observer* module of the Interceptor records the start-up instant of the workload t_{WStart}, the activated system calls and their resultant responses in the experiment execution trace. This trace collects also 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 *Injector* module of the Interceptor checks whether the current system call is the system call in which a parameter is to be corrupted. If this is not the case, the execution is simply resumed. Otherwise, the execution is interrupted, a parameter value is substituted and then the execution is resumed. At the end of the execution of the TPC-C client, the SUB notifies the BC of the end of the experiment by sending an end signal along with the experiment end instant, t_{ExpEnd}.

When the workload does not complete (e.g., in case of a hang), then t_{ExpEnd} is mainly governed by the value of a watchdog timer that controls the monitoring of the workload execution.

The experiment steps are illustrated in Figure 2.2-a in case of workload completion. Figure 2.2-b shows the experiment steps in case of workload Ab/Hg (i.e., without workload completion).

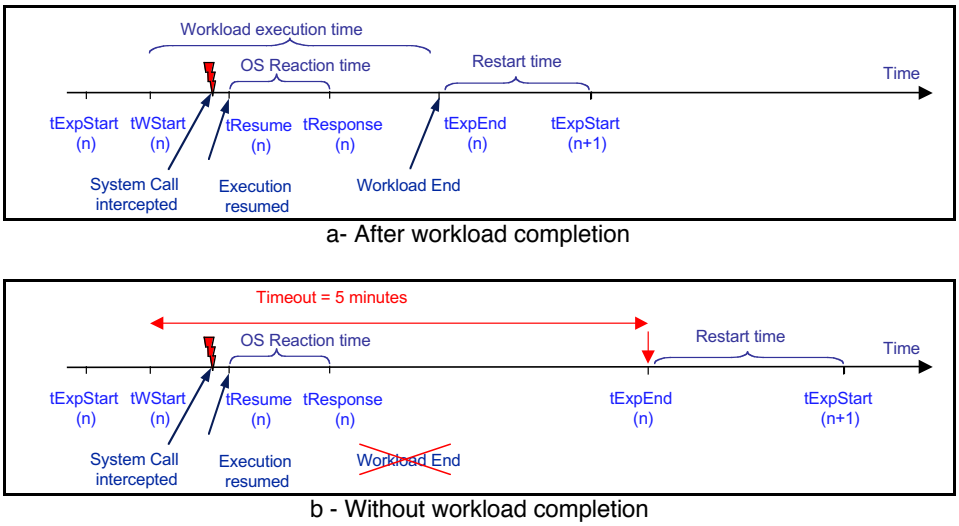

Figure 2.2. Benchmark execution sequence and temporal measures

Each experiment requires the following steps:

- The system call is intercepted and interrupted.
- A parameter is replaced by a substitution value.
- The execution of the intercepted system call is resumed and the instant of resumption tResume is saved in the experiment execution trace.
- The state of the OS is monitored so as to diagnose one of the possible outcomes (SEr, SXp, SNS). The corresponding OS response time (tResponse) is recorded in the experiment execution trace.
- The system is restarted.

For each run, the OS reaction time (either tSEr, tSXp or tSNS) is calculated as the difference between tResponse and tResume.

The BC collects the SHg state of the SUB and WAb and WHg states of the workload. It is in charge of restarting the SUB in such occurrences. The average time necessary for the OS to execute the TPC-C client is about 70 seconds when no faultload is applied. We have currently fixed a maximum delay of 5 minutes (counted from the instant when the BC receives the signal of the start of the execution, tWStart), during which the workload is expected to be executed. If at the end of this delay the BC has not received the end signal from the OS, it then attempts to connect to the OS. If this connection is successful, then a workload abort/hang state is diagnosed, otherwise SHg is assumed. Thus tExpEnd specifies either the completion of the workload or the triggering of the watchdog timer. In case of workload completion, the workload execution time (see Table 2.5) is calculated with reference to the launch of the workload at time tWStart.

Finally, the OS restart time for experiment n-1 is determined as the difference between tExpStart of experiment n and tExpEnd of experiment n-1.

2.6. Results

The benchmark and the prototype defined in the previous sections are used to compare the behaviour of Windows NT4, 2000 and XP [Kalakech et al. 2004a]. Let us recall that the faultload used in the benchmark addresses the 28 system calls invoked by the four Windows functions that we have identified as the most critical ones. For these system calls, 75 parameters have been corrupted in several ways, leading to 552 corrupted system calls. Hence the number of experiments in the benchmark is 552.

In this section, we present the results related to the basic set of benchmark measures defined in Section 2.3.4 (OS robustness, OS reaction and restart times). These measures give information on the global behaviour of the OSs. Annex 2-B shows how they can be refined taking into account the complementary set of measures defined in Section 2.3.4.

2.6.1 OS Robustness

The OS robustness is given in Figure 2.3. No panic and hang states were observed for the three OSs. Exceptions have been notified in 11.4 % to 12 % of the cases, while the number of experiments with error code return varies between 31.2 % and 34.1 %. More than half of the experiments lead to a no signalling outcome. Figure 2.3 shows a *similar behaviour* for the three OSs with respect to robustness (the larger difference is observed for the no signalling case, it is less than 3%).

It could be argued that robustness is sensitive to the parameter corruption technique used as well as to the system calls selected. We have made a sensitivity analysis with respect to these two elements. This analysis confirmed the equivalence of the three OSs with respect to the OS robustness measure (see Annex 2-A).

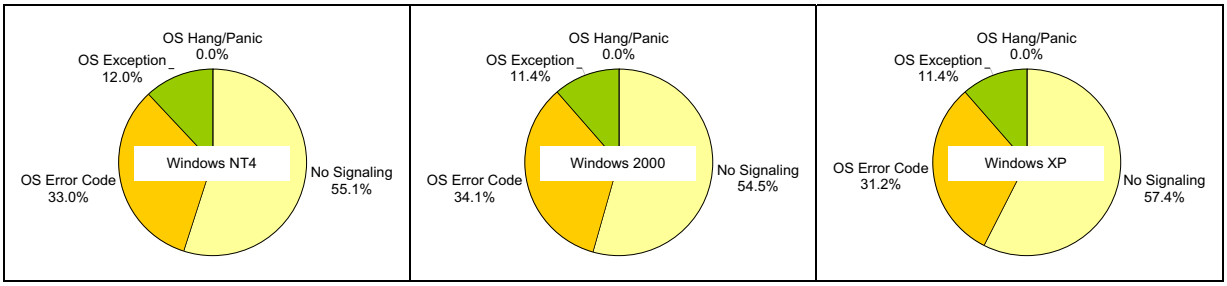

Figure 2.3: OS Robustness measure

2.6.2 OS Reaction Time

The OS reaction time in the absence of faults, τ_{exec}, is evaluated as the average reaction times of the 28 selected system calls whose parameters are being corrupted for the experiments. Table 2.7 shows that, in absence of fault, the three OSs have different reaction times.

The OS reaction time, Texec, corresponds to the average reaction time of the selected 28 system calls in the presence of faults. Table 2.7 shows that the shortest time is obtained for Windows XP while the longest one corresponds to Windows 2000. For Windows XP, this time is slightly larger than the reaction time in absence of faults while it is significantly lower for the two others. This may be explained by the fact that in about 45% of cases the OS detects the injected fault. It does not execute the faulted system call and returns an error code or notifies an exception. The standard deviation is significantly larger than the average for the three OSs. Annex 2-B will provide more detailed information to explain the various behaviours.

Table 2.7: OS reaction time

	Windows NT4		Windows 2000		Windows XP	
	Average	St. deviation	Average	St. deviation	Average	St. deviation
τexec	344 µs		1782 µs		111 µs	
Texec	128 µs	230 µs	1241 µs	3359 µs	114 µs	176 µs

2.6.3 System Restart Time

The system restart time is given in Table 2.8 which shows that Windows XP restart time is 70% of that of Windows 2000, without fault and 73% of this time in the presence of faults. For all systems, the restart time is only few seconds larger than without faults. The standard deviations are small.

Table 2.8: System restart time

	Windows NT4		Windows 2000		Windows XP	
	Average	St. deviation	Average	St. deviation	Average	St. deviation
τres	92 s		105 s		74 s	
Tres	96 s	4 s	109 s	8 s	80 s	8 s

2.6.4 Result Summary

The above results suggest that Windows XP is equivalent to Windows NT4 and Windows 2000 from the robustness point of view, but has shorter reaction time as well as shorter restart times, both with and without faults.

The results related to time measures are in conformance with the statement provided by Microsoft. When Windows XP is installed, the following message is displayed:

"Your Computer will be faster and more reliable[1]

[1] It is worth to mention that the term "reliability" as defined in the above statement is different from the robustness measure evaluated in our dependability benchmark.

Windows XP professional not only starts faster than any previous version, but it also runs your programs more quickly and reliably than ever. If a program becomes unstable, you can close it without having to shutdown Windows".

2.6.5. Time Needed to Develop and to Run the Benchmark

Developing and running an OS dependability benchmark 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, working on faultload definition and studying its representativeness.

The implementation of the benchmark itself was not too much time consuming. It took us less than one month, spread as follows:

- The installation of the TPC-C client took three days (as stated earlier, the implementation used in the current set-up is the same as the one used in Chapter 5).
- The implementation of the different components of the controller took about two weeks, including the customisation of the “Detours” tool.
- The implementation of the faultload took one week, during which we have i) defined the set of the values related to the 28 system calls with their 75 parameters to be corrupted and ii) created the database of the corrupted values. The same database is used for the three Windows OSs.

The benchmark execution time for each OS is two days.

Indeed, the duration of an experiment with workload completion is less than 3 minutes (including the time to workload completion and the restart time), while it is about 7 minutes without workload completion (including the workload watchdog timeout of 5 minutes and the restart time). Thus, on average, an experiment lasts less than 5 minutes. Let us recall that 552 experiments have been performed for each OS (Cf. Section 2.5.2). There experiments are fully automated. The whole benchmark execution duration is thus about 46h for each OS.

2.7. Benchmark Validation

The OS dependability benchmark specified, implemented and used in the previous sections is a robustness benchmark with respect to erroneous system calls sent by the application software to the OS. The execution profile has been defined in such a way that the workload and the faultload are implemented as separate modules. The faultload consists in substituting the correct values of system call parameters by erroneous values using an interceptor module. The latter runs between the application layer and the OS API layer. The results obtained for the three Windows OSs do not contradict the *a priori* common knowledge about the three OSs. As the OSs belong to the same family, it is not surprising that their robustness with respect to the selected faultload is similar. On the other hand, the OS reaction time and restart time show that Windows XP is faster than Windows NT and Windows 2000 and do not contradict the information provided by Microsoft.

Of course, it will be interesting to use another workload to check the validity of the obtained results. In particular, OS performance benchmarks, such as *lmBench* [McVoy and Staelin 1996], that is based on workloads specifically dedicated to general purpose OSs, are good candidates and could be investigated.

Moreover, the validation of the concepts requires implementation of the benchmark specification to another OS family. We are currently modifying the developed prototype to benchmark Linux.

In addition to the above considerations, validation of the benchmark specifications and implementation should address the properties stated in Chapter 1: representativeness, repeatability and reproducibility, portability, scalability, and non-intrusiveness. These properties are addressed successively in the rest of this section.

2.7.1. Representativeness

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

Regarding measures, we emphasize that the three basic measures evaluated in our OS dependability benchmark are of interest to a system developer (or integrator) for selecting the most appropriate OS for his/her own application. The basic set gives information on the OS state and temporal behaviour (reaction and restart times) after execution of a corrupted system call. The three basic measures can be refined using information of the workload state (this is illustrated in Annex 2-B).

We have selected a workload that is well known and commonly used for transactional systems. However, as stated earlier, other workloads should be used to check the validity of the results. Nevertheless, the selection of any other workload does not affect the concepts and specifications of our benchmark. We expect to investigate other workloads. Indeed, we preferred putting emphasis on faultload representativeness as shown herefater.

The faultload is without any doubt the most critical dimension of the OS benchmark and more generally of any dependability benchmark. In our work [Jarboui et al. 2002], we have

used two techniques for system call parameter corruption: the *bit-flip* technique consisting in flipping systematically all bits of the target parameters (i.e., flipping the 32 bits of each parameter considered) and the *selective substitution technique* described in Section 2.4.1. This work was performed for Linux and allowed us to conclude the equivalence of the errors provoked by the two techniques. The application of the bit-flip technique requires much more recurrent time (i.e., experimentation time) compared to the application of selective substitution technique. Indeed, in the latter case, as shown in Section 2.4.1, the set of values to be substituted is simply determined by the data type of the parameter. Therefore, this set leads to a more focused set of experiments. Additionally, Annex 2-A shows that the benchmark results obtained using the selective substitution technique are very similar to those obtained using the bit-flip for Windows 2000, as well.

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-know and accepted bit-flip fault injection technique, with much less experiments.

As a consequence, the benchmark presented in this chapter is based on selective substitutions of system call parameters to be corrupted.

2.7.2. Repeatability and Reproducibility

An 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 our first benchmarks three times to check for repeatability.

We have not checked explicitly and directly the reproducibility of the benchmark results. However, the comparison results obtained for fault representativeness increases our confidence in reproducibility. The benchmark comparison results seem to be independent from the technique used to corrupt system call parameters. Also, the results seem to be not affected by the system calls involved. This makes us confident about reproducibility. However, more verification is still required.

2.7.3. Portability

Portability concerns essentially the faultload (i.e., it applicability to other OS faminlies).

At the specification level, in order to insure portability of the faultload, the system calls to be corrupted are not defined by name. They are specified with respect to the criticality of OS functional components, because OSs from different families do not necessarily comprise exactly the same system calls. They may have different APIs. However, most OSs feature comparable functional components.

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

Our prototype is portable across the Windows OS family that shares the Win32 API used in our experiments.

2.7.4. Non-intrusiveness

The corrupted parameters are inserted instead of the correct ones, without introducing any modification in the target kernel. However the library has been modified in two ways: i) interception of Win32 functions required the use the Detours Library and ii) the observation of the OS reaction required additional modules in the Library.

2.7.5. Scalability

Usually, the functionalities of general purpose OSs are comparable, leading most likely to comparable numbers of system calls whose parameters are to be corrupted (or at least of the same order of magnitude). If it happens that more system calls are involved for some OSs, the concepts of the OS benchmark remain unchanged but more experiments may be required. It is worth to recall that an experiment lasts on average less than 5 minutes. Scalability is thus not a real problem in this case.

2.8. Conclusion

In this chapter we have presented the specifications of a dependability benchmark of general-purpose operating systems and an example of an implementation prototype used to benchmark Windows NT4, 2000 and XP.

The benchmark addresses the user perspective. The OS is considered as a black box and the only required information is the description of its API. We put emphasis on the OS robustness as regards application induced erroneous behaviours. We have defined two sets of benchmark measures (basic and complementary). We have illustrated how these measures complement each other in Annex 2-B.

The benchmark has been defined so that the workload executed on the OS can be any performance benchmark workload (and, more generally, any user tailored application) intended to run on top of the benchmarked OS. We have used a TPC-C client.

The comparison of the three OSs showed that i) they are equivalent from the robustness point of view and that ii) Windows XP has the shortest reaction and restart times. Sensitivity analyses with respect to the parameter corruption technique and to the system calls to be corrupted (performed in Annex A) showes that, even though for each OS the robustness is slightly impacted by the technique used and the system call considered, the three OSs are impacted similarly and remain equivalent.

In addition to the comparison of the three OSs, the results presented in Annex 2-A showed that using a reduced set of experiments (113) targeting only out-of-range data has led to results similar to those obtained from the 552 initial experiments targeting additionally incorrect data and addresses. If this is confirmed for other operating system families, this

would divide the benchmark execution duration (that is proportional to the number of experiments) by almost 5, which is substantial. We will further investigate this issue.

The results allowed us to confirm that the specifications can be implemented and to illustrate the kind of results that could be obtained from such a benchmark.

The specifications, the prototypes and the results could be improved and should be enhanced towards several directions. In particular, i) we have to investigate the impact of other workloads on the results for the same OSs and i) we have to validate the results with respect to other OSs from different families, e. g., using POSIX API.

Annex 2-A Faultload Validation

Let us recall that the faultload used in the benchmark and results presented in this chapter includes a mix of three parameter corruption techniques (see Section 2.4.1): i) out-of-range data, ii) incorrect data and iii) incorrect addresses. In total 552 corrupted values for the 75 parameters related to the 28 selected system calls (see Section 2.5.2). This faultload is referred to as *FL0*.

In this Annex, we first analyse the impact of these three parameter corruption techniques on the benchmark results. Then, we present a comparative analysis between the results obtained for the selective substitution technique and those obtained using a bit-flip corruption technique (see Section 2.7.1), for the same set of parameters. Finally, we make a sensitivity analysis with respect to system calls whose parameters are corrupted. These analyses are aimed at checking the validity of the parameter corruption technique retained and of the subset of functions whose parameters are corrupted.

2-A.1 Impact of Parameter Corruption Technique

It can be argued that incorrect data is not representative of application faults that should be detected by the OS. In order to analyse its impact on the benchmark results, we have considered a reduced faultload *FL1* including only out-of-range data and incorrect addresses. Thus FL1 is composed of 325 corrupted values. Figure 2-A.1 gives the robustness of the three OSs using FL1. Comparing these results with those of Figure 2.3 shows that even though the robustness of each OS has been slightly affected by the corruption technique used, the three OSs have still very similar robustness. As for Figure 2.3, the larger difference is observed for the No signalling case, it is less than 3%.

Incorrect addresses usually point to out-of range or incorrect data. Taking a pessimistic view, let us assume that they only point to incorrect data and could be discarded too as in FL1. We have thus considered a faultload, *FL2*, comprising only out-of-range data (composed of 113 corrupted values). Figure 2-A.2 shows that using FL2 also leads to similar robustness of the three OSs. Here also, the larger difference is observed for the No signalling case, it is less than 4.3%.

Besides, the results corresponding to FL2 are very close to those corresponding to faultload FL0 used in our benchmark (Figure 2.3). More investigation is required to check the validity of this result with respect to more system calls and with respect to other workloads. Indeed, if this result is confirmed for other OS families, the number of experiments can be considerably reduced by using only out-of-range data.

This result will allow corruption of the parameters of all system calls involved in the workload using only the out-of-range technique, without increasing significantly the benchmark run duration. This result will be used in Section 2-A.3.

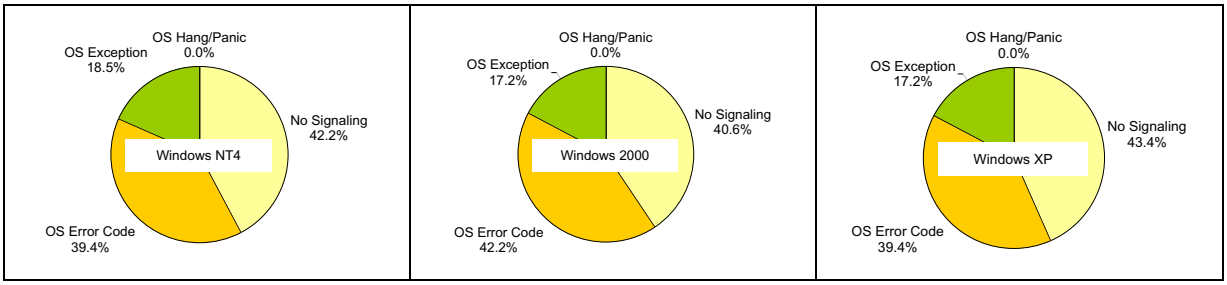

Figure 2-A.1: OS Robustness, with respect to out-of-range data and incorrect addresses (FL1)

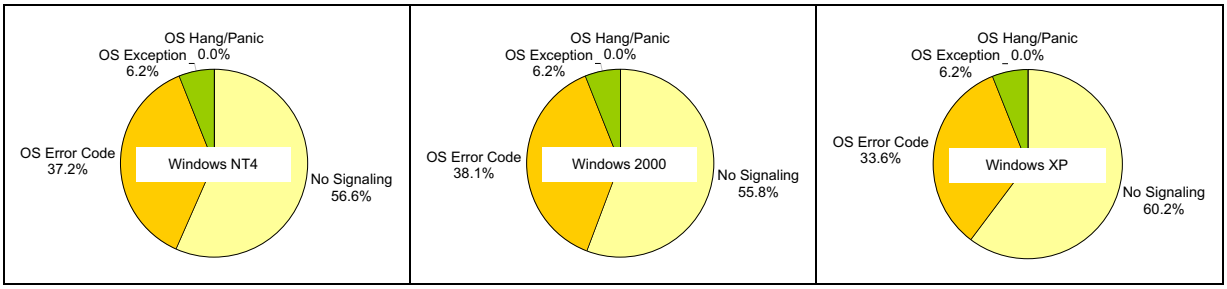

Figure 2-A.2: OS Robustness, with respect to out-of-range data (FL2)

2-A.2 Bit-Flip Technique and Selective Substitution Technique

The sensitivity of the robustness to the parameter corruption technique can be further analysed, using a bit-flip parameter corruption technique, referred to *FL3*. We use it here to corrupt the same set of 75 parameters in a systematic way (i.e., flipping the 32 bits of each parameter considered). This leads to 2400 corrupted values (i.e., 2400 experiments). The results are given in Figure 2-A.3 for Windows 2000[2]. This figure shows that the OS robustness is very similar using the two parameter corruption techniques, which confirms our previous work on fault representativeness [Jarboui et al. 2002].

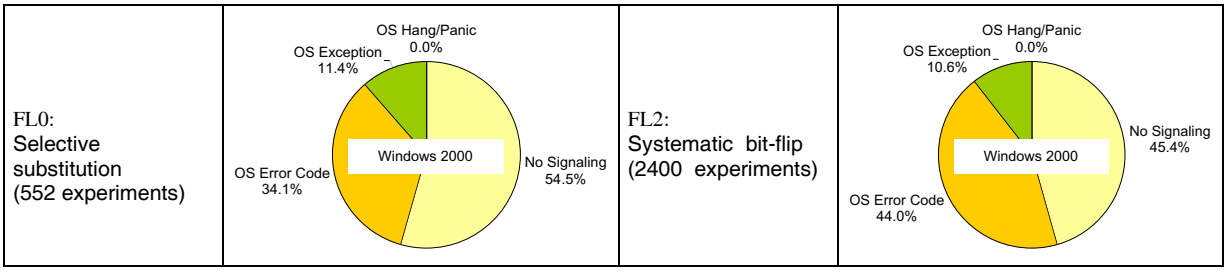

Figure 2-A.3: Windows 2000 robustness with respect FL0 and FL3

2 Due to the fact that the three OSs exhibit the same behaviour with respect to the faultload used and the time required to perform the 2400 experiments, we have not made the experiments for Windows NT and XP.

It can thus be concluded that the three OSs present *similar robustness results regardless of the parameter corruption technique* used, when corrupting the same set of parameters of the same set of system calls used in the same workload.

Our recommendation is to use a mix of the three corruption techniques as we did in Section 2.4 and 2.5, to be in the safe side. Nevertheless, for Windows family, out-of-range data give satisfactory results and the benchmark could include only out-of-range data.

2-A.3 Impact of system calls considered

In order to analyse the impact of system calls whose parameters are corrupted, we have corrupted the parameters of all the 132 systems calls with parameters, involved in TPC-C. Based on the results obtained above (Section 2-A.1), related to FL2 (out-of-range data) compared to FL0 (out-of-range data as well as incorrect data and addresses), we have considered only out-of-range data (FL2). 353 parameters have thus been corrupted and 468 experiments have been performed for each OS. Let FL4 be this faultload. The results are given in Figure 2-A.4.

They show the three OSs have *similar robustness*, when corrupting all system calls involved in TPC-C client workload. Here also, the larger difference is observed for the No signalling case, it is less than 5.3%.

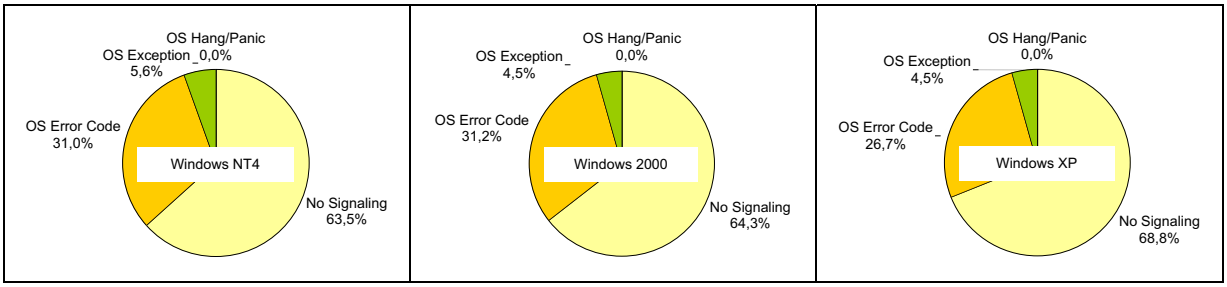

Figure 2-A.4: OS Robustness, with respect to out-of-range data (FL2) for all the 132 system calls involved in TPC-C workload (FL4)

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

Table 2.4.1 summarises the experiments performed for benchmark validation

Table 2-A.1: Summary of experiments performed for benchmark validation

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

Annex 2-B Benchmark Measure Refinement

This Annex is intended to show how the benchmark basic measures related to the OS in the strict sense can be complemented and refined based on the complementary set of measures. We will show how the results presented in Section 2.6 can be enriched, by examining additional information that can be provided by the current benchmark prototype.

We will consider successively the three benchmark measures (robustness, OS reaction time and restart time).

2-B.1 Robustness

The current benchmark prototype does not allow distinction between the workload correct and erroneous completion states. Additional instrumentation is required to do so. Also, we do not distinguish between Abort and Hang states for sake of simplicity. Hence, lines 3 and 4 of Tables 2.2 and 2.3 are grouped.

Table 2-B.1 gives the number of experiments that led the workload to the associated state (workload completion, WC and workload Abort/Hang).

Table 2-B.1: Number experiments and workload states

	Windows NT4	Windows 2000	Windows XP
WC	451	445	424
WAb/WHg	101 (18.3%)	107 (19.4%)	128 (23.2%)

Table 2-B.2 gives the combined state occurrences. It shows that:

- After an error code issue, the workload is an Abort/Hang state in 25% (46 / 182) of cases for Windows NT (resp. 28% and 42 % for Windows 2000 and XP).
- After an exception notification, the workload is an Abort/Hang state in 12% of cases for Windows NT (resp. 9% for Windows 2000 and XP).

The latter result shows that even though the TPC-C benchmark is intended to exercise the system by submitting transactions and is not intended to process exceptions, only a small number of experiments lead to Workload Abort/Hang after notification of an exception. This percentage is higher after an error code issue.

The right most column gives the workload robustness P_{SNS} (as defined in Table 2.4), that is composed of two elements in this case. When the OS is in the SNS state, the workload is in the Abort/Hang states in about 16 % of cases, for the three OSs. This result substantiates the equivalence of the three OSs.

Table 2-B.2: Number of combined state occurrences

Windows NT4→ ↓ Workload	Error code (182)	Exception (66)	Panic	Hang	No-signalling (304)
Workload completion	136	58		—	267
Abort / Hang	46	8	0	0	47

Windows 2000 → ↓ Workload	Error code (188)	Exception (63)	Panic	Hang	No-signalling (301)
Workload completion	136	57	—	—	252
Abort / Hang	52	6	0	0	49

Windows XP → ↓ Workload	Error code (172)	Exception (63)	Panic	Hang	No-signalling (317)
Workload completion	99	57	—	—	268
Abort / Hang	73	6	0	0	49

* Non-decidable states

2-B.2 OS Reaction Time

The three last lines of Table 2-B.3 complete the information provided in Table 2.7. They give the OS reaction time with respect to OS outcomes after execution of a corrupted system call. It can be seen that i) the time to issue an error code is very short and comparable for the three systems, ii) the time to notify an exception is higher than that of error code issue but it is still acceptable for Windows NT4 and XP, but very large for Windows 2000 and iii) the largest execution time is obtained when the OS does not detect/signal the error (SNS).

Table 2-B.3: Detailed OS reaction times

	Windows NT4		Windows 2000		Windows XP	
	Average	St. deviation	Average	St. deviation	Average	St. deviation
τexec	344 µs		1782 µs		111 µs	
Texec	128 µs	230 µs	1241 µs	3359 µs	114 µs	176 µs
TSEr	17 µs	18 µs	22 µs	28 µs	23 µs	17 µs
TSXp	86 µs	138 µs	973 µs	2978 µs	108 µs	162 µs
TSNS	203 µs	281 µs	2013 µs	4147 µs	165 µs	204 µs

The very high standard deviation is due to a large variation around the average. Figures 2-B.1 to 2-B.3 confirm this variation. They identify the system calls that led to each of the above outcomes and give the associated average reaction time in the presence of faults.

Figure 2-B.1 shows the different system calls that have generated an error code, with the average error code generation time of each of them. Globally, we see that these average times

are very close to the global average TSEr (indeed, the standard deviation is smaller than the average in this case).

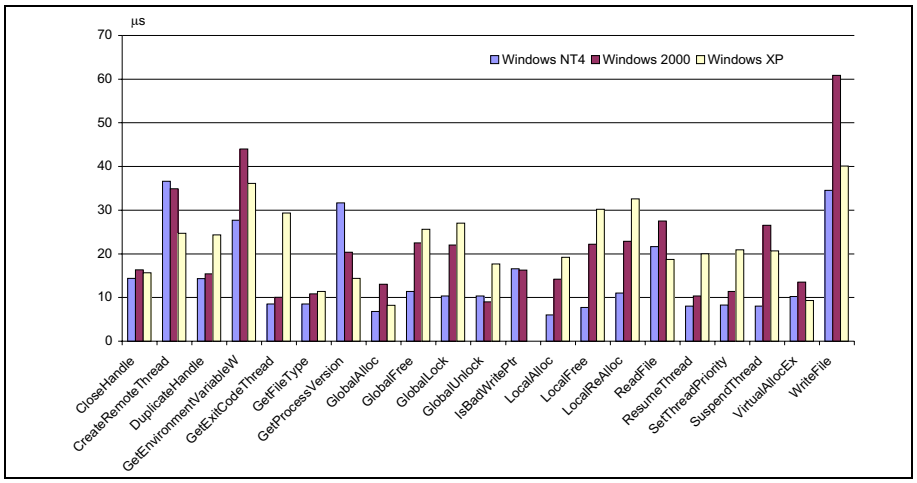

Figure 2-B.1: Detailed OS reaction time, in SEr, with respect to system calls

Figure 2-B.2 gives the system calls for which an exception was notified with the average notification time of each of them. The large standard deviation seems to be due to `GetPrivateProfileStringA`.

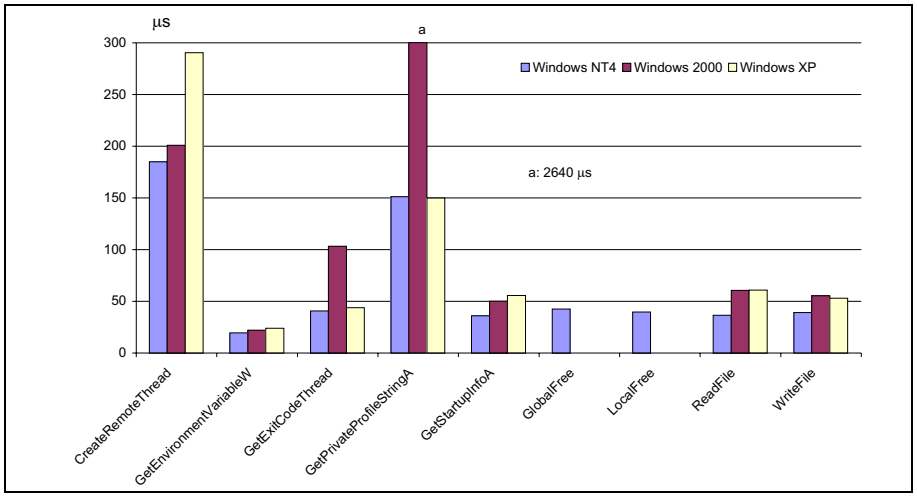

Figure 2-B.2: Detailed OS reaction time, in SXp, with respect to system calls

Figure 2-B.3 gives the system calls for which the corrupted parameter was not detected, leading to SNS. Here two system calls have largely contributed to the high standard deviation, `GetPrivateProfileStringA` and `GetPrivateProfileIntA`.

These figures suggest that more analyses are required to explain the behaviour with respect to `GetPrivateProfileStringA` and `GetPrivateProfileIntA` system calls. However, if we re-evaluate the various average times without these two system calls, we can see in Table 2-B.4 that all system reaction times have been reduced compared to those of Table 2-B.3. Windows XP still has the shortest reaction times.

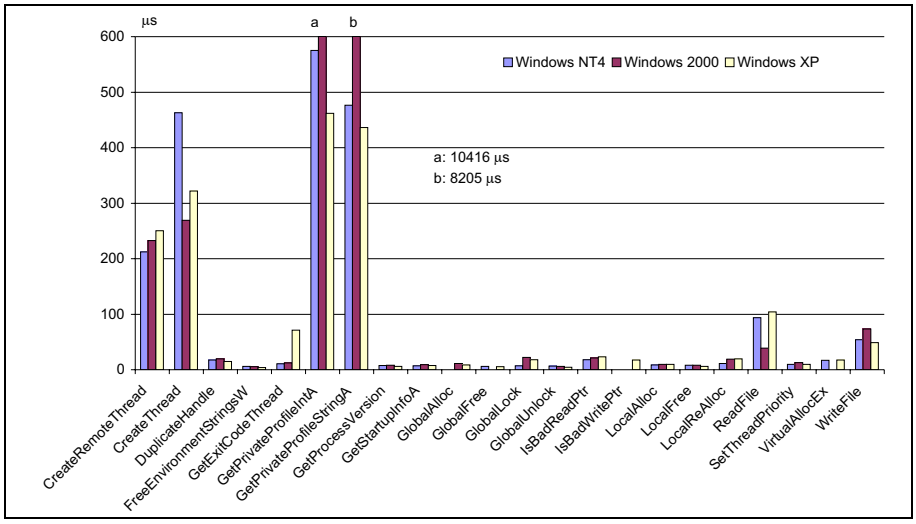

Figure 2-B.3: Detailed OS reaction time, in SNS, with respect to system calls

Table 2-B.4: Reaction times without `GetPrivateProfileIntA` and `GetPrivateProfileStringA` system calls

	Windows NT4		Windows 2000		Windows XP	
	Average	St. deviation	Average	St. deviation	Average	St. deviation
τexec	323 µs		1054 µs		60 µs	
Texec	72 µs	180 µs	61 µs	96 µs	63 µs	120 µs
TSEr	16 µs	18 µs	21 µs	28 µs	23 µs	16 µs
TSXp	56 µs	60 µs	79 µs	76 µs	85 µs	127 µs
TSNS	117 µs	241 µs	87 µs	120 µs	96 µs	146 µs

2-B.3 System Restart Time

Even though the system restart time in the presence of faults in not very different from the system restart time without fault, the plots of this time in presence of faults with respect to all experiments (given in Figure 2-B.4) show the existence of two distinct values.

Careful analysis of the collected data suggests a correlation between the system restart time and the state of the workload. When the workload is completed, the average restart time is very close to the one obtained without fault injection, and when the workload is aborted or hangs, the restart time is 8% to 18% higher. Indeed, the number of experiments that led to workload abort/hang was respectively 101, 107 and 128 for Windows NT4, 2000 and XP. Even though Windows XP had induced more workload abort/hang, it still has the lowest system restart time as indicated in Table 2-B.5. The latter recalls in lines 1 and 2 the restart times without faults, τres, and in presence of faults, Tres, and refines Tres in the last two lines according to the workload state, i.e., completion or abort/hang, irrespective of the OS outcome.

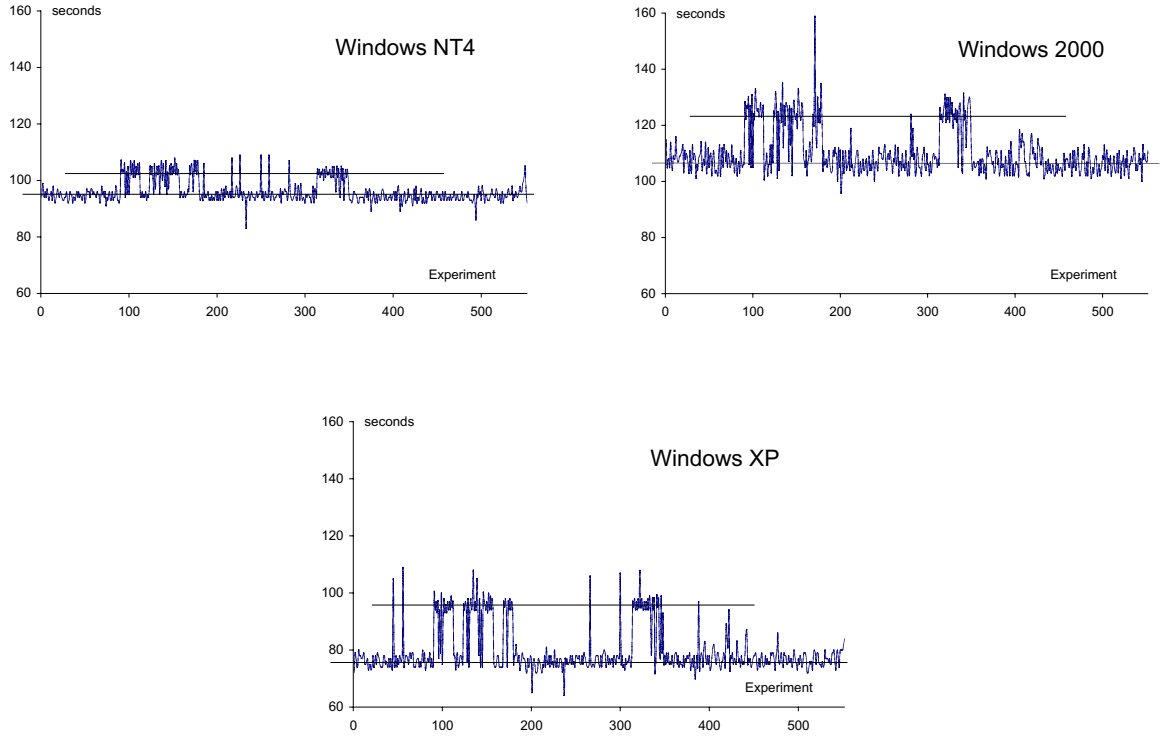

Figure 2-B.4: Detailed OS restart times

Table 2-B.5: System restart time according to the workload state

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

2-B.4 Workload execution time

Table 2-B.6 summarizes the time of workload completion without fault (τWC) and in the presence of faults (TWC). It can be noted that, compared to the workload execution time in absence of faults, the increase of workload execution time is 3% for Windows XP, 6 % for Windows 2000 and 8% for Windows NT. In addition the standard deviations are relatively small denoting small variations in workload execution time.

Table 2-B.6: Workload execution times

	τWC	TWC	Standard deviation
Windows NT4	74 s	80 s	12 s
Windows 2000	70 s	74 s	13 s
Windows XP	67 s	69 s	10 s