14

BENCHMARKING THE IMPACT OF FAULTY DRIVERS: APPLICATION TO THE LINUX KERNEL

Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre

14.1. INTRODUCTION

Dependability concerns, encompassing robustness assessment, are essential questions to answer before a developer can make the decision whether to integrate off-the-shelf (OTS) components into a dependable system. Here, and in what follows, robustness is understood as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions, in compliance with the generic definition (dependability with respect to external faults) given in [Avižienis et al. 2004].

From a cost-effectiveness viewpoint, operating systems and kernels are privileged OTS components as candidates for integration into a system. However, integrators are often reluctant to make such a move without obtaining a deeper knowledge and understanding about such a component beyond functional issues, in particular with respect to its failure modes and its behavior in the presence of faults. Due to the opacity that is often attached to the commercial offer and to the difficulty and significant cost associated with the availability of the source code, the Open Source option, for which access to the source code is granted, is progressively making its way as an attractive and promising alternative. Also, results of many studies have shown that Open Source solutions did not exhibit significantly more critical failure modes and in some cases they were even found to demonstrate behaviors superior to commercial options [Koopman and DeVale 1999, Arlat et al. 2002, Marsden et al. 2002, Vieira & Madeira 2003]; see also Chapter 5 for the latter. In this chapter, we will simply denote such components (either commercial or Open Source) as OTS components.

In the past years, several experimental studies have addressed this important issue from different perspectives [Koopman and DeVale 1999 (see also Chapter 11), Arlat et al. 2002, Madeira et al. 2002]. This has also led to the proposal of tentative dependability benchmarking approaches, aimed at characterizing the robustness of computer systems and OTS [Tsai et al. 1996, Mukherjee and Siewiorek 1997, Brown and Patterson 2000]. However, such proposals were still preliminary and did not reach the level of recognition attached to performance benchmarks. As pointed out earlier (e.g., in the Preface and several chapters), the DBench project, in which this work was included, was another major contribution aimed at promoting such an approach by defining a comprehensive framework for the definition and implementation of dependability benchmarks, see also [Kanoun et al. 2002, Kanoun et al. 2005a].

A large part of the code that makes up an operating system consists of device driver programs and the OS configuration may change not only at installation time, but also in operation. For example, in the case of Linux, drivers have consistently represented more than half of the source code [Godfrey and Tu 2000]. This ratio is smoothly increasing; recent releases have accounted for almost 60% of the code [Gu et al. 2003]. More importantly, as the whole size of the kernel is rapidly growing, this results in an exponential increase in the number of lines of code of the driver programs. Such programs are commonly developed by third-party hardware device experts and integrated by kernel developers. This process is not always well mastered and erroneous behavior of such programs that are intimately connected to the kernel may have dramatic effects. As pointed out in [Murphy and Levidow 2000] for Windows and as shown by the analysis of the Linux source code carried out in [Chou et al. 2001], a significant proportion of operating system failures can be traced to faulty drivers. Things are not improving much; indeed, as quoted in [Swift et al. 2004], in Windows XP, driver programs are reported to account for 85% of recently reported crash failures.

It is thus necessary to investigate and propose new methods, beyond the collection of field data, for specifically analyzing the impact of faulty drivers on operating systems. Fault injection techniques, whereby faulty behaviors are deliberately provoked to simulate the activation of faults, provide a pragmatic and well-suited approach to support such an analysis. Among the fault injection techniques, the software-implemented fault injection (SWIFI) technique (e.g., see [Carreira et al. 1998]) provides the proper level of flexibility and low intrusiveness to address this task. Based on these principles, we have developed an experimental environment for the evaluation of the robustness of the Linux kernel when faced to abnormal behaviors of its driver programs.

To our knowledge, very few research studies have been reported on this topic. The work reported in [Edwards and Matassa 2002] also concerns the Linux kernel but focuses, rather, on the dual problem of characterizing the robustness of driver programs when subjected to hardware failures. The authors have devised a sophisticated approach to inject faults in the driver under test that relies on the appealing notion of the common driver interface (CDI) that specifies the driver interactions within the kernel space. In [Gu et al. 2003], the authors have conducted a comprehensive dependability analysis of the Linux kernel. However, in this study fault injection has been related to the execution stream of the kernel code; more precisely, a selected set of functions of the kernel has been targeted, namely, the processor dependent code, the file system support code, the core kernel code, and the memory management code.

Our goal is instead to benchmark the robustness of an operating system kernel in the presence of faulty drivers. In line with this objective, but considering several instances of the Windows series, in [Durães and Madeira 2003] the authors have used mutations of the executable code of the driver to simulate a faulty driver. The work reported in this chapter is rather complementary, in the sense that we investigate an alternative approach: fault injection is targeting the parameters of the kernel core functions at the specific interface between the driver programs and the kernel. To support this approach, we revisit and adapt the notion of the common driver interface of [Edwards and Matassa 2002] by focusing the injection on the service/system calls made by the drivers to the kernel via the core functions. This way, the definition of the fault injection experiments more thoroughly impact the interactions between the drivers and the kernel. Also, the errors provoked can simulate both the consequences of design faults or hardware-induced faults. It is worth noting that a subsequent and complementary work has been reported recently that adopts the same system model (explicit separation between the kernel and the drivers) and a similar fault model to study the impact of faulty drivers on a Windows CE-based system [Johansson and Suri 2005]. The main goal was to study the error propagation process to support the placement of protection wrappers (e.g., see [Fraser et al. 2003]), as was already carried out in [Arlat et al. 2002].

The material reported herein elaborates on the work published in [Albinet et al. 2004]. The organization of the chapter is as follows. Section 14.2 describes the specific issues addressed by the approach we propose, in particular in the light of other works dealing with the characterization of operating system robustness. In Section 14.3, we briefly describe the various types of driver programs and introduce the specific interface considered to simulate faulty drivers for the Linux kernel, the *driver programming interface.* Section 14.4 presents the benchmarking attributes, namely, the faultload, the workload, the measurements, and the measures that allow for performing analyses accounting for several end-user viewpoints. Section 14.5 describes the main features of the RoCADE (Robustness Characterization Against Driver Errors) platform that supports the conduct of the experiments. Section 14.6 presents a sample of results and illustrates how the proposed benchmarking framework can support the analysis of the results. Finally, Section 14.7 concludes the chapter.

14.2. CONTEXT AND DEFINITION OF THE APPROACH

Due to their central role in the functioning of a computer system, operating systems are the primary targets for developing dependability benchmarks. Figure 14.1 depicts the

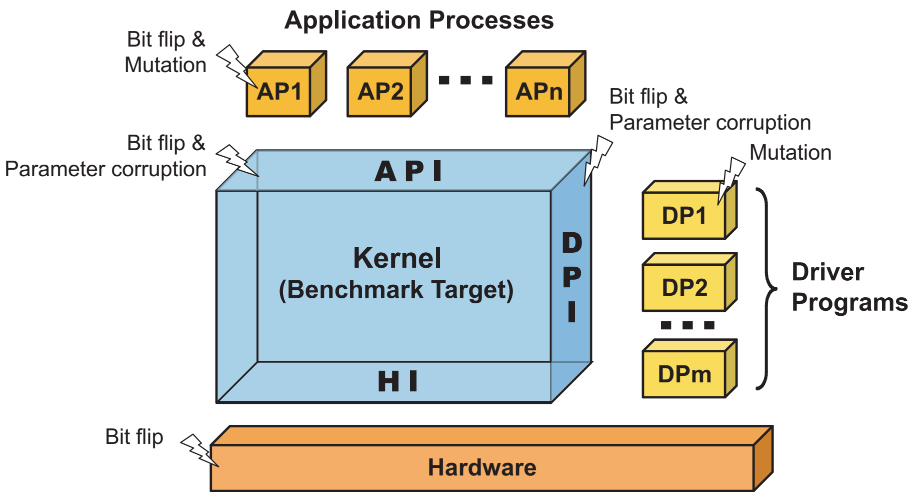

Figure 14.1. Interactions between an operating system kernel and its environment.

software architecture of a computer system. In this chapter, due to the emphasis put on the analysis of the impact of the driver programs, the benchmark target (BT), according to the terminology put forward by the DBench project (e.g., see [Kalakech et al. 2004], as well as Chapters 5 and 6), is the operating system kernel. Drawing further on that terminology, the whole figure describes the system under benchmark (SUB), that is, the supporting environment and context within which the analyses are conducted. As shown on the figure, the kernel features three main interfaces with its environment. The first one is basically concerned with hardware interactions, whereas the other two are software related.

The "lightning" symbols in Figure 14.1 identify possible locations where faults can be applied. The interfaces and related faults are briefly described as follows:

1. The hardware interface (HI) is primarily related to the hardware layer. The main interactions are made via the raising of exceptions. At this level, several studies have been reported in which faults were injected by means of bit flips into the memory of the SUB; for example, see [Arlat et al. 2002, Gu et al. 2003]; see also Chapter 15, for what concerns the latter reference.
2. The interface with the applications processes corresponds to the application programming interface (API). The main interactions are made by means of system calls. A significant number of studies were reported that target the API to assess the robustness of the operating systems (e.g., in the form of code mutations [Durães & Madeira 2006]; see also Chapter 6), by means of bit flips [Jarboui et al. 2003] or by corrupting the system calls [Koopman and DeVale 1999, Kanoun et al. 2005b]; see also Chapters 11 and 12 for what concerns these respective references.
3. The interactions with drivers are made through the driver programming interface (DPI), which is the programming interface for the development of the drivers. The detailed definition of this interface is presented in Section 14.3.

Concerning the third item, the form of fault injection we are advocating is carried out by way of corruption of the calls to the functions of the kernel made by the drivers. Such an approach has been motivated by the work carried out in [Jarboui et al. 2003]. In this work, assertions issued from traces characterizing the actual erroneous behavior induced by faulty drivers were used to assess whether similar error patterns could be obtained by using several fault injection techniques (either bit flip or parameter corruption) at the API level. This study showed that API-level fault injection was not able to produce errors that matched the error patterns provoked by real faults in drivers. This result further substantiates the need to conduct investigations specifically aimed at closely simulating the impact of faulty drivers. To this end, we have concentrated our efforts on intercepting and corrupting the parameters of the system calls issued by the drivers at the DPI [Edwards and Matassa 2002]. Compared with the mutation of the code of the drivers used in [Durães and Madeira 2003], this approach allows for carrying out a more focused and efficient set of experiments that are suitable to thoroughly test the various kinds of interactions between the drivers and the kernel. The price to pay for this is a precise identification of the DPI upon which the faults are specified. However, it is worth noting that (as for the approaches targeting the API), such a preliminary analysis has to be carried out only once for each kernel family and can be reused for analyzing most drivers. More details on the types of faults considered are given in Section 14.4.1.

The interfaces depicted in Figure 14.1 (especially the API and the DPI) also provide suitable locations to observe the consequences of the injected faults. At the API level, the typical relevant behaviors include error codes returned to the calling application processes and kernel hangs. In practice, although a lot of exceptions are raised by the SUB at the hardware layer, for the sake of efficiency they are often caught at the API when reported via the kernel. The DPI provides a favorable level of observation for the detailed characterization of the reactions of the kernel to the corrupted service calls issued by the drivers. The related measurements include error codes returned to the driver. Comprehensive measurements can also be made at the application level, for example, workload abort and completion. Although time measurements can be collected to evaluate the workload execution in the presence of faults [Kalakech et al. 2004], the reported results only consider nontimed robustness measures expressed as frequencies of occurrence of each considered outcome.

As a general comment, it is important to stress that the founding of the benchmarking attributes proposed in this chapter (especially the faultload and measurements) on explicit interfaces (namely the DPI and API) that can be easily shared by several candidate BTs is fundamental for guaranteeing a fair assessment. Indeed, this is an essential and favorable feature for sustaining the development and dissemination of relevant dependability benchmarks, elaborating on the approach proposed in this chapter.

14.3. THE DRIVER PROGRAMMING INTERFACE

In this section, we briefly recall the functional interactions that characterize the communication between the drivers and the kernel. This allows for the kernel functions and parameters involved in these interactions to be identified and, thus, the DPI on which we define the types of faults to be injected.

14.3.1. Application Processes, Kernel, and Drivers

Classically, the application processes execute in a restricted address space (user space). This is meant to reduce the risk for an application process to corrupt the kernel addressing space. This way, it is likely that the errors caused by a faulty application process mainly impact its own address space and, thus, are limited to its execution. Conversely, mainly for performance reasons, the kernel and the drivers are executed in privileged mode (kernel mode) and thus share the same address space. This means that the risk for a defective driver to impact the behavior of the kernel cannot be neglected. Due to the fact that it is not always possible to associate a "pedigree" to the whole set of drivers that can potentially be integrated, drivers are a potential threat for the kernel. This is further exacerbated by the programming languages (such as C language) that use pointer arithmetic without IMM (integrated memory management). This applies to several popular general-purpose operating systems (e.g., Linux and Windows9x). The drivers can also access the whole set of functions of the kernels, not only those that are used to carry out operations on the kernel space, but also on the application space.

14.3.2. The Various Drivers

An interesting comparative study of driver interfaces for several popular operating systems is presented in [Zaatar and Ouaiss 2002] as an initial step toward the standardization of the Linux driver interface. Irrespective of the different solutions adopted for a specific operating system family, in practice two main categories of drivers can be distinguished:

1. Software drivers—they have no direct access to the hardware layer of the devices, but rather to an abstraction (e.g., tcp/ip stack, file system).
2. Hardware drivers—they are concerned with hardware devices, either peripheral (network card, disk, printer, keyboard, mouse, screen, etc.) or not (bus, RAM, etc.).

In both cases, the role of a driver is to provide an abstract interface for the operating system to interact with the hardware and the environment:

- More specifically, a driver is meant to implement a set of basic functions (read, write, etc.) that will activate peripheral devices.
- On top of drivers, the input–output instructions no longer depend on the hardware architecture.
- Drivers define when and how the peripheral devices interact with the kernel.

For example, in the case of a driver relying on polling, an application process issues a request, via a system call (`open, read or ioctl`) to access a peripheral device (network card, disk, printer, keyboard, mouse, screen, etc.). The processor enters the supervisor mode, via a stub in the case of Linux, and executes the code of the driver corresponding to the proper operation. After completion of the operation, the driver frees the processor and the processor then resumes the execution of the application process in user mode.

Although device drivers may induce a strong influence on the kernel (as most of them are run in kernel mode), they are often developed by third parties and then integrated into the kernel after its distribution. This explains why it has been found that they significantly contributed to the failure of the operating system [Chou et al. 2001].

14.3.3. Specification of the DPI

The drivers make use of specific system calls (denoted symbols for dynamic module drivers in the case of Linux) in order to perform tasks. The most salient categories are depicted in Table 14.1.

Each of these categories gathers a set of functions that are devoted to the programming of the kernel and drivers using execution privileges within the kernel address space. For example, in the case of Linux, functions allow for acquiring and releasing of an interrupt channel (`request_irq, free_irq`) and for retrieving the status of such a channel (`irq_stat`). These symbols form the basis for the development of drivers for managing the interrupt channels. All such symbols feature a calling protocol that is similar to the system calls of the Linux API.

This is illustrated by the *signature* of the `request_irq`:

```
int request_irq(unsigned int irq,
     void (*handler)(),
     unsigned long irqflags,
     const char * devname,
     void *dev_id)
```

TABLE 14.1. Outline of the categories of symbols for Linux

Categories	Examples of typical symbols
Memory Management	Kmalloc, kfree, free_pages, exit_mm, . . .
Interrupt Management	add_timer, del_timer, request_irq, free_irq, irq_stat, add_wait_queue, _wait_queue, finish_wait, . . .
File System Management	fput, fget, iput, follow_up, follow_down, filemap_fdatawrite, filemap_fdatawait, lock_page, . . .
Control Block Management	blkdev_open, blkdev_get, blkdev_put, ioctl_by_bdev, . . .
Registration	register_sysctl_table, unregister_sysctl_table, sysctl_string, sysctl_intvec, . . .
Others: Software interrupts, dma management, buffering management, resource handling, process management, interfaces, debug, miscellaneous "tools"	raise_softirq, open_softirq, cpu_raise_softirq, dump_stack,ptrace_notify, current_kernel_time, sprintf, snprintf, sscanf, vsprintf, kdevname

The `request_irq` function allocates a peripheral device to an interrupt channel. The function returns a success (error) code (an integer value) to inform the calling driver program of the proper (or not) handling of the reservation of the channel. The first argument, `irq`, is an unsigned integer that designates the channel to allocate. The second one, `handler`, is a pointer to the interrupt manager. The third one is an unsigned long integer that represents the flags that define the type of the reservation (exclusive or not, etc.). The parameter `devname` is the name of the peripheral device that is reserving the channel. The last parameter is a pointer to a "cookie" for the interrupt manager.

From more than a thousand symbols (including functions, constants, and variables), Linux release 2.4.18 includes about 700 kernel functions. Some are more used than others. The kernel functions devoted to memory reservation are definitely much more solicited than the ones attached to the handling of a pcmcia device. The types of the parameters being used in kernel programming are voluntarily restricted to integers (`short` or `long`, `signed` or `unsigned`) and `pointers`. We have referenced all these functions along with their signature, which allows for the number of parameters and their types to be specified for each symbol. These types are defined over a validity space (see extreme values in Section 14.4.1)

In the same way that the API gathers all the available system calls issued by the application processes, the DPI gathers all the functions of the kernel that are available to be used by the drivers. These kernel symbols constitute the features offered to the developers in kernel mode.

14.4. THE BENCHMARKING ATTRIBUTES

This section briefly describes the *execution profile* and the *measures* that are considered for the benchmarking analysis [Arlat et al. 1990]. The execution profile includes both the *workload* (the processes that are executed to activate the drivers and the kernel) and the *faultload* (the set of faults that are applied during the fault injection experiments via the DPI). The experimental *measures,* which are meant to characterize the reaction and/or behavior of the kernel in presence of a faulty driver, are elaborated from a set of observations (*readouts* and *measurements*) that are collected during each experiment.

In order to illustrate how measurements can be used to derive useful measures, we will consider several dependability viewpoints according to the perceptions that different users can have of the observed behaviors. In the sequel to this section, we consider in turn the faultload, the workload, the measurements, and the measures that build up on these measurements with the objective of dependability benchmarking.

14.4.1. The Faultload

For corrupting the parameters of the symbols of the DPI, we have used the SWIFI technique because of its flexibility and ease of implementation. More precisely, in order to generate more efficient test conditions, we have focused the corruption of function parameters on a set of specific values. In particular, this provides a better control of the types of corruptions that are made, which significantly facilitates the interpretation of the results obtained. Faults are injected into each parameter of each relevant function of the DPI, as shown in Figure 14.2.

The principle of the method is to intercept a function when it is called, to substitute the value of its parameters with a corrupted value, and then to resume the execution of the function with this faulted value. The value that is substituted to the original value of the faulted parameter depends upon the type of the parameter. Table 14.2 shows the values considered for each relevant type. For the first three types, bounding and mid values are considered. For pointers, the set of corrupted values are: NULL, a max bounding value, and a random value.

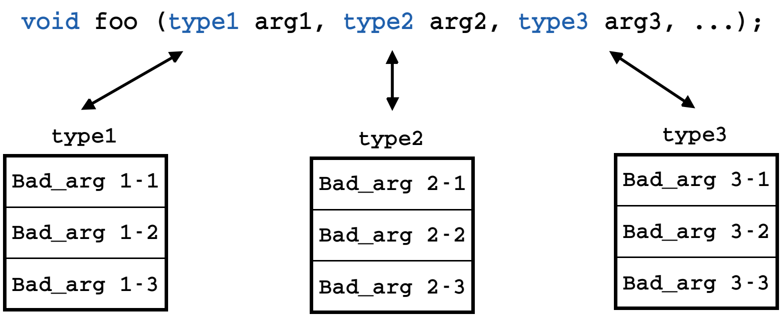

Figure 14.2. Principle of corruption of the parameters of a function.

TABLE 14.2. The faulty parameters for each type

Type	Bad_Arg 1	Bad_Arg 2	Bad_Arg 3
int	INT_MIN	0	INT_MAX (0x7FFFFFFF)
uint	0	INT_MIN (0x80000000)	ULONG_MAX (0xFFFFFFFF)
ushort	0	SHRT_MIN (0x8000)	USHRT_MAX (0xFFFF)
pointer	NULL	random( )	All bits = 1 (0xFFFFFFFF)

14.4.2. The Workload

In order to provoke the activation of the DPI by the driver programs, so as to mimic the nominal behavior, we rely on an indirect activation procedure by means of a workload applied at the level of the API. We consider a synthetic and modular workload combining several activation processes, each targeting one (or several) of the drivers evaluated. Each application process carries out a set of elementary operations concerning a specific hardware or software driver component:

1. Deinstallation of the target component that permits (only if the driver is currently used by the system) starting the test later on by registering the component.
2. (Re-)installation of the component allowing for testing component registration.
3. Series of requests meant to test the driver's operation.
4. Deinstallation, by which the unregistration of the component is tested.
5. Reinstallation, whenever needed, in particular if the driver is mandatory for the SUB's operation (e.g., network card or file system).

For example, in the case of a network card, the application process disables the network, unloads the network driver, reloads it, enables the network, runs a test on a private Ethernet network (intranet), disables the network, unloads the driver, reloads it, and, finally, enables the network.

The main differences between the application processes that form the workload concern the specific requests to be applied to stimulate the driver.

In order to better assess the impact of the fault on the whole SUB, a subsequent workload execution is carried out after the fault has been withdrawn; this is particularly useful to improve the diagnosis in the cases when no outcome is observed as the result of the run when a fault is injected (the so-called "silent" behavior as reported in the CRASH scale proposed in [Koopman and DeVale 1999, see also Chapter 11]. In the reported experiments the workload that is executed for improving the diagnosis is the same as the workload used for the fault injection experiments. Accordingly, hereafter we will refer to it as the "replay" workload.

14.4.3. The Measurements

The goal of the set of experiments reported here is to determine the set of relevant observations to be incorporated into a prototype dependability benchmark, focusing on robustness with respect to faulty drivers. Accordingly, to get relevant insights from the conducted experiments, it is necessary to obtain a good variety of results. To that end, we have specified two levels of observation: (i) *external or user-oriented,* meant to characterize the faulty behavior, as perceived at the level of the API, (ii) *internal or peripheral device oriented,* which detail the impact of the faults on the kernel, as perceived at the level of the DPI.

The *external* level includes the observation of the errors reported by the kernel to the application processes in the workload (exceptions, error codes, etc.). These observations can be augmented by a more *user-oriented* perception by means of observations directly related to the application processes (e.g., the *execution time* of the workload or the *restart time*). The *internal* level focuses on the exchanges between the kernel and the faulted driver. The specific observations made at each level as well as the related appraisals are depicted in Table 14.3.

TABLE 14.3. Observation levels, events, and appraisals

Level	Event	Appraisal good/bad	Abbreviated name
Internal	DPI Error Code: Code returned by the kernel at the level of the DPI	good	EC
External	Exception: Processor's exceptions observed at the API level	good	XC
	Kernel Hang: The kernel no longer replies to a request issued via the API	bad	KH
	Workload Abort: The workload has been abruptly interrupted (some API service requests could not be made).	bad	WA
	Workload Incorrect: The workload completes, but not all the return codes are "success."	bad	WI
	Workload Completion: This event allows for the execution time of the workload programs to be measured.	good	WC

The *error code* returned by a function of the kernel provides an essential insight into the impact of the fault on the intimate behavior of the kernel. Indeed, from a robustness viewpoint, the kernel symbol should be able to react to a service call including an argument with a corrupted value by returning an error code that matches the type of fault being injected. When a hardware *exception* is raised while a process executes in the kernel address space, the kernel tries to abort the process or enters the "panic" mode. The consideration of *workload*-related events (WA or WI) allows for additional insights to be obtained, especially in cases when no error is reported by the kernel. In that respect, the "replay" workload that is executed after each run during which a parameter is corrupted allows for the damage caused by the application of faulty call to be assessed by identifying whether the SUB was able to recover a stable state on its own or if a specific restart is necessary.

A *hang* of the kernel is diagnosed when the kernel is no longer replying to requests. The main reasons for such blocking are either because it executes an infinite loop or it is waiting for an event while interrupts are masked. Such outcomes cannot be observed by the system and, thus, require external monitoring.

Measuring the execution time of the workload programs provides useful information on the capacity of the kernel to handle the applications processes in presence of faults and is thus a desirable feature from the benchmarking point of view. Due to the specific nature of the workload (synthetic workload), such a measurement was not carried out in the study described here. The interested reader can refer to the work reported in [Kanoun et al. 2005b]; see also Chapter 12. The technique used therein can be applied to obtain the corresponding measurements.

14.4.4. Interpretation of Measurements to Yield Benchmarking Measures

The observations described in the previous section offer a basis upon which various types of analyses can be carried out, depending on how one interprets the impact of the combined behaviors observed for various dependability concerns. In practice, different interpretations of the measurements are possible depending on the specific context in which the kernel is to be integrated. In particular, when one is favoring safe behavior of the workload, then error notification via error code return or even kernel hangs might be proper or acceptable behaviors. Conversely, returned error codes or selective application process aborts are much more suited for cases in which availability of the kernel is the desired property. This is further exacerbated in cases in which several outcomes (e.g., error code returns and hangs) are observed simultaneously within the same experiment run. So as to reliably account for various points of view, one has to carefully analyze such cases. It is worth pointing out that the types of analyses that we are proposing here are in line with and elaborate on the related study reported in [Rodríguez et al. 2002] and on the assessment framework used in [Durães and Madeira 2003].

14.4.4.1. Outcomes and Diagnoses. Table 14.4 identifies the outcomes and several criteria that can be considered for exploiting the outcomes. The first set of columns shows the possible outcomes (i.e., combinations of the events defined in Table 14.3). Two categories are distinguished: error notification (explicit error reporting) and failure modes. The second part of the table illustrates how the outcomes of several event collections per experiment can be diagnosed according to a set of simple criteria (order of occurrence of observed events and priority given either to error notification or failure modes).

First, it is worth noting that all events considered are not fully independent; accordingly, not all combinations are valid. In particular, this is the case for workload abort (WA) and workload incorrect (WI); indeed, WA dominates WI, that is, no WI can be observed when a WA has been diagnosed. This is identified by "X" in Table 14.4. This explains

TABLE 14.4. Possible outcomes and diagnoses

	Outcomes					Priority to		
	Notification			Failure modes				
#	EC	XC	WA	WI	KH	First event	Error notification	Failure modes
O1	0	0	0	0	0	No Obs.	No Obs.	No Obs.
O2	1	0	0	0	0	EC	EC	EC
O3	1	1	0	0	0	EC	EC+XC	EC+XC
O4	0	1	0	0	0	XC	XC	XC
O5	1	1	0	0	1	EC	EC+XC	KH
O6	1	0	0	0	1	EC	EC	KH
O7	0	1	0	0	1	XC	XC	KH
O8	0	0	0	0	1	KH	KH	KH
O9	1	1	1	X	1	EC	EC+XC	KH+WA
O10	1	0	1	X	1	EC	EC	KH+WA
O11	0	1	1	X	1	XC	XC	KH+WA
O12	0	0	1	X	1	KH	KH+WA	KH+WA
O13	1	1	1	X	0	EC	EC+XC	WA
O14	1	0	1	X	0	EC	EC	WA
O15	0	1	1	X	0	XC	XC	WA
O16	0	0	1	X	0	WA	WA	WA
O17	1	1	0	1	0	EC	EC+XC	WI
O18	1	0	0	1	0	EC	EC	WI
O19	0	1	0	1	0	XC	XC	WI
O20	0	0	0	1	0	WI	WI	WI
O21	1	1	0	1	1	EC	EC+XC	WI+KH
O22	1	0	0	1	1	EC	EC	WI+KH
O23	0	1	0	1	1	XC	XC	WI+KH
O24	0	0	0	1	1	WI	WI+KH	WI+KH

Legend—EC: error code, XC: exception, KH: kernel hang, WA: workload abort, WI: workload incorrect.

why the table has only 24 rows. Among these, row O1 designates cases in which none of the events has been observed. This is a classical issue in testing and experimental studies when no impact is observed. This might be due to several alternatives (fault was not activated, error masked, etc.); we will come back on this in Section 14.6.1.

When several events are observed within the same experiment, various decisions can be made in order to categorize the outcomes. One usual approach is to give priority to the first event that has been observed. However, it is not always possible to have precise timing measurements for all events. Indeed, in some cases this may require sophisticated and heavy instrumentation (e.g., see [Rodríguez et al. 2003]), which might be out of the scope of the proposal for a dependability benchmark that should be portable, minimally intrusive, and cost-effective.

Other alternatives include giving priority (i) to error notifications [e.g., error codes returned (EC) and exceptions (XC)] or (ii) to the failure modes observed [workload abort (WA), workload incorrect (WI) and Kernel Hang (KH)]. Considering the last two strategies, the first one is clearly optimistic (it assumes that notification will be able to preempt and confine any subsequent impact), whereas the second one is rather pessimistic (the system is assumed to always fail, irrespective of the possible handling of the error ensuing from the notification). In both cases, when multiple events are observed pertaining to the prioritized category, they are recorded for further analysis. The order of occurrence is also highlighted in the table. For example, when priority is given to failure modes, "WI+KH" in row O21 means that WI precedes KH. It is worth noting that, due to the way the considered events are collected, "priority to error notification" closely matches "priority to first event," because error notifications always precede all considered failure modes.

Adopting a classification relying only on end-user perception (i.e., considering only observations made via the API) would have resulted in discarding EC events. For example, in that case O2 would have been merged into O1 and it would not be possible to discriminate O3 from O4, and so on.

14.4.4.2. Viewpoints and Interpretation. More elaborate interpretations can be defined that feature more dependability-oriented measures. We will consider three of such interpretations that correspond to three distinct contexts: (1) responsiveness of the Kernel (RK), that is, maximize error notification; (2) availability of the kernel (AK), that is, minimize kernel hangs; and (3) safety of the workload (SW), that is, minimize delivery of incorrect service by the application processes. These constitute top-level perceptive viewpoints that support the selection criteria for a system integrator to rank SUBs of interest. This perception follows from the clustering of the detailed outcomes from Table 14.4 (see Table 14.5).

The main rationale for the interpretation associated with RK is to positively consider outcomes gathering both notification events and failure modes. The fact that the kernel is able to report an error is considered as positive, even when failure modes are observed at workload level. Conversely, AK will rank differently the cases in which either a KH or a WA is observed: indeed, the occurrence of a KH has a dramatic impact on the availability of the system, whereas an abort of the workload can be recovered more easily. The measure associated with SW characterizes the case in which safe behavior of the workload is required. Accordingly, we advocate that most favorable outcomes correspond to events prone to induce "fail-safe" or "fail-silent" behaviors, that is, error notifications and kernel hangs, whereas workload abort is assumed to correspond to a critical event, and incorrect completion to an even worse one. Nevertheless, as safety is typically an application-level property, alternative viewpoints could have been devised; in particular, from a "fail-fast"

TABLE 14.5. Viewpoints and dependability measures

Viewpoint: Responsiveness/Feedback of the Kernel			
	#	Outcomes [–O1]	Rationale
+	RK1	O2–O4	An error is notified by the kernel before the workload completes correctly
+	RK2	O5–O7, O9–O11, O13–O15, O17–O19, O21–O23	An error is notified by the kernel before a failure is observed
–	RK3	O16	No error is notified and the workload is aborted
–	RK4	O8, O12, O24	No error is notified and the kernel hangs
–	RK5	O20	No error is notified and the workload completes incorrectly
Viewpoint: Availability of the Kernel			
	#	Outcomes [-O1]	Rationale
+	AK1	O2–O4	An error is notified by the kernel before the workload completes correctly
+	AK2	O13–O20	The workload is aborted or completes incorrectly
–	AK3	O5–O8	The workload completes correctly and the kernel hangs
–	AK4	O9–O12, O21–O24	The workload is aborted or completes incorrectly and the kernel hangs
Viewpoint: Safety of the Workload			
	#	Outcomes [-O1]	Rationale
+	SW1	O2–O4	An error is notified by the kernel before the workload completes correctly
+	SW2	O5–O8	The workload completes correctly and the kernel hangs
+	SW3	O9–O16	The workload is aborted or the kernel hangs
–	SW4	O21–O24	The workload completes incorrectly and the kernel hangs
–	SW5	O17–O20	The workload completes incorrectly and the kernel does not hang

perspective, one may well consider that workload abort could be preferred to error notification.

It is worth pointing out that in Table 14.5 several outcomes are grouped into clusters that can be considered as equivalent with respect to a specific measure; each cluster characterizes a relevant "accomplishment level" for the considered measure. These clusters are ranked according to an increasing severity level (i.e., index 1 indicates the most favorable case). We have appended labels (+) and (–) to explicitly indicate what we are considering as positive and negative clusters. However, we recommend keeping the data for each cluster so that a finer tuning of these categorizations is always possible. The rightmost column gives the rationale that defines the various clusters.

14.5. THE EXPERIMENTAL TESTBED: THE RoCADE PLATFORM

Figure 14.3 describes the RoCADE (robustness characterization against driver errors) platform that has been set up for conducting the experiments (only one target machine is shown).

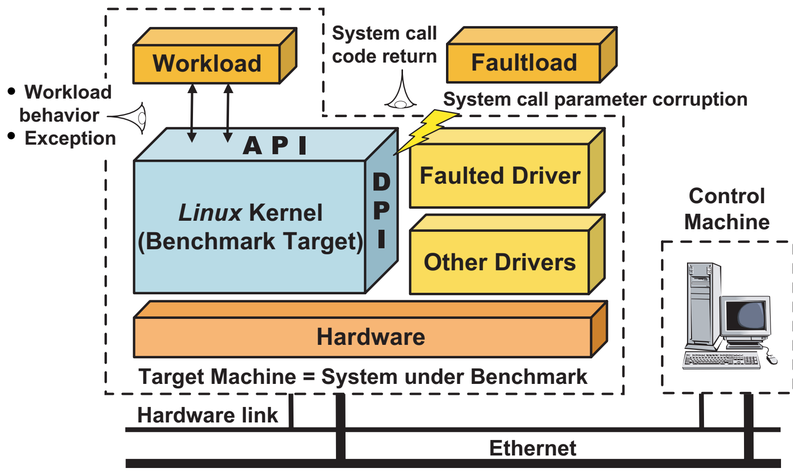

Figure 14.3. Overview of the RoCADE platform.

The experiments were carried out using a rack of four Intel Pentium machines, each featuring 32 Mb of RAM and several commonly used peripheral devices, including a hard disk, a floppy disk, a CD ROM, two network cards, a graphic card, and a keyboard. All four machines run the GNU/Linux distribution. Three of them are the target machines on which faults are injected and behaviors observed; each is supporting two versions of the Linux kernel: 2.2.20 and 2.4.18. The use of three target machines is meant to speed up the conduct of the experiments. The fourth machine (control machine) is connected to the target machines via a private Ethernet network to control the experiments and provide an external means for monitoring these machines. In particular, it is used to restart the target machines, should they be blocked after an experiment. Indeed, for the sake of repeatability, for each experiment the SUB is restored to a specific (fault-free) state.

The injection of faulty parameters into each target machine is carried out via the RAM. The processor uses a stack residing in RAM to store various data, including the parameters of the calls to the functions of the DPI. This stack is accessible via the registers of the processor. At the same time, another area in the memory stores the instructions to be executed. When a DPI function is being used, the processor raises an interrupt. Upon occurrence of this interrupt, the fault injection process takes over; it modifies a parameter in the stack and resumes the execution of the program. When the fault has been applied once, the fault injection process is disabled. The corruptions provoked in this way correspond to transient faults. This choice for the fault model illustrates the kind of pragmatic compromise one has to make among benchmarking properties (e.g., see [Kanoun et al. 2002]), namely here, fault representiveness and low intrusiveness.

In order to recognize the symbols used by the driver, we have developed scripts that automatically extract their names from the driver's object code file. Then, thanks to the list referencing all symbols, we can determine what faults can be injected into these symbols. Hence, all parameters of the selected functions are subjected to fault injection (according to all the fault types defined in Table 14.2). The codes returned after a system call are obtained with similar techniques. The code returned by the symbol subjected to a fault is collected from the stack. In addition to these error codes, the symbols may also display other error messages, such as "blue screen" or "panic." Such error messages are collected at the end of each experiment.

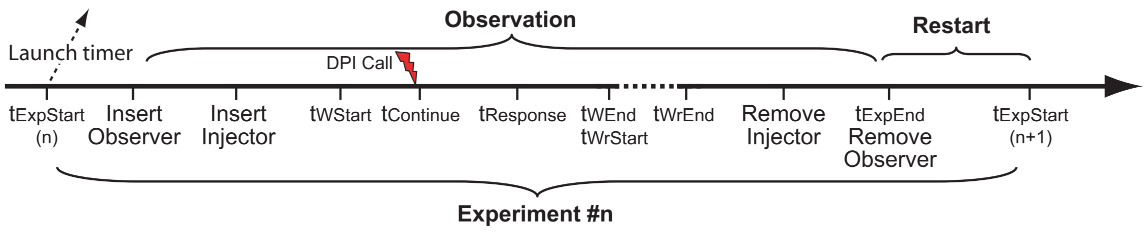

Figure 14.4. Scheduling of relevant events for an experiment.

At the start of each experiment that is indicated by the target machine, the control machine sets a timer. At the end of each experiment, the target machine is rebooted and it is expected to be able to retrigger this timer at the end of the reboot. If the timer overruns, the control machine provokes a hardware restart of the target machine. This situation is interpreted as a hang of the kernel. The hardware exceptions are collected via the log of the target machine. The duration of each fault injection experiment ranges from 2.5 to 5 minutes (the latter when a kernel hang occurs).

The diagram in Figure 14.4 presents the nominal scheduling of a fault injection experiment.

The various important events are identified and described in Table 14.6, where related actions are also detailed.

14.6. RESULTS AND ANALYSES

This section illustrates how the insights one can get from the measurements obtained vary according to the priorities or dependability measures that are considered. We present and analyze a restricted set of results obtained with the RoCADE platform for three representative drivers running on the Linux kernel. Additional results and, in particular, more detailed analyses can be found in [Albinet 2005]. We restrict the presentation of the results to a selected set, in order to facilitate the exposition of the analyses. We voluntarily emphasize two drivers supporting the network card (namely the SMC-ultra and the Ne2000). Network drivers account for the largest part of the code among the drivers whose size is increasing the most [Godfrey and Tu 2000]; we consider also another driver (namely, SoundBlaster) that ranges in the midsize category. An additional set of drivers has been tested (e.g., file system, process memory, etc.).

The main goal that supports the selection of this set of results is to be able to carry out the following types of analyses on:

- Two distinct drivers running on the same version of the kernel: SB + Linux 2.2 and SMC + Linux 2.2.
- Two implementations of the same functionality running on the same kernel version: SMC + Linux 2.4 and NE + Linux 2.4.
- The same driver* running on two different kernel versions: SMC + Linux 2.2 and SMC + Linux 2.4.

*It is worth pointing out that the code of the driver is adapted to fit each version of the kernel.

TABLE 14.6.—Detailed description of events and related actions

IDs	Events	Actions
tExpStart Insertion X*	System verification Setup of the modules of the tool and selection of the fault to be injected	Launch of e2fsck utility to check the file system integrity† Countdown start (on the control machine) Insertion of a breakpoint
tWStart	Initiation of the workload	Start-up of the workload
DPI call	Injection of the fault on the targeted kernel function call Wait for (error) code returned by the kernel function	Raising of an interrupt and injection of the fault Insertion of the breakpoint for observing the returned code
tContinue	Resumption of the workload after execution of the function being faulted	Observation (internal) of the error code returned
tResponse	Observation of the events perceived externally	Collection of the results provided by the workload
TWEnd	Termination of the workload	Signaling of workload termination
tWrStart	Initiation of the replay workload	Start up of the replay workload
tWrEnd	Termination of the replay workload and observation of the related events perceived externally	Signaling of replay workload termination and collection of the related results provided
TexpEnd Removal X*	End of current experiment	Removal of the modules of the tool and restart

*X= Injector, Observer.

†This proved a very useful procedure as in several instances the file system had been damaged due to the corruption of the system call.

Based on the workload and fault types considered, about 100 experiments were carried out for each driver plus kernel combination.

14.6.1. Basic Results and Interpretation

Table 14.7 illustrates the distribution of the basic results obtained when considering the "first event" approach to diagnosing outcomes for which multiple events were collected.

In addition to the specific events previously defined (see Table 14.4), two interesting outcomes are included:

1. Not Activated (Not Act.)—injected faults could not be activated (i.e., the workload was not able to activate the function on which the fault was meant to be injected).
2. No Observation (No Obs.)—none of the notification or failure modes events were observed; of course when the fault is not activated, none of these events can be observed.

The proportion of "Not Activated" cases varies significantly, both among the tested drivers and Linux versions—from 0% (SB + Linux 2.2) to 17% (SMC + Linux 2.4). The fact that in most cases nonnull ratios are observed means that the respective workloads

TABLE 14.7. Distribution of events according to first event collected

Driver + kernel	Not Act.	No Obs.	EC	XC	KH	WA	WI
SB + Linux 2.2	0%	18%	47%	22%	3%	1%	9%
SMC + Linux 2.2	7%	22%	19%	23%	21%	0%	9%
SMC + Linux 2.4	17%	17%	21%	34%	11%	0%	0%
NE + Linux 2.4	14%	10%	15%	30%	17%	0%	13%

Legend—EC: error code, XC: exception, KH: kernel hang, WA: workload abort, WI: workload incorrect.

have to be improved from a *testability* viewpoint—more precisely, *controllability* here. However, these rates are much lower than those reported in related studies on the Linux kernel (e.g., see [Gu et al. 2003]). To our understanding, this better *controllability* is most likely due to the fact that, in our case, faults are targeting the parameters of the system calls made by the driver, rather than the flow of execution of the whole kernel. In the sequel, for further analyses, we will normalize the results presented with respect to experiments in which faults were actually activated.

As already pointed out, the interpretation of the "no observation" cases is highly subject to the specific context in which the analysis is conducted. These outcomes may be counted either as positive or negative depending on the responsiveness, safety, and availability viewpoints. However, as is commonly accepted in testing scenarios, uncertainties still remain about the real situations that such outcomes describe. Accordingly, we have preferred to adopt a conservative approach that consists in ignoring these outcomes. Besides the "replay" mode has been devised to increase the confidence in our analyses, a "No Obs" outcome probably still reveals a lack of *observability* of the tests conducted. But, such an outcome may also be due to controllability-related problems: the kernel does not (or no longer) use(s) the faulted parameter, the faulted parameter has no impact on the kernel, or the error provoked is masked (in our case, injecting a "0" value on a parameter already equal to "0," etc.). However, although the "No Obs." ratios reported are higher than the "Not Act.", the values are significantly lower than the ones presented in [Gu et al. 2003].

Figure 14.5 illustrates the relative distribution among the events observed while considering the "first event" collected, which is a classical approach in most related experimental studies.

A quick examination of these results shows a very low proportion of workload aborts for all tests conducted. The results also reveal that a large percentage of experiments are notified by the kernel (this includes the error code and exception events). Should it be possible to handle equally both types of notifications, then, as the provision of an error code usually features a lower latency, such a notification would be preferable to an exception in order to carry out a successful recovery action. Accordingly, in that respect, the results for SB + Linux 2.2 are more positive than those observed for the experiments concerning network card drivers. However, adopting an end-user perspective would lead to a different assessment; indeed, in that case only exceptions would actually matter.

The comparison of the results obtained for the SMC driver for the two releases indicates clearly an improvement of the robustness for SMC + Linux 2.4 due to the increased percentage of exceptions raised. This results in a reduction of the ratios of kernel hangs and, more importantly, in the "disappearance" of critical cases in which a workload incorrect (WI) event was reported. Indeed, due to the precedence in the collection of the events, the fact that a WI event is counted as a first event means that neither a notification

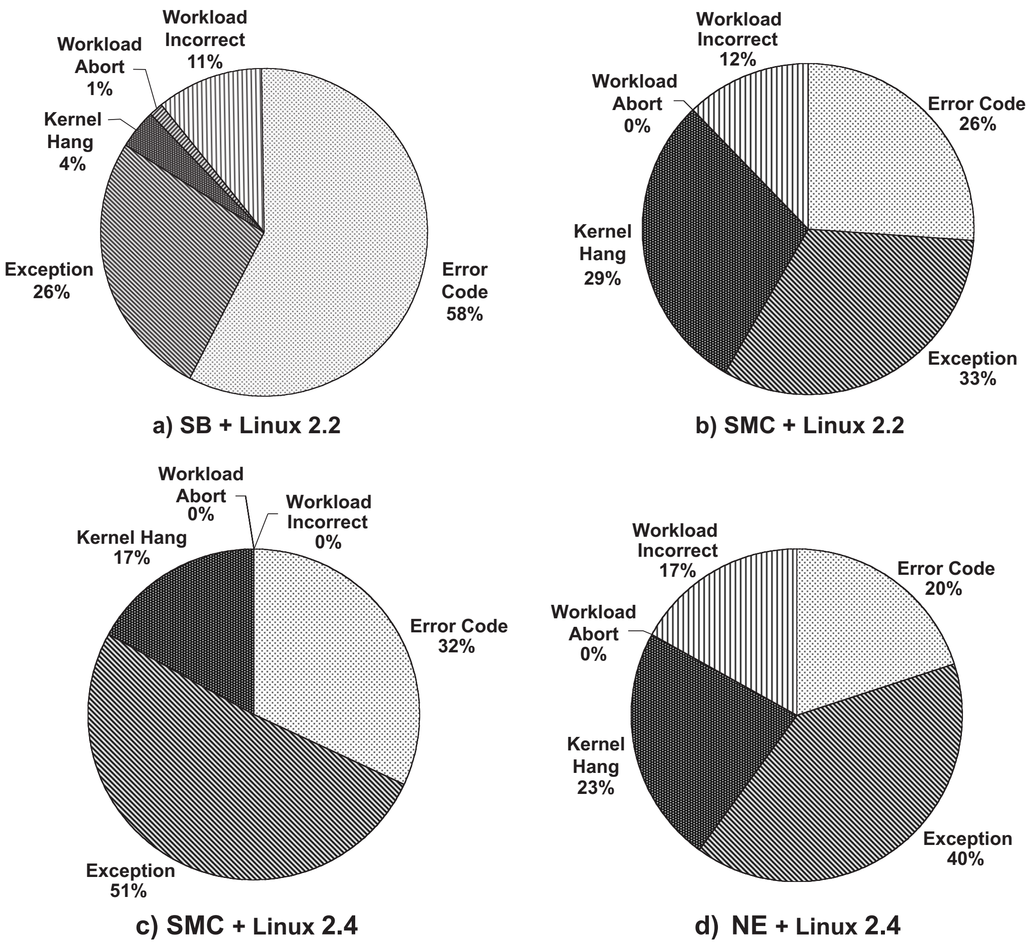

Figure 14.5. Distribution among observed events according to first event collected.

has been made nor an abort has been observed. It is also very likely to be the only event to be collected, unless a hang has occurred after the end of the workload (such cases are actually very infrequent), but, in practice, a deeper analysis of the data collected is necessary to ascertain this statement.

14.6.2. Impact of the Comprehensive Viewpoints

In this section, we revisit the observations made during the conducted experiments in the light of the three comprehensive viewpoints defined in Section 14.5.2. Figure 14.6 summarizes the corresponding measures for the four series of experiments reported here. In each case, the percentages of the various clusters that support the corresponding measure are detailed. It is important to note that the clusters corresponding to the most positive outcomes appear on the top of the histograms (light grey), whereas the critical ones are at the bottom (darker shade). Here we consider the set of outcomes defined in Table 14.4.

Let us consider first the kernel responsiveness (RK) viewpoint. Here we assume that RK1 and RK2 form the most positive outcomes (Table 14.5). The distribution observed for SB + Linux 2.2 indicates a very positive behavior—84% of the outcomes observed correspond to error notifications (RK1 = 43% + RK2 = 41%). However, among the 16% of outcomes for which a failure mode was observed without prior notification, more than two-thirds correspond to WI events (RK5 = 11%). The remaining third is dominated by KH events and few WA events. It is worth noting that this is the only set of experiments for which WA events have been diagnosed without any prior notification. The SMC + Linux 2.2 configuration features a much less positive behavior. For example, the distribution shows that 41% of the outcomes observed correspond to failure modes without prior notification. This is mainly due to KH events; indeed, in that case RK4 = 29%, whereas the RK5 cluster (not notified WI events) amounts to a rather similar value (12%). This also means that almost 30% of the workload failures that are not notified led to an incorrect completion. The results shown for SMC indicate that the evolution to release 2.4 has significantly improved the behavior—the percentage of failures without prior notification is reduced to 17% and corresponds to KH events only. Globally, more than three-quarters (60/77) of the failure modes observed are preceded by an error notification. This confirms the observations already made on the basis of the analysis of the pie charts displaying the distributions of the first event collected (Figure 14.5). The results for NE + Linux 2.4 indicate a much lower error notification ratio, which is similar to the one reported for the SMC + Linux 2.2 case.

For the kernel availability viewpoint (AK), the most critical issue is characterized by a KH event, because this has a dramatic impact on the ability to keep delivering the service. This is why AK3 and AK4 are considered the most critical clusters. The results shown for SB + Linux 2.2 indicate that faults have also a significant impact with respect to availability. Indeed, a KH event is observed in 23% (18% + 5%) of the cases. Moreover, the WI event is observed in more than 78% of these cases (the 18% of the 23%); this means that in most cases, before the kernel hangs, the services that are active in the kernel are no longer able to maintain the proper operation of the application processes. The results also indicate that faults associated with network drivers have consistently a very significant impact—about half of the fault injection experiments conclude with a KH event. For the SMC driver, the results also show that the modifications made between the two releases had no impact from the availability viewpoint.

Concerning the workload safety viewpoint (SW), what matters most is the ability to avoid the delivery of incorrect results. This is why SW4 and SW5 are considered as the most critical clusters. The results shown for SB + Linux 2.2 suggest a much less positive behavior than was deduced from the analysis of the results from the RK viewpoint—the occurrence of the most severe case (cluster SW5 = WI and no KH) amounts to 22%. In addition, it is worth noting that the significant improvement observed with respect to responsiveness (error notification) between SMC + Linux 2.2 and SMC + Linux 2.4 has no impact (actually slightly the opposite) in reducing the WI events. Such a behavior can be explained by the fact that most additional error notifications correspond to exceptions, rather than error code returns (see Table 14.7). As a matter of fact, such exceptions signal already severe erroneous behavior; the erroneous behavior is reported, but no suitable recovery procedure is being launched. The rather poor behavior observed with respect to responsiveness is also confirmed by the analysis with respect to the safety viewpoint; in 53% of the cases (SW5 = 42% + SW4 = 11%) the observed outcome is a WI event, which is the most critical event here.

Figure 14.7 illustrates how these various viewpoints, and the associated properties, can be used by system integrators in making a decision as to whether to incorporate a driver into their system. The histograms plot the percentage for cases in which these properties

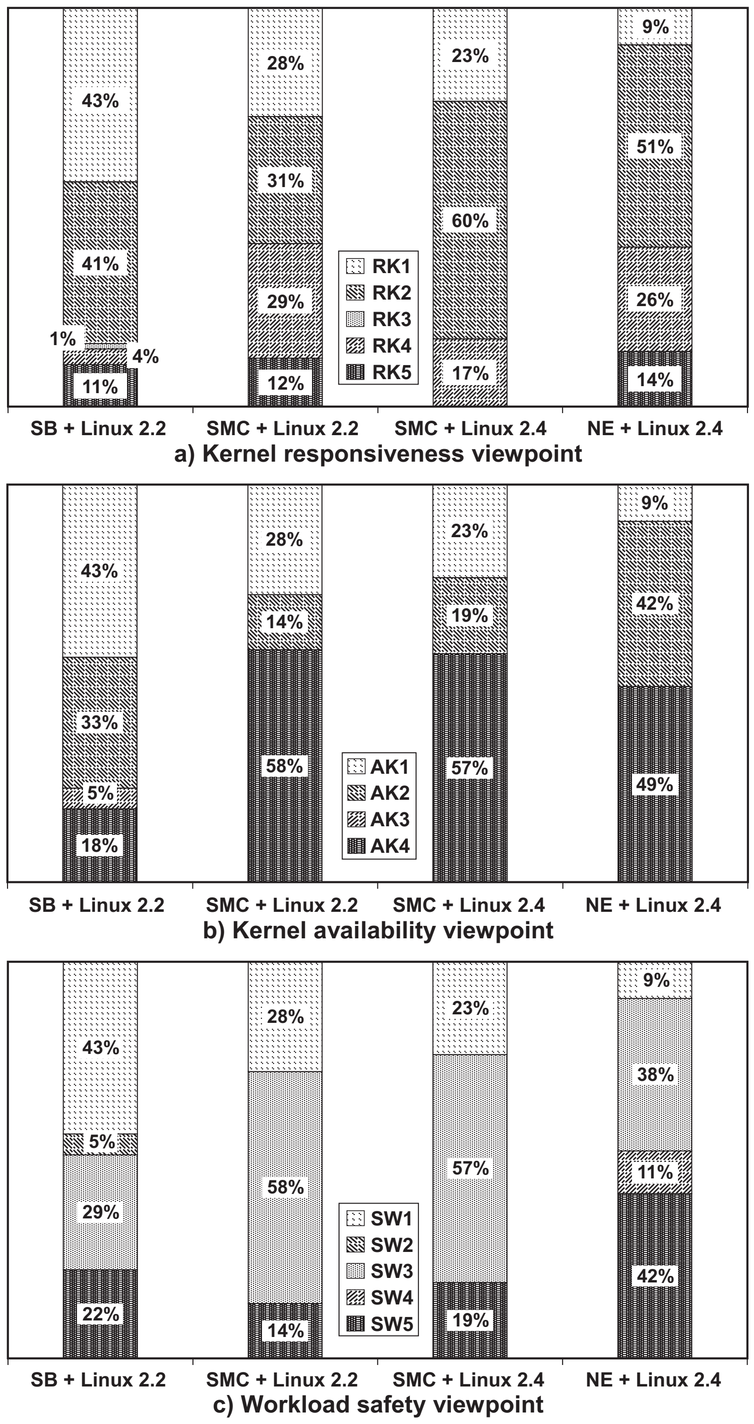

Figure 14.6. Interpretation of the results according to the considered viewpoints.

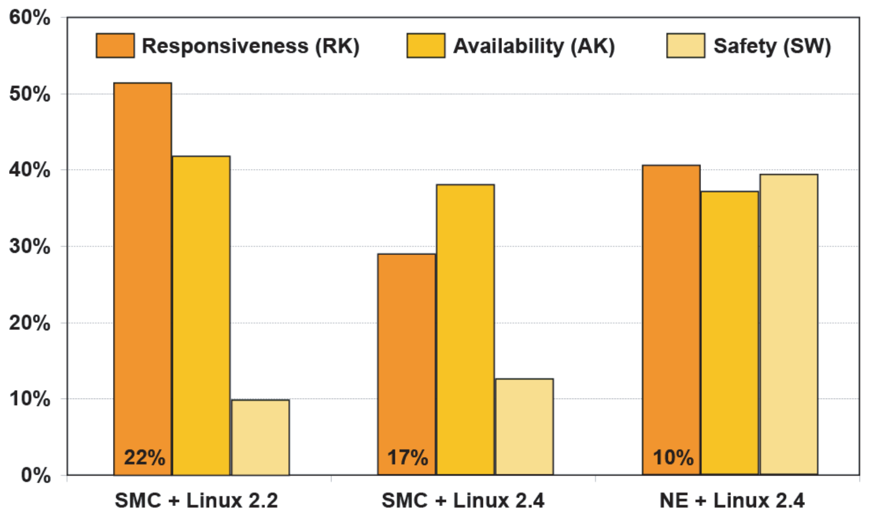

Figure 14.7. Comparison of the property deficiencies induced by the network card drivers.

were not verified (i.e., the cases corresponding to the clusters labeled with index "–" in Table 14.5) when considering the experiments involving the tested network drivers. In this case, the figures being considered explicitly account for the ratios of "No Obs." that appeared in Table 14.7. It is worth noting that these ratios contribute negatively to the evaluation of the responsiveness property (i.e., RK is not verified*); however, they correspond to the verification of AK and SW.

The histograms concerning the two versions of the SMC driver clearly illustrate that the significant decrease in lack of error signaling obtained for version 2.4 does not result in a significant reduction in the weaknesses with respect to the other viewpoints (actually, a slight increase is observed for safety); the improvement in the coverage procured by the error detection mechanisms was not accompanied by an improvement in the handling of error signals. For example, the application of the concept of *shadow driver* reported in [Swift et al. 2004] would help improve the behavior by complementing such a *problem-revealing only* strategy with a specific low-level *recovery* strategy. During normal operation, the shadow driver tracks the state of the real driver by monitoring all communication between the kernel and the driver. When a failure occurs, the shadow driver substitutes *temporarily* for the real driver, servicing requests on its behalf, thus shielding the kernel and applications from the failure. The shadow driver then restores the failed driver to a state in which it can resume processing requests. It is also interesting to observe that, while the network driver NE 2000 features similar or slightly better behaviors than the SMC driver with respect to responsiveness and availability, it exhibits a much poorer behavior with respect to safety. This reflects the fact that very distinct implementation choices were made for these two drivers.

*This is consistent with the rationale underlying this viewpoint (i.e., a reaction from the kernel is expected in presence of activated faults). Still, considering the complete "No Obs." ratio as contributing to a deficiency of the RK property might lead to a pessimistic assessment. This is why the related percentages are explicitly shown in the figure.

14.7. CONCLUSION

Popular operating systems (COTS or open source) rapidly evolve into increasingly complex software components. Drivers are known to account for the major part of the increase in terms of lines of source code. These components are often crafted by third-party developers and then integrated within the operating system. This whole process is not always well mastered, as evidenced by the vast consensus that attributes a large proportion of operating system failures to driver malfunctions.

The work reported in this chapter proposed a practical approach to benchmarking the robustness of operating systems with respect to faulty device drivers. In order to facilitate the conduct of fault injection experiments, we have introduced the notion of the driver programming interface (DPI) that precisely identifies the interface between the drivers and the kernel, in the form of a set of kernel functions. In the same way that the API is used to simulate the consequences of faulty application processes, the DPI provides a suitable interface for simulating the potential erroneous behaviors induced by a faulty driver. In practice, we have used a SWIFI technique to corrupt the parameters of these functions. In order to collect relevant outcomes for a detailed characterization of the faulty behaviors, we have considered both internal (error codes returned by the kernel) and external measurements (e.g., exceptions raised, kernel hangs, and workload behavior).

To analyze the experimental results, we have proposed a comprehensive framework for interpreting the results that accounts for several dependability viewpoints. We have considered three viewpoints, namely, responsiveness of the kernel (maximize error notification), availability (minimize kernel hangs), and safety of the workload (minimize delivery of incorrect service). They provide a practical means for analyzing three different facets of the dependability requirements that one can expect from a robust operating system, either simultaneously or individually.

In order to illustrate and assess our approach, we have set up an experimental platform, RoCADE (robustness characterization against driver errors). We have focused here on the series of experiments conducted on two releases of the Linux kernel (2.2.20 and 2.4.18) and on three drivers [sound (sound blaster) and network (SMC and NE 2000)]. The analyses carried out have shown that although the sound blaster driver got a very good rating according to responsiveness, it exhibited poor behavior with respect to the safety and availability viewpoints. The experiments conducted on the SMC driver were able to reveal a significant improvement with respect to responsiveness between the two releases considered, but this did not result in any improvement from the safety and availability viewpoints. Finally, we identified a slightly better behavior concerning availability for the experiments conducted on the NE 2000 driver than for those on the SMC driver, whereas the opposite was obtained for safety and responsiveness.

The results we have obtained and the analyses we have carried out thanks to RoCADE are encouraging with regard to viability of the proposed methodology. The whole approach can thus be considered as a sound basis on which to develop a set of practical dependability benchmarks focusing on the characterization of the impact of faulty drivers on the behavior of an operating system kernel. As witnessed by the insights gained from the measures obtained, although the proposed framework is primarily geared toward the characterization of kernel behaviors, it is also suitable to support the choice of drivers to be associated with a given kernel.

We consider the fact that a large proportion of error codes had been observed (especially as first-collected events) as a positive result in order to perform a detailed characterization of the erroneous behaviors induced by the corrupted parameters. In addition, these codes form a useful basis on which specific error handling could be implemented. Another recommended approach to restrict the impact of faulty drivers would be to enforce a clear separation between the driver address space and the kernel address space (e.g., see [Härting et al. 1997]). The use of specific languages excluding pointer arithmetic and explicitly including IMM (e.g., see [Réveillère and Muller 2001]) is another promising approach to develop more robust drivers. More recently, several proposals have been made to attain a clear separation of concern using virtual machine constructs; for example, see [Fraser et al. 2004, LeVasseur et al. 2004]. The contemporary Nooks approach and its extension in the form of shadow drivers [Swift et al. 2004] offer other attractive approaches.

Finally, it is worth pointing out that the notion of the DPI (driver programming interface) that we have advocated and defined in order to structure the conducted experiments matches very well the concept of separation of concerns that underlies several frameworks that were proposed recently, both by academic studies (e.g., see [Swift et al. 2004]) and by an increasing number of operating system and hardware manufacturers. Let us simply mention the various CDI (common driver interface), DDI (device driver interface), or DKI (driver kernel interface) proposals that have been put forward for several operating systems. Among these initiatives, the Extensible Firmware Interface (EFI) that was recently promoted by the Unified EFI Forum* as an emerging standard deserves special attention. The EFI defines a new model for the interface between operating systems and platform firmware. The UEFI is primarily meant to provide a standard environment for booting an operating system. Nevertheless, the data tables (containing platform-related information, plus boot and run-time service calls) that implements it can be useful also to facilitate run-time access to internal variables and, thus, better structure the design of device drivers. Accordingly, it should be possible to reuse the principles underlying the DPI identified herein and/or to adapt them easily in the forthcoming arena that this emerging standard is promising for structuring the interactions between the operating systems and the related hardware layers, including the device drivers.

ACKNOWLEDGMENT

This work was partly supported by the European Commission (Project IST-2000-25425: DBench and Network of Excellence IST-026764: ReSIST). Arnaud Albinet was supported in part by the Réseau d'Ingénierie de la Sûreté de fonctionnement (Network of Dependability Engineering); he is now with Continental Automotive, Toulouse, France.

REFERENCES

[Albinet 2005] A. Albinet, *Dependability Characterization of Operating Systems in presence of Faulty Drivers,* Ph.D. Dissertation, National Polytechnic Institute, Toulouse, 2005 (in French, also LAAS Report 05-248).

[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 *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2004),* Florence, Italy, IEEE Computer Science Press, Los Alamitos, CA, pp. 867–876, 2004.

*http://www.uefi.org.

[Arlat et al. 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation—A Methodology and Some Applications," *IEEE Transactions on Software Engineering,* vol. 16, no. 2, pp. 166–182, February 1990.

[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.

[Avižienis et al. 2004] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing,* vol. 1, no. 1, pp. 11–33, Jan.–March 2004.

[Brown and Patterson 2000] A. Brown and D. A. Patterson, "Towards Availability Benchmarks: A Case Study of Software RAID Systems," in *Proceedings of 2000 USENIX Annual Technical Conference,* San Diego, CA, USENIX Association, 2000.

[Carreira et al. 1998] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering,* vol. 24, no. 2, pp. 125–136, February 1998.

[Chou et al. 2001] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating System Errors," in *Proceedings of 18th ACM Symposium on Operating Systems Principles,* Chateau Lake Louise, Banff, Canada, ACM Press, New York, 2001, http://www.cs.ucsd.edu/sosp01.

[Durães and Madeira 2003] J. Durães and H. Madeira, "Mutidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior," *IEICE Transactions on Information and Systems,* vol. E86-D, no. 12, pp. 2563–2570, December 2003.

[Durães and Madeira 2006] J. Durães and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Transactions on Software Engineering,* vol. 32, no. 11, pp. 849–867, November 2006.

[Edwards and Matassa 2002] D. Edwards and L. Matassa, "An Approach to Injecting Faults into Hardened Software," in *Proceedings of Ottawa Linux Symposium,* Ottawa, ON, Canada, pp. 146–175, 2002.

[Fraser et al. 2004] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe Hardware Access with the Xen Virtual Machine Monitor," in *First Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS),* Boston, 2004.

[Fraser et al. 2003] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers," in *Foundations of Intrusion Tolerant Systems—Organically Assured and Survivable Information Systems (OASIS),* J. H. Lala (Ed.), IEEE Computer Science Press, Los Alamitos, CA,pp. 399–413, 2003.

[Godfrey and Tu 2000] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *Proceedings of IEEE International Conference on Software Maintenance (ICSM-200),* San Jose, CA,IEEE Computer Science Press, Los Alamitos, CA, pp. 131–142, 2000.

[Gu et al. 2003] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux Kernel Behavior under Errors," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2003),* San Francisco, CA, IEEE Computer Science Press, Los Alamitos, CA, pp. 459–468, 2003.

[Härting et al. 1997] H. Härting, M. Ohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The Performance of μ-Kernel-Based Systems," in *Proceedings of 16th ACM Symposium on Operating Systems Principles (SOSP-16)* Saint-Malo, France, pp. 66–77, 1997.

[Jarboui et al. 2003] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau, "Impact of Internal and External Software Faults on the Linux Kernel," *IEICE Transactions on Information and Systems,* vol. E86-D, no. 12, pp. 2571–2578, December 2003.

[Johansson and Suri 2005] A. Johansson and N. Suri, "Error Propagation Profiling of Operating Systems," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and*

Networks (DSN-2005), Yokohama, Japan, IEEE Computer Science Press, Los Alamitos, CA, pp. 86–95, 2005.

[Kalakech et al. 2004] A. Kalakech, T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun, "Benchmarking Operating System Dependability: Windows 2000 as a Case Study," in *Proceedings of 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC-2004),* Papeete, French Polynesia, EEE Computer Science Press, Los Alamitos, CA, pp. 261–270, I2004; see also http://www.laas.fr/DBench.

[Kanoun et al. 2002] K. Kanoun, H. Madeira, and J. Arlat, "A Framework for Dependability Benchmarking," in *Supplemental Volume of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2002)—Workshop on Dependability Benchmarking,* Washington, DC, pp. F.7–F.8, 2002; see also http://www.laas.fr/DBench.

[Kanoun et al. 2005a] K. Kanoun, H. Madeira, M. Dal Cin, F. Moreira, and J. C. Ruiz Garcia, "DBench (Dependability Benchmarking)," in *Proceedings of 5th European Dependable Computing Conference (EDCC-5)—Project Track,* Budapest, Hungary, 2005; available as LAAS Report no. 05197, see also http://www.laas.fr/DBench.

[Kanoun et al. 2005b] K. Kanoun, Y. Crouzet, A. Kalakech, A. E. Rugina, and P. Rumeau, "Benchmarking the Dependability of Windows and Linux Using Postmark Workloads," in *Proceedings of 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005),* Chicago, IEEE Computer Science Press, Los Alamitos, CA, pp. 11–20, 2005.

[Koopman and DeVale 1999] P. Koopman, and J. DeVale, "Comparing the Robustness of POSIX Operating Systems," in *Proceedings of 29th IEEE International Symposium on Fault-Tolerant Computing (FTCS-29),* Madison, WI, IEEE Computer Science Press, Los Alamitos, CA, pp. 30–37, 1999.

[LeVasseur et al. 2004] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," in *Proceedings of 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI '04),* San Francisco, USENIX Association, pp. 17–30, 2004.

[Madeira et al. 2002] H. Madeira, R. Some, F. Moreira, D. Costa, and D. Rennels, "Experimental Evaluation of a COTS System for Space Applications," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2002),* Washington, DC, EEE Computer Science Press, Los Alamitos, CA, pp. 325–330, I2002.

[Marsden et al. 2002] E. Marsden, J.-C. Fabre, and J. Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection," in *Proceedings of 21st IEEE International Symposium on Reliable Distributed Systems (SRDS-2002),* Osaka, Japan, IEEE Computer Science Press, Los Alamitos, CA, pp. 276–285, 2002.

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

[Murphy and Levidow 2000] B. Murphy and B. Levidow, "Windows 2000 Dependability," in *Digest of Workshops and Abstracts of the IEEE/IFIP International Conference on Systems and Networks (DSN-2000),* New York, pp. D.20–D.28, 2000.

[Réveillère and Muller 2001] L. Réveillère and G. Muller, "Improving Driver Robustness: An Evaluation of the Devil Approach," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2001),* Göteborg, Sweden, IEEE Computer Science Press, Los Alamitos, CA, pp. 131–140, 2001.

[Rodríguez et al. 2002] M. Rodríguez, A. Albinet, and J. Arlat, "MAFALDA-RT: A Tool for Dependability Assessment of Real Time Systems," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2002),* Washington, DC,IEEE Computer Science Press, Los Alamitos, CA, pp. 267–272, 2002.

[Rodríguez et al. 2003] M. Rodríguez, J.-C. Fabre, and J. Arlat, "Building SWIFI Tools from Temporal Logic Specifications," in *Proceedings of IEEE/IFIP International Conference on Depend-*

able Systems and Networks (DSN-2003), San Francisco, pp. 95–104, IEEE Computer Science Press, Los Alamitos, CA, 2003.

[Swift et al. 2004] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering Device Drivers," in *Proceedings of 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI '04),* San Francisco, pp. 1–16, 2004, USENIX Association, http://nooks.cs.washington.edu.

[Tsai et al. 1996] T. K. Tsai, R. K. Iyer, and D. Jewitt, "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems," in *Proceedings of 26th International Symposium on Fault-Tolerant Computing (FTCS-26),* Sendai, Japan, EEE Computer Science Press, Los Alamitos, CA, pp. 314–323, I1996.

[Vieira and Madeira 2003] M. Vieira and H. Madeira, "Benchmarking the Dependability of Different OLTP Systems," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2003),* San Francisco, IEEE Computer Science Press, Los Alamitos, CA, pp. 305–310, 2003.

[Zaatar and Ouaiss 2002] W. Zaatar and I. Ouaiss, "A Comparative Study of Device Driver APIs: Towards a Uniform Linux Approach," in *Proceedings of Ottawa Linux Symposium,* Ottawa, ON, Canada, pp. 407–413, 2002.