JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 23, 1241-1263 (2007)

*n*Driver: Online Driver Replacement for Increasing Operating System Availability*

DA-WEI CHANG, ZHI-YUAN HUANG+ AND RUEI-CHUAN CHANG+
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701 Taiwan
+Department of Computer Science
National Chiao Tung University
Hsinchu, 300 Taiwan

Device drivers are the most unreliable part of an operating system. In this paper, we propose a framework called *n*Driver. Based on the design diversity concept, it uses multiple implementations of a device driver to survive from driver faults. Once a fault happens in a driver, *n*Driver can dynamically replace the faulty driver with another implementation, instead of allowing the faulty driver to crash the system. The unique features of *n*Driver are as follows. First, it can detect two major kinds of driver faults, the exception and blocking faults. Second, the requests issued to the driver will not be lost due to the driver replacement. Third, the driver replacement is transparent to all the other kernel subsystems. Fourth, *n*Driver requires no modification to the existing operating system or driver codes.

The major contribution of this work is that *n*Driver implements the concept of design diversity at the device driver layer. Moreover, it achieves the goal of seamless driver replacement and improves operating system availability without modifying the existing operating system or driver codes.

We implemented *n*Driver as a kernel module in Linux. Currently, it can recover the system from faults in network device drivers. However, the mechanisms can be adapted to other module-based device drivers with a slight extension. According to the performance evaluation, the overhead of *n*Driver is no more than 3.5% and the recovery time is quite small. This indicates that *n*Driver is an efficient mechanism to increase the availability of an operating system.

Keywords: fault recovery, device driver, design diversity, driver replacement, operating system availability

1. INTRODUCTION

With the current high reliance on computer systems, system availability is increasingly important, and for a growing number of systems, keeping them always available is no longer optional but mandatory. According to previous research, software faults account for a larger portion of system unavailability than hardware failures [1-4]. Moreover, the latter can generally be masked through component redundancy [5-8]. Therefore, software plays a critical role in system availability.

Since most of the software relies on the underlying operating systems, reliability of an operating system is critical for a highly available computer system. Unfortunately, due

Received May 25, 2005; revised November 2, 2005; accepted January 9, 2006.
Communicated by Michael R. Lyu.

to the high complexity of operating systems, it is nearly impossible to make them error-free. The most error-prone part of an operating system is the device drivers. It has been shown that the error rate of device drivers can be three to seven times higher than that of the other kernel subsystems [9, 10]. This is because most drivers are developed by the engineers of the hardware device vendors, who are not as familiar with kernel programming as the original kernel developers are.

Since a device driver is a part of the kernel, a fault occurring in a driver is a kernel fault, and it usually results in a kernel panic or a system hang. This causes the services running on top of the operating system to become unavailable. However, a faulty driver usually does not pollute the state of the other kernel subsystems. Therefore, it is possible to recover the system from the driver faults, and hence allow the services running on top of the kernel remain available.

In this paper, we propose a framework called *n*Driver to enable device drivers surviving from software faults. We address on two kinds of faults, *blocking* and *exception* faults. The former leads to kernel hang, while the latter causes kernel panic. According to a previous study [9], these two kinds of faults are responsible for most of the faults happening in a driver.

The basic idea of the *n*Driver approach is to try another driver implementation if the current one fails. Based on the design diversity concept [11, 12], we use multiple driver implementations for a device so that if the current driver fails, *n*Driver can detect it and replace the faulty driver with another one.

Multiple driver implementations can be obtained in the following ways. First, there are usually patches for a driver implementation, and by applying these patches to the original driver implementation, another implementation is produced. Second, there may be multiple driver implementations (from different developers) for a device or multiple devices that use a common chipset. For example, the Linux kernel 2.4.2 includes two drivers for the RealTek RTL8129 Fast Ethernet device. Third, there may exist a generic but regressive driver for the device. For example, the ne2000 NIC (Network Interface Card) device driver can also be used to drive many NICs of different vendors.

To achieve the goal of seamless driver replacement, the following requirements must be satisfied.

- Non-stop services. The services or applications running on top of the system should continue running without interruption even when a driver fails.
- Automatic fault detection. Blocking and exception faults should be detected automatically, without the help of the system administrators.
- Zero-loss system requests. Generally, removing a driver causes the loss of its internal data, including the requests issued to it. However, to achieve the goal of seamless driver replacement, all the uncompleted requests should be retained and then re-issued to the new implementation.
- Kernel state maintenance. A driver may have made changes to the global kernel state (*e.g.*, it may have requested some kernel resources). Therefore, the kernel state should be recovered when the faulty driver is removed. Moreover, all the external references to the original driver should be redirected to the new one. Otherwise, the kernel will be likely to crash due to these dangling references.

In this paper, we will show how the *n*Driver framework satisfies the above requirements. We have implemented a prototype of this framework in the Linux kernel, and currently it can survive NIC driver faults. We believe that the mechanisms can be adapted to other module-based device drivers with a slight extension. According to the experimental results, *n*Driver leads to only little performance overheads (*i.e*., less than 3.5% in the case of a Gigabit Ethernet driver). Moreover, the recovery time is quite small. This indicates that *n*Driver is very efficient for increasing the availability of an operating system.

The rest of this paper is organized as follows. In section 2 we describe our design issues and the flow of recovering a device driver fault, which is followed by the description of the implementation details in section 3. Section 4 shows the experimental results. Section 5 presents the limitations and further extensions of our current implementation. Section 6 shows the related work, which is followed by the conclusion presented in section 7.

2. DESIGN

In this section, we elaborate on the method of surviving from driver faults. When a fault occurring in the driver is detected, the recovery mechanism is triggered. Briefly speaking, the recovery process involves removing the faulty driver, undoing the changes caused by it, inserting the new driver, reconfiguring it, and retrying the previously-failed function in the new driver.

The same fault may not occur again in the new driver due to the following reasons. First, if the fault happens due to a known bug that has been fixed in the new driver, the fault will not happen again. Second, the new driver may be implemented by different developers so that the same bug can seldom happen in both the old and the new drivers.

Fig. 1 illustrates the components of the *n*Driver. The guard wrapper and the fault detector are responsible for detecting exception and blocking faults. The undo manager is responsible for removing the faulty driver, undoing the changes it made, and inserting the new driver. Finally, the configuration manager is responsible for reconfiguring the new driver.

In the following sections, we provide a detailed description for the *n*Driver framework. First, we will present the fault detection approach, which is followed by the description of how to keep the system state correct and consistent after a fault occurs. Then, we present the approach for solving the problem of dangling references. Finally, we describe the detailed flow of the recovery process.

2.1 Fault Detection

The fault detector is responsible for detecting exception and blocking faults. An exception fault occurs when the driver performs an operation such as accessing a NULL page, dividing the operand by zero, executing an invalid opcode, *etc*. To detect such faults, we replace the kernel exception handlers (such as the page-fault and the divide-by-zero handlers) with our own ones. Therefore, raising a CPU exception will trigger our exception handler, which will then invoke the undo manager to recover the fault.

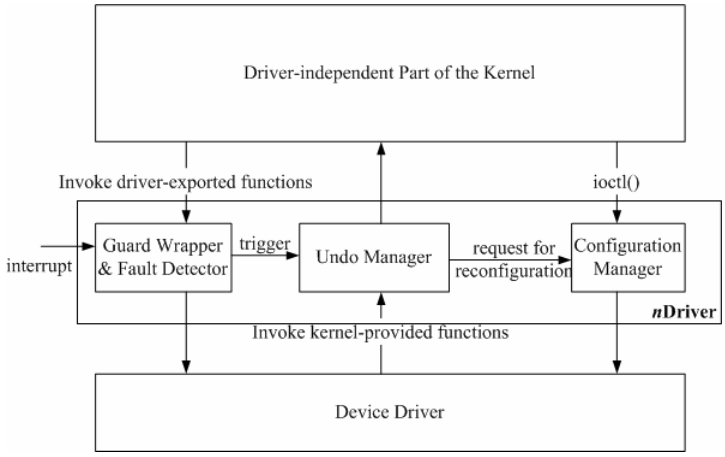


Fig. 1. Architecture overview.

In addition to exception faults, a faulty driver may also lead to system hangs (*i.e.*, blocking faults), which cause the system not to respond. Blocking faults usually result from careless driver design such as entering an infinite loop (*e.g.*, *for*, *while*, or *goto* loops) or trying to get a spinlock which is grabbed by another blocked kernel thread. To recover from such faults, we use a timeout-based approach. Before executing a driver function, we set up a software timer in order to measure the time it takes to execute the driver function. If the driver function occupies the CPU for a long time, it will be regarded as a faulty function and the time-out handler will be triggered to recover from the fault. The accounting of the execution time is through timer interrupts, which occur every 10ms.

Although the approach based on time-outs is straightforward, two issues must be addressed to make it become an effective technique for preventing driver hangs. The first issue is how to determine the time-out value of a driver function. Because the execution time of different driver functions varies, we can not use a fixed time-out value for all the driver functions. Instead, the time-out value of a driver function is set to its average execution time plus a guard time.[1] Note that the time-out values are not required to be highly accurate, and the 10-ms granularity is accurate enough for detecting blocking faults.

Another issue is how to prevent the software timer approach from becoming useless if the driver function disables interrupts. This is possible since many existing drivers disable interrupts for synchronization. To solve this problem, we replace the original interrupt-disabling/enabling functions, namely cli() and sti(), and the timer interrupt handler. Instead of disabling the interrupt pin of the CPU, the new cli() function masks all the interrupts except for the timer interrupt. In this way, our software timer still works after calling cli(). Note that our timer interrupt handler will not invoke the original timer-interrupt handler when the interrupts are disabled. This preserves the interrupt-disabled semantic.

[1] Currently, we measure the average execution time of the function on the target machine in advance and then set the time-out value accordingly. Obviously, such measurement should be performed again if we want to apply the framework to another machine with different performance. To ease the effort, we plan to make such measurement automatic and integrate it into the *n*Driver framework in the future.

Note that the replacement of the cli()/sti() is achieved by linking the driver with the new version of the functions. Thus, the existing operating system and driver codes are not needed to be modified.

2.2 State Maintenance

We divide the system state that the driver may modify during its execution into three parts: *driver state*, *kernel state*, and *driver requests*. The driver state is the local state of the device driver; the kernel state is the global kernel state that may be changed by the driver; and the driver requests are the requests that are currently processed by the driver and the corresponding device. During the recovery period, we undo the changes the driver made to the kernel state, and the driver state is discarded and rebuilt from scratch. For the driver requests, we record them so that they can be re-issued to the new driver implementation after recovery.

Generally, a driver changes the kernel state through functions provided by the kernel. Such functions may request kernel-managed resources, register a new driver, or exchange information between the driver and the kernel. For example, a driver may acquire IRQs, timers, IO regions, or locks from the kernel. We refer such functions as *callout functions* since the control transfers from the driver to the kernel during the invocations of these functions. Note that the kernel provides a counterpart for each callout function. For example, the callout function request_irq() can be used to acquire an IRQ, and its counterpart free_irq() is used for releasing an IRQ. Such counterparts are referred as *undo routines*. Generally, such undo routines should be invoked during the termination of a driver so as to undo the changes a driver made to the global kernel state. However, they will not be invoked if a driver fault happens. In order to undo the changes, we intercept the invocations of the callout functions and record them in an action list. During the recovery period, the undo routine of each callout function on the list will be invoked in order to undo the changes caused by the function.

It is worth noting that a device driver may invoke only a small subset of kernel-provided functions.[2] This is because the main purpose of a device driver is just to drive the device. For example, a driver does not usually perform IPC operations, which are difficult to rollback.[3] Thus, we focus on the set of functions which may be invoked by the driver, and find out the corresponding undo routines manually.

As mentioned above, we discard the driver state and rebuild it from scratch during the recovery period. The reasons are as follows. First, the driver state is polluted after a fault emerges from the driver. Second, different driver implementations may use different data structures, and thus the old driver state can not directly be used by the new driver implementation. The new driver should implement a state transfer function if it wants to reuse the old state. This implies that all the driver implementations are needed to be modified, which is impractical.

For the driver requests, we backup all the unfinished requests in case they will be lost when the driver fails. Each time the kernel sends a request to the driver, we make a copy of the request and insert the copy to a per-driver unfinished request list. When the

[2] We examined more than 40 drivers and found that the number of kernel functions called by those drivers is 89.

[3] It is not enough to rollback an IPC operation by canceling it or undoing it. The receiver may be triggered by the sent message to take some corresponding actions, which are usually difficult to rollback.

request is finished, the corresponding copy will be removed from the list. If a driver fails, all the requests in the list will be re-issued to the new driver.

Note that it is not necessary to handle the application state since the driver replacement is totally transparent to applications. An application can continue its execution after the driver replacement (unless the faulty driver had corrupted the memory space of the application, which will be discussed in section 5.1). Application processes only experience a short delay when driver replacement happens. Moreover, as we will mention in section 4.1, such short delay may also happen due to process scheduling, disk I/O, or network congestion, which are common in a real system.

2.3 External References

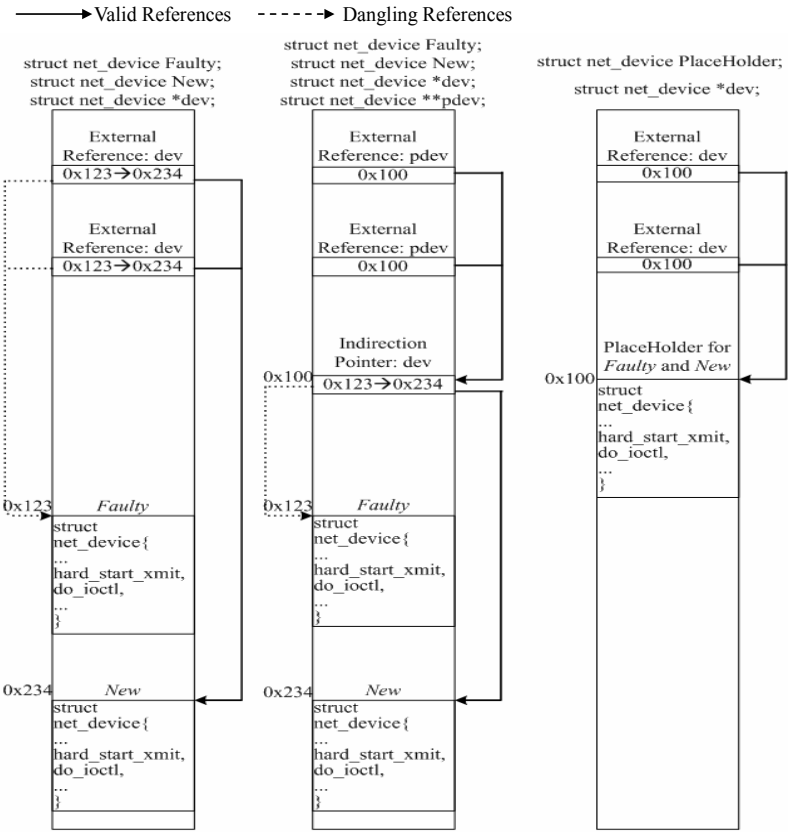
After replacing the faulty driver with the new one, some external references (such as data or function pointers) still point to the data or functions of the original faulty driver. Therefore, we must update all the external references to point to the new implementation. Fig. 2 shows an example. The structure net_device is the main data structure managed by an NIC device driver in Linux. During the recovery process, for instance, the faulty driver is removed and the new driver is inserted and initialized. Thus, all external references to *Faulty* become dangling pointers.

Soules *et al*. [13] proposed two approaches (*i.e*., *backward reference* and *indirection*) as shown in Figs. 2 (a) and (b) to solve this problem. In brief, the backward reference approach keeps track of all external references to *Faulty*, and updates them to point to *New* after the new driver has been inserted. As shown in Fig. 2 (a), the values of all the external references are changed to 0×234. The drawback of this approach is that the operating system must be modified to record all the external references. The indirection approach, as shown in Fig. 2 (b), lets all the external references point to a single indirection pointer. If the target is changed due to the driver replacement, only the indirection pointer needs to be updated. This approach also requires modification to the existing operating system code since the data type of all the external references must be modified (*e.g*., from net_device* to net_device**). Besides, it needs an extra dereferencing to access the target.

In *n*Driver, we take a different approach to avoid modifying the existing operating system code. Fig. 2 (c) shows the approach. We make *Faulty* and *New* share a single memory region (*i.e*., the placeholder), and all the external references point to the placeholder. Thus, the problem of dangling references can never occur. The sharing of the memory region is achieved by intercepting the memory region allocation function. Specifically, when the new driver allocates a net_device structure, the memory region that was previously used for storing the net_device structure of the faulty driver will be returned. In this way, neither the maintaining of the backward references nor the modification to the data types of the external references is necessary.

3. IMPLEMENTATION

To demonstrate the feasibility of the *n*Driver framework, we implemented the mechanisms mentioned in section 2 as a kernel module in Linux. The current implementation



(a) Backward reference. (b) Indirection. (c) Placeholder.

Fig. 2. Solving the external reference problem.

can recover the faults that happen on an Ethernet driver. In the following, we describe the implementation details.

3.1 Fault Detection

3.1.1 Guard wrapper

Since we regard driver functions as unreliable, we put a guard wrapper on each function (including ISR) exported by the driver to prevent a driver fault from crashing or halting the kernel. Each guard wrapper takes the following actions. First, it sets up the fault detection routines. As mentioned above, it substitutes our exception handlers with the original ones for exception faults, and initiates a software timer to measure the time it takes to execute the wrapped function for blocking faults.

Second, the wrapper saves the system context, which is followed by the invocation of the wrapped driver function. If an exception fault happens during the execution of the wrapped function, our exception handler will trigger the recovery process. Similarly, if

the wrapped function does not return before the timer expires, the timeout handler will also trigger the recovery process. If the function returns without faults, the wrapper restores the exception handlers as well as stops the timer.

3.1.2 Software timer

As mentioned in section 2.1, we use an interrupt-based timer to measure the time it takes to execute a driver function. Before we start to execute a driver function, we initialize a counter to the time-out value of the function. Each time the timer interrupt occurs, our software timer decreases the counter by 1. If the counter reaches 0, our software timer will trigger the time-out handler.

Note that a driver function may be preempted by other interrupt handlers, and we should stop counting during the execution of these handlers. However, we did not integrate this technique into *n*Driver. This is because, according to our experimental result, the time used by interrupt handlers (and bottom halves) is quite small compared to the 10-ms timer interrupt interval. Therefore, this time does not have a perceptible impact on the performance of fault detection on our machines.

3.2 Undoing the Kernel State

As we mentioned in section 2.2, we intercept all the callout functions in order to record the changes a driver makes to the global kernel state. The interception is done by linking the object code of the driver module with the interception wrappers before the driver is installed into the kernel. After the linking, all the references to the callout functions are redirected to the corresponding interception wrappers.

We use an action list for each driver to record the invocations of the callout functions. Fig. 3 shows an example of the action list. When an interception wrapper is invoked, we allocate an entry to record the function identifier, the arguments, and the return value. Then, we add this entry to the action list. Keeping the arguments and the return value is necessary since they are needed by the undo routine. For example, the arguments of request_irq() (*i.e.*, irq and dev_id) must be used as arguments of free_irq(), which is the undo routine of the request_irq(), in order to release the allocated IRQ resources. Once the driver invokes the undo routine by itself, the interception wrapper will delete the corresponding entry in the action list. For instance, if a driver calls free_irq(), the interception wrapper will remove the entry for request_irq(). During the recovery process, we remove the entries of the action list in the reverse order of their insertion time. Once an entry is removed, the corresponding undo routine is invoked to undo the kernel state change.

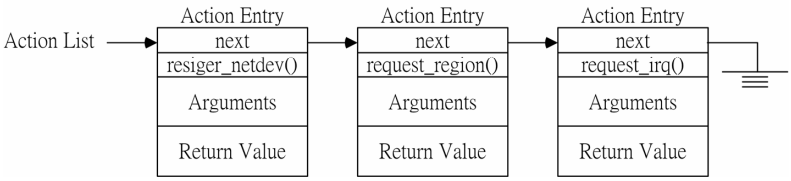


Fig. 3. Structure of the action list.

3.3 Preventing Lost of Driver Requests

In this subsection, we describe how *n*Driver keeps track of the unfinished driver requests in order to re-issue them to the new implementation during the recovery process. We take the Accton EN1207F Series PCI Fast Ethernet driver as an example for illustration.

Fig. 4 illustrates how the driver sends and receives packets. For the sending side, the kernel removes a packet from the send queue of the driver (*i.e.*, qdisc) and hands the packet to the driver. The job of the driver is to insert the packet into its Tx ring buffer and drive the NIC to transmit the packet. For the receiving side, the device receives a packet from the network, puts the packet in its Rx ring buffer, and raises an interrupt to notify the driver. The driver then inserts the packet into the backlog queue for layer-3 processing.

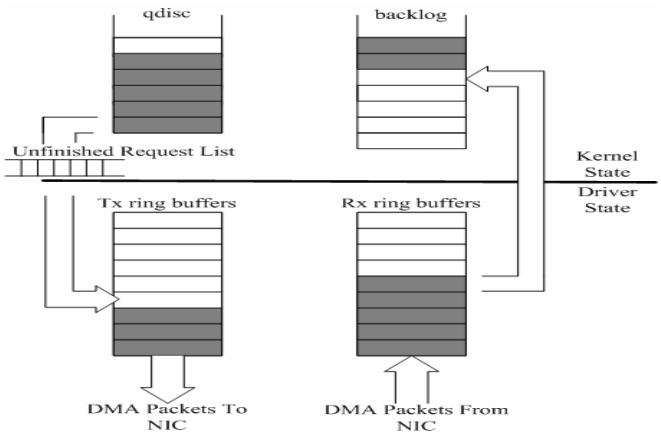


Fig. 4. The data flow of NIC device driver.

Note that the Tx and Rx ring buffers are part of the local driver state. If the driver crashes suddenly, packets in the ring buffers will be lost since we discard the local driver state. To avoid this problem, we maintain an unfinished request list. When the kernel orders the driver to send a packet, we make a copy of the packet and add the copy to the list. When the NIC raises an interrupt to notify that the packet has been sent, we remove the packet from the list. Therefore, after the driver replacement, the packets in the list represent the lost packets and can be re-issued to the new driver.

However, this approach can not be used on the receiving side, where input packets are inserted into the Rx ring buffer via the DMA hardware. The Rx ring buffer can be recovered only if the DMA hardware can notify software before it inserts a packet into the Rx ring buffer, or if we know the address of the Rx ring buffer. However, the former needs special hardware support, while the latter requires digging into the driver code. We do not take these approaches since both of them limit the feasibility of the *n*Driver framework. Instead, we leave the handling of the data loss problem to the upper layers. For example, packet loss can also result from network congestion or overflow of the Rx ring buffer, and it can be resolved by reliable network protocols such as TCP. Therefore,

we consider that losing a small number of Rx packets in layer 2 due to the NIC driver failure is acceptable.

The Rx data loss problem will not happen for block devices. This is because all the read/write operations of a block device are issued by the kernel, instead of the hardware. Therefore, all the requests sent to a block device driver can be intercepted and backed up to the unfinished request list. For character devices, some of the Rx data may be lost due to the reason mentioned above. This problem can be solved by error checking mechanisms such as checksum, CRC, and data-length checking, performed by the upper layers. Although some Rx data may be lost, *n*Driver is still beneficial for such drivers since it opens an opportunity for application programmers to implement non-stop services on those drivers, providing that they use some error checking mechanisms described above.

3.4 Recovery Flow Implementation

In this subsection, we describe details for the process of replacing a module-based NIC device driver in Linux.

As mentioned before, the guard wrapper saves the system context before the execution of a driver function. If a fault is detected during the execution of the driver function, the undo manager will be invoked. The first step of the undo manager is to undo the kernel state changes caused by the driver and to remove the faulty driver module. To undo the kernel state changes and release the resources held by the faulty driver, we invoke the undo routine for each entry in the action list. Although each driver provides functions (*i.e.*, cleanup() and close()) to release its resources, we consider that it is unsafe to execute these functions after a fault has happened in that driver. After the kernel state changes are undone, the kernel function sys_delete_module() is invoked to remove the code and data of the faulty driver module.

Note that undoing the kernel changes may cause some events to be sent to other kernel subsystems in order to signal that the status of the driver has been changed. After the subsystems receive the events, they will take corresponding actions. For example, if we remove a network device driver, any routing table entries depending on it will be deleted. This situation should be prevented since we do not want the rest of the kernel be aware of the driver replacement, so we block the events.

Originally, the second step of the undo manager is to install the code and data of the new driver module into the kernel. However, the installation requires time to load the module from the disk and time to resolve the symbols that the module refers. In order to reduce the recovery time, we load the new driver module (via a modified *insmod* program) into the kernel in advance (*i.e.*, before the fault occurs on the faulty driver). Thus, we can simply locate the new driver module and call its init() function after the faulty module is removed. The init() function usually both resets the hardware and causes some initialization events to be sent to other kernel subsystems. Similar to what we have described above, we also block these events.

After the new driver module is initialized, the configuration manager is asked to reconfigure the driver. During the normal operation period, the configuration manager logs the configuration operations (*i.e.*, ioctl() calls) performed on the faulty driver. Therefore, the reconfiguration can be done simply by performing these operations again to the new driver. After the reconfiguration, the undo manager restores the system context. Finally,

the undo manager retries the previously-failed function if it is not an ISR. An ISR is not retried since it is invoked when the device has some information for the driver (*e.g.*, data is received or data transmission is completed), and the information is lost because of the device reset during the driver replacement. As mentioned in section 3.2.1, this may cause the data loss problem which can be solved by the error checking mechanisms performed by the software layers on top of the driver.

It is worth emphasizing that all the mechanisms described above can be adapted to other types of device drivers (*e.g.*, block device drivers) with a slight extension. This is because Linux defines an interface for each driver type, which can be intercepted to place the fault detectors. Moreover, the mechanisms such as undoing the kernel state, rebuilding the driver state, and preventing requests from being lost are all independent of driver type.

4. PERFORMANCE EVALUATION

In this section, we measure the performance of the *n*Driver framework. The overall goal of the experiments is to show that *n*Driver can make the system survive from driver faults with only little performance degradation. The testbed consists of three machines, one server and two clients. All the machines are connected via a Gigabit Ethernet switch. Each machine is equipped with Pentium 4 2.0GHz CPU, 256MB DRAM. The operating system is Linux (kernel version 2.4.20-8).

4.1 Functionality

In this experiment, we demonstrate that *n*Driver can recover the system when a fault happens in an Ethernet driver. For this, we inserted a fault in the server-side Ethernet driver, and made the client get a file from the server. During the file transfer, we use the tcpdump utility to record the numbers of bytes received by the client, which are derived from the ACK sequence numbers contained in the client-to-server packets. Fig. 5 shows

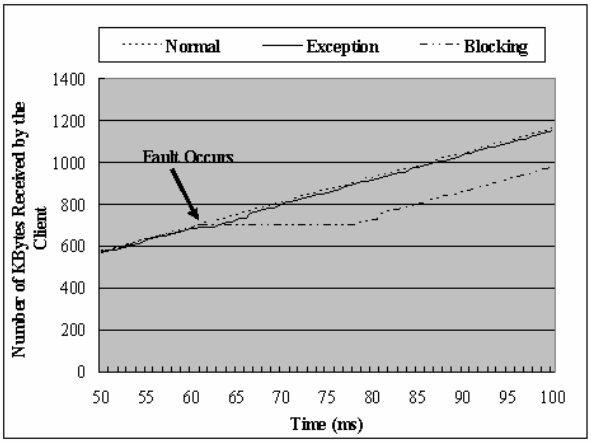


Fig. 5. Functionality of *n*Driver.

the results. The *Normal* line represents the case that no fault happens during the file transfer. The *Exception* and *Blocking* lines represent the cases that an exception fault and a blocking fault happen, respectively. The exception fault results from a divide-by-zero instruction and the blocking fault results from an infinite loop, which are both real bugs that ever existed in drivers [14, 15]. From this figure we can see that *n*Driver can effectively detect the inserted fault and recover the system without stopping the ongoing connection. An exception fault can be detected and recovered in about 2ms, while a blocking fault can be detected and recovered in about 18ms. The latter requires a longer time since a blocking fault is detected using a timeout based approach.

Note that the recovery does not cause the kernel or the applications to timeout for most of the cases since the recovery time is quite short. Process scheduling, disk accesses, and network congestion also cause such short-term delay so that the recovery time should be tolerable for most of the applications.[4]

4.2 Performance Overhead

In this section we use two benchmarks, ttcp [16] and Webstone [17] to measure the overhead of *n*Driver. Specifically, we compare the performance of systems with and without *n*Driver under the condition that no faults happen during the experiment time.

4.2.1 Micro benchmark: ttcp

In this experiment, we set up the ttcp client to send a 5MB file to the ttcp server, and we compare the network throughput (*i.e.*, 5MB/file transfer time) with and without the presence of the *n*Driver framework. The *n*Driver framework is installed on the client side. During the experiment, we control the packet size by limiting the Maximum Transmission Unit (MTU), and we run 1000 times for each packet size. Fig. 6 shows the network throughput degradation caused by *n*Driver. The figure shows that the throughput degradation is only 0.2% under the typical packet size (*i.e.*, 1,500 bytes). With the decreasing of the packet size, the number of packets increases and the network throughput degradation grows. This is because each packet transmission causes a request to the driver, and the overhead of *n*Driver is proportional to the number of driver requests. For each driver request, *n*Driver imposes the following overheads: guard wrapper, action list maintenance, and unfinished request list maintenance. Table 1 shows the breakdown of the

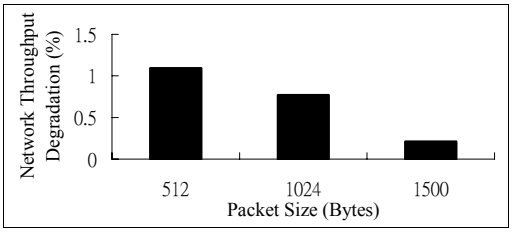


Fig. 6. Network throughput degradation.

[4] The recovery delay may be unacceptable for hard real time applications. However, such applications are not the target of the *n*Driver framework.

Table 1. The results of per-request overhead.

Guard Wrapper (us)	Action List (us)	Unfinished Request List (us)	Total (us)
1.97639	0.34242	0.41712	2.73593

per-request overhead, which is measured using the Pentium Timestamp Counter [18]. The table shows that the per-request overhead is small. Moreover, guard wrapper requires more time than the others. The former performs more jobs such as setting/stopping the fault detection routines and saving/restoring the system context, whereas the latter just involves simple list management.

4.2.2 Macro benchmark: webstone

In this section, we measure the overhead of *n*Driver under a more realistic workload. We install an Apache server (version 2.0.40) on the server machine. The two client machines are used to simulate the web clients. The workload is obtained from the Webstone benchmark, and each round lasts for 10 minutes. We compare the server throughput with and without the presence of the *n*Driver framework. Fig. 7 shows the throughput results. The x-axis represents the number of web clients. Each client tries its best establishing connections and sending requests to the web server during the experiment time. The y-axis represents the number of connections per second the server can handle. Note that the

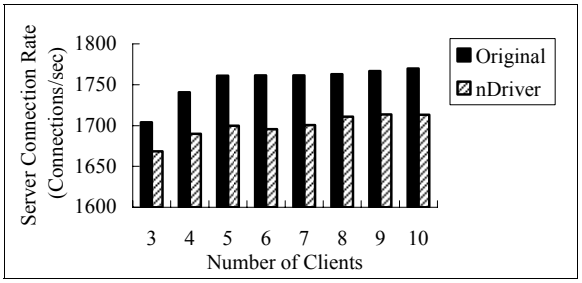


Fig. 7. Throughput of the HTTP server.

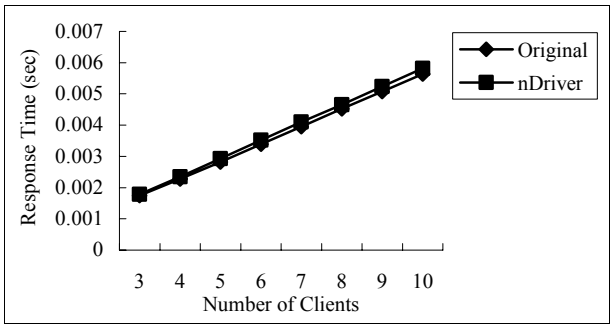


Fig. 8. Response time to access web pages.

minimum value of the y-axis is not zero. From this figure we can see that the performance degradation is between 2.0% ~ 3.5%.

Note that the degradation is more than that indicated in Fig. 6, which is because the Webstone benchmark involves many more small-packet transfers such as three-way hand- shaking and HTTP request transmission. Fig. 8 compares the performance in terms of average response time. The y-axis represents the average response time to access a web page. From this figure we can see that the *n*Driver framework results in an increase of the average response time by 2% to 3%. These results indicate that *n*Driver adds only little overhead to the system.

4.3 Recovery Time

In the final experiment, we manually insert a divide-by-zero fault into the Ethernet driver, and measure the time required to recover the fault. As shown in Fig. 9, the recovery time can be divided into several parts. T*u* is the time that the undo manager spends on invoking the undo routine for each entry in the action list. T*s* is the time that the undo manager spends on replacing the drivers. T*c* is the time that the configuration manager spends on configuring the new driver. Finally, T*r* is the time spent in restoring the system context. Table 2 shows the values of these time periods, which are measured using the Pentium Timestamp Counter. From this table we can see that, the total recovery time is quite small, which indicates that *n*Driver can recover the system from a driver fault very efficiently.

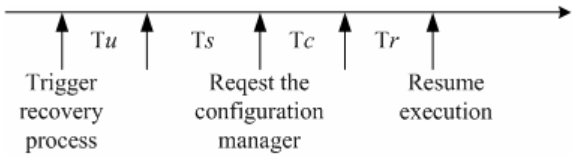


Fig. 9. Different parts of the recovery time.

Table 2. The values of different parts of the recovery time.

T*u* (us)	T*s* (us)	T*c* (us)	T*r* (us)	T*total* (us)
31.95	145.47	290.33	0.0092	467.76

5. DISCUSSIONS

In this section, we first describe the limitations of the current implementation of the *n*Driver framework, which is followed by the description of how to extend the current implementation to support more other drivers.

5.1 Limitations

Although *n*Driver can detect the major kinds of faults (*i.e.*, exception and blocking faults) when they happen on drivers, it can not detect all the driver faults. Specifically, it

can not detect a driver bug if the bug does not cause an exception or cause the system to hang. For example, if a network driver can not actually send out packets due to an incorrect hardware configuration (*e.g.*, wrong Tx buffer address), *n*Driver can not detect it. As another example, *n*Driver also can not detect the condition that a network driver does not update its statistic counters. Therefore, *n*Driver can not detect latent faults in the driver state. Detecting such bugs/faults requires more knowledge about the driver itself. For example, for an Ethernet driver, the Tx statistic counter should be increased by 1 for every packet sent out and the Rx statistic counter should be increased by 1 for every packet comes in. In the future, we will investigate the feasibility of providing an API for driver writers to express such knowledge so that more incorrect behaviors of drivers can be detected.

The current *n*Driver implementation is built on the assumption that drivers do not corrupt the global kernel state. A driver should request kernel resources through predefined kernel interfaces, and should not change the kernel state arbitrarily. This holds for most drivers. However, it can not prevent malicious drivers from corrupting the kernel state since the drivers and the kernel are in the same protection domain. It can not either prevent such drivers from corrupting the application state since a driver has the privilege to write the memory space of an application. To solve this problem, we plan to integrate a technique based on page table switching, which is similar to what is used in Shadow Driver [19], into *n*Driver in the future.

As mentioned in section 3.2, although *n*Driver can prevent the driver requests from being lost, some of the driver state may still be lost (*e.g.*, the packets in the Rx ring buffer of an Ethernet device). The state loss problem will not happen for request-based drivers such as block device drivers and the sending side of network and character device drivers. For the receiving side of character and network device drivers, the problem can be solved by error checking mechanisms performed by the upper software layers. For example, the timeout mechanism of TCP can trigger the retransmission of the lost packets, and the mechanisms such as checksum, CRC, and data-length checking can also be used by the software that performs communication through character devices. It is worth emphasizing that although some Rx data may be lost, *n*Driver is still beneficial for such drivers since it opens an opportunity for application programmers to implement non-stop services on these drivers, providing that they use some error checking mechanisms described above.

5.2 Transient Faults and Software Aging

Although *n*Driver focuses on recovering the system from driver bugs by realizing the design diversity concept in the driver layer, it can also be used to solve the problems of transient faults and driver software aging [20]. Traditionally, these problems are solved by the restart and rejuvenation techniques, which release the resources held by the old instance of the software and then restart a new instance.

The *n*Driver framework can achieve the same effect by replacing the driver with a new instance of the same implementation. The reasons are as follows. First, we undo the changes the old driver instance has made to the global kernel state and release its resources. Second, we discard the old driver state and rebuild the state from scratch during the driver replacement.

5.3 Supporting Driver Replacement on Other Drivers

The current implementation of the *n*Driver framework can survive NIC driver faults. For the following reasons, we believe that all the mechanisms described in section 3 can be adapted to other types of device drivers with a slight extension. First, each driver type provides a standard interface to the kernel, on which we can place the fault detectors. Second, the mechanisms for undoing the kernel state, rebuilding the driver state, and preventing lost requests are all independent of the types of the drivers.

To support dynamic replacement on other drivers, the following tasks should be performed. First, the current implementation should be extended to intercept more callout functions. That is, more interception wrappers should be included into the framework. In the current implementation, we only intercept the callout functions that are used by the NIC driver. Other drivers may invoke other callout functions, which should also be intercepted if we want to include these drivers into the framework. By using the *objdump* utility to list all the un-linked symbols of a driver module, the callout functions of the driver can be identified, and the corresponding interception wrappers can be implemented. Note that, as mentioned in section 2.2, the set of the callout functions is small so that it is achievable to identify and intercept them.

Second, the undo routines corresponding to the callout functions should also be identified so that they can be invoked when the action list entries are removed. This can only be done by tracing the kernel source code to find the counterpart of the callout function. For example, the undo routine of the callout function request_irq() is free_irq(). Fortunately, as mentioned before, the set of the callout functions and the corresponding undo routines is small, and the interception wrappers and the undo routines can be used by all of the drivers once they are included into the *n*Driver framework.

Third, the framework should include the guard wrappers that correspond to the type of the target driver. In the current implementation, *n*Driver can only guard operations of an Ethernet driver. If, for example, we want to incorporate IDE disk drivers into the framework, we should implement more wrappers to guard the operations of a block device driver such as the open, release, and block request functions. Once the guard wrappers are implemented, they can be used by all the block device drivers.

In summary, supporting dynamic replacement on other drivers mainly involves wrapper implementation, which is straightforward. In the future, we will implement more wrappers so as to supporting dynamic replacement on more other drivers.

5.4 Porting *n*Driver to Windows Operating Systems

Many mechanisms of *n*Driver can be ported to Windows kernel without much effort. For example, Windows kernel supports a layered driver model and allows a filter driver to be placed between the kernel and the target driver, and thus we can put the guard wrappers on the filter driver. In addition, we can also setup the fault detectors in Windows. For exception fault, we can replace the kernel exception handlers with our own ones so as to catch exceptions and trigger driver replacement. For blocking fault, we can also replace the original timer interrupt handler with our own one so as to measure the execution time of a driver function. In an x86 machine, these are all done by replacing the entries of the Interrupt Descriptor Table (IDT). Moreover, we can intercept all the

callout functions and redirect them to our interception wrappers by modifying the Import Address Table (IAT) of the driver executable file. Finally, we can find out the undo routines that correspond to the callout functions since they are typically documented.

However, porting the whole *n*Driver framework to Windows is still challenging since Windows does not provide an interface for loading/unloading a driver. The kernel has a total control over the driver loading/unloading activities, and determines when to perform these activities by itself. As a result, we can not replace a driver when a fault is detected. Dynamic driver replacement requires the Windows kernel exporting more APIs.

6. RELATED WORK

Most of the related work can be divided into three categories: driver quality improvement, dynamic replacement of kernel components, and fault tolerance in operating systems. We will describe them in the following three sections, which are followed by the description of the other related work.

6.1 Driver Quality Improvement

Some techniques have been proposed to help driver developers to improve design and reduce bugs in their drivers. Merillon *et al.* [21] proposed a language named Devil to develop device driver code. The developer writes the driver specification in Devil, which is checked by the Devil compiler. After checking, the compiler automatically generates low-level code, which is more error-prone, for driving the device hardware.

To reducing the design bugs, IBM, Microsoft and Intel provided guidelines for designing and implementing drivers for high availability systems [22, 23]. Microsoft also provides a tool called Driver Verifier [24] for verifying the correctness of a driver. It can simulate low resource conditions, verify I/O and DMA operations, detect deadlocks and the like. However, it does not address the issue of how to recover the system from a driver fault.

6.2 Dynamic Replacement of Kernel Components

Drivers are usually implemented as modules [25], and dynamic module loading/unloading can be used as a basic mechanism for hot-swapping module-based device drivers. However, this mechanism alone is not sufficient for fault recovery since it does not consider undoing the kernel state changes, reconfiguring the new driver, and solving the problem of external references.

Design diversity [11, 12] uses multiple independent implementations of the same software to prevent software errors from crashing the whole system. The basic idea is that these functional-equivalent software implementations may not have the same software bugs. Therefore, the system may survive from software bugs by retrying different implementations. Specifically, recovery block uses a set of alternative implementations for the same application to improve the availability. If one alternative fails, another one will be tried. The *n*Driver framework realizes the concept of recovery block at the device driver layer. It is much more challenging to achieve the goal of seamless alternative

swapping at the kernel level. Specifically, we have to address the issues that were not mentioned by the authors such as undoing the global kernel state changes made by the driver, retaining the driver requests, and solving the external reference problem.

Soules *et al*. [13] proposed a mechanism to replace an operating system component during run time. Before a component can be replaced, it has to be in the *quiescent state* (*i.e*., all active use of the component has concluded). When the replacement happens, the old component transfers its state to the new one. Finally, the external references are redirected to the new component. Basically, this mechanism is not appropriate for dealing with faults since the component may not always in the quiescent state when a fault happens. Moreover, the state transferring approach is not suitable for drivers. If the approach is taken, we have to implement a state transferring function for each pair of driver implementations, which requires a large effort.

6.3 Fault Tolerance in Operating Systems

TARGON/32 [26] moved the operating system functionality out of the kernel into server processes, and used the process pair technique to make the server processes fault tolerant. Tandem [27] also used the process pair approach. This approach requires synchronization between the two processes, which increases the implementation complexity and the runtime overhead.

Linux-HA [28] provides a high availability clustering solution for Linux, which contains two major software packages: *Heartbeat* and *Fake.* The former is used to detect whether a host is available or not, while the latter is used to take over the IP address of the failed host. Instead of trying to improve the availability of the operating system, Linux-HA focuses on using another host to take over the job of the failed one.

Shadow Driver [19] allows a driver to be removed and then reloaded when a fault happens on that driver. It uses several techniques that are similar to those used in *n*Dirver, such as exception fault detection, callout function interception, and configuration logging. However, *n*Driver differs from Shadow Driver in three aspects. First, Shadow Driver aims at resolving transient faults instead of driver bugs. It assumes that the same driver implementation will be reloaded, and some techniques it employs are based on that assumption. For example, it reuses the driver code directly instead of reloading it. As a result, if the driver has a bug, the fault may happen again soon after the system is recovered. In contrast, *n*Driver can recover the system not only from transient faults but also from driver bugs since it realizes the design diversity concept at the driver layer. Once a fault happens, another driver implementation can be tried, further improving the system availability. Second, Shadow Driver can not detect blocking faults, which are one of the two major kinds of faults observed in the Linux driver.[5] In contrast, *n*Dirver uses a timeout based approach to detect these faults. Third, Shadow Driver requires a larger runtime overhead since it is based on a protection domain architecture, which uses call-by-value-result semantic for data communication between domains (*e.g*., kernel and driver). As a result, an extra copy is needed when a packet is sent from the driver to the kernel, and vice versa. Such extra copy causes obvious performance degradation. In contrast, *n*Driver does not have such overhead.

[5] Shadow Driver can detect livelocks by checking if the driver-kernel domain crossing happens too frequently. However, it can not detect blocking faults caused by infinite for/while/goto loops or deadlocks.

User level driver [29] allows a driver to be implemented in user mode and thus a driver fault can not crash the kernel. This approach improves availability of the kernel but does not improve availability of the whole system. For example, when a user mode driver crashes, the service applications based on that driver still becomes unavailable. A fault detection and recovery approach is still needed for user mode drivers, although running a driver in user mode can make the approach easier to implement. Moreover, user level driver results in more context switches and user-kernel domain-crossing overheads, and thus it is not suitable for performance critical drivers.

6.4 Others

Autonomic Computing [30] proposed self-healing techniques that can automatically detect, diagnose, and repair software and hardware problems. Recovery-Oriented Computing [31] also proposed new techniques to deal with hardware faults, software bugs, and operator errors. The basic idea of these two projects is similar to that of *n*Driver. That is, systems should deal with faults, instead of preventing them.

Checkpointing [32-35] is a common technique for system recovery. It saves the system state periodically or before entering critical regions. Once the system fails, it can be recovered by restoring the last checkpointed state. The major problem of this technique is that it can neither resolve the software aging problem nor make the system survive faults caused by driver bugs since it restores the aged state and re-executes the same code after recovery. Moreover, many checkpointing implementations incur overheads due to the storing of vast amounts of state.

Lakamraju [36] introduced a low-overhead fault tolerance technique to recover a Myrinet NIC from network processor hangs. When the network processor hangs, it resets the NIC and rebuilds the hardware state from scratch to avoid duplicate and lost messages. The limitation of this work is that it focuses only on hardware failures instead of software errors. The former is easier to handle since it does not consider complex software state maintenance problems such as undoing the kernel state changes, reconfiguring the new driver, and solving the problem of external references.

7. CONCLUSIONS

Device driver is the most unreliable part of an operating system. In this paper, we propose the *n*Driver framework, which uses multiple implementations of a device driver to survive from driver faults. This framework can detect two major types of driver faults, exception and blocking faults. With the help of *n*Driver, driver faults will not always result in kernel panics or system hangs. Instead, if a fault is detected, *n*Driver substitutes another driver implementation for the faulty one to enable the system to continue working. In order to achieve the goal of seamless driver replacement, *n*Driver undoes the kernel state changes made by the faulty driver, retains the unfinished driver requests, and solving the external reference problem. In addition, *n*Driver blocks the driver-removing and installation events so that the other kernel subsystems are not aware of the driver replacement.

The major contribution of this work is that *n*Driver realizes the concept of recovery

blocks at the device driver layer. It achieves the goal of seamless driver replacement. Most importantly, it improves operating system availability without modifying the driver codes.

We implement *n*Driver as a kernel module in Linux. Currently, it can enable the system to recover from faults in Ethernet device drivers. However, the mechanisms can be adapted to other module-based device drivers with a slight extension. According to the performance evaluation, the overhead of *n*Driver is no more than 3.5% in the case of a Gigabit Ethernet driver, and the recovery time is quite small. This indicates that *n*Driver is an efficient mechanism to increase the availability of an operating system.

CODE AVAILABILITY

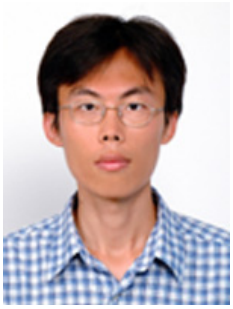
Information and source code of the current implementation of *n*Driver are available from http://www.os.nctu.edu.tw/research/nDriver/index.htm.

REFERENCES

1. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, ISBN: 1558601902, 1993.
2. J. Gray and D. P. Siewiorek, "High-availability computer systems," *Computer*, Vol. 24, 1991, pp. 39-48.
3. I. Lee and R. K. Iyer, "Software dependability in the Tandem GUARDIAN system," *IEEE Transactions on Software Engineering*, Vol. 21, 1995, pp. 455-467.
4. D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do Internet services fail, and what can be done about it?" in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, http://www.usenix.org/publications/library/proceedings/usits03/tech/oppenheimer.html, 2003.
5. T. Davis, "Linux channel bonding," http://www.sourceforge.net/projects/bonding/usr/src/linux/Documentation/networking/bonding.txt, 2003.
6. Intel Corporation, "Intel networking technology – load balancing," http://www.intel.com/network/connectivity/resources/technologies/load_balancing.htm, 2003.
7. J. Jann, L. M. Browning, and R. S. Burugula, "Dynamic reconfiguration: basic building blocks for autonomic computing on IBM pSeries servers," *IBM Systems Journal*, Vol. 42, 2003, pp. 29-37.
8. D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz, "Introduction to redundant arrays of inexpensive disks (RAID)," *Digest of Papers for 34th IEEE Computer Society International Conference* (*COMPCON Spring*), 1989, pp. 112-117.
9. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating system errors," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 73-88.
10. J. Corbet, "2003 kernel summit: high availability," http://lwn.net/Articles/40620/, 2003.
11. C. Inacio, "Software fault tolerance," http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/, 1998.

12. M. Lyu, (ed.), *Software Fault Tolerance*, John Wiley & Sons, Chichester, 1995.
13. C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. S. Rosenburg, and J. Xenidis, "System support for online reconfiguration," in *Proceedings of the USENIX Annual Technical Conference*, 2003, pp. 141-154.
14. R. Lievin, "Patches to tipar character driver," http://www.ussg.iu.edu/hypermail/linux/kernel/0404.2/0018.html, 2004.
15. D. Benham, "Bug number 89192: won't boot Adaptec's AIC-7899 SCSI controller," *Debian Bug Tracking System*, http://groups.google.com.tw/group/debian.bugs.closed/browse_thread/thread/86dc29023e094447/2aeaef46165d6132?lnk=st&q=Adaptec%27s+AIC-7899+SCSI+controller+infinite+loop&rnum=2&hl=zh-TW#2aeaef46165d6132, 2001.
16. Netcordia Inc., "The test TCP tool," http://www.netcordia.com/tools/tools/TTCP/ttcp.html, 2003.
17. Mindcraft Inc., "WebStone: the benchmark for web servers," http://www.mindcraft.com/benchmarks/webstone/, 2004.
18. A. Rubini, "Making system calls from kernel space," *Linux Magazine*, http://www.linux-mag.com/2000-11/gear_01.html, 2000.
19. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, http://www.cs.washington.edu/homes/mikesw/nooks/recovering-drivers.pdf, 2004.
20. V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, "Proactive management of software aging," *IBM Journal of Research and Development*, Vol. 45, 2001, pp. 311-332.
21. F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller, "Devil: an IDL for hardware programming," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, http://portal.acm.org/citation.cfm?id=1251229.1251231&coll=ACM&dl=ACM&CFID=15151515&CFTOKEN=6184618, 2000.
22. IBM Corporation and Intel Corporation, "Device driver hardening," http://hardeneddrivers.sourceforge.net/, 2004.
23. Microsoft Corporation, "Writing drivers for reliability, robustness and fault tolerant systems," *Microsoft Windows Hardware Engineering Conference*, http://www.microsoft.com/whdc/system/platform/server/FTdrv.mspx, 2002.
24. L. Columbus, "How Windows XP's device driver verifier works," http://www.informit.com/articles/article.asp?p=22085&redir=1, 2001.
25. B. Henderson, "Linux loadable kernel module HOWTO," http://www.tldp.org/HOWTO/Module-HOWTO/, 2004.
26. A. Borg, W. Blau, W. Craetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Transactions on Computer Systems*, Vol. 7, 1989, pp. 1-24.
27. J. Gray, "Why do computers stop and what can be done about it?" in *Proceedings of the Symposium on Reliability in Distributed Software Database Systems*, 1986, pp. 3-12.
28. A. Robertson, "High-availability Linux project," http://linux-ha.org/, 2004.
29. P. Chubb, "User level device drivers," http://www.disy.cse.unsw.edu.au/Software/ULDD/, 2004.

30. J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer Journal*, Vol. 36, 2003, pp. 41-50.
31. D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery-oriented computing (ROC): motivation, definition, techniques, and case studies," Technical Report No. UCB//CSD-02-1175, Computer Science Department, UC Berkeley, 2002.
32. D. E. Lowell, S. Chandra, and P. M. Chen, "Exploring failure transparency and the limits of generic recovery," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000, pp. 289-304.
33. J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: transparent checkpointing under UNIX," in *Proceedings of the Usenix Winter Technical Conference*, 1995, pp. 213-223.
34. Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. Kintala, "Checkpointing and its applications," in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 1995, pp. 22-32.
35. A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *IEEE Transactions on Computers*, Vol. 46, 1997, pp. 976-985.
36. V. Lakamraju, I. Koren, and C. M. Krishna, "Low overhead fault tolerant networking in Myrinet," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2003, pp. 193-202.



Da-Wei Chang (張大緯) received his B.S., M.S., and Ph.D. degrees in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan, R.O.C., in 1995, 1997 and 2001 respectively. He has been a postdoctoral researcher in National Chiao Tung University in 2002-2005, and an assistant professor in Electrical Engineering at National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C. in 2006. He is currently an assistant professor in Computer Science and Information Engineering at National Cheng Kung University, Tainan, Taiwan, R.O.C. His research interests include operating systems, fault-tolerant systems, and embedded systems.

Zhi-Yuan Huang (黃致遠) received his B.S. degree in Computer Science and Information Engineering from National Central University, Taiwan, R.O.C., in 2004. He is currently a master student in Computer Science at National Chiao Tung University, Taiwan, R.O.C. His research interests include operating systems, driver design, and embedded systems.

Ruei-Chuan Chang (張瑞川) received his B.S. degree (1979), his M.S. degree (1981), and his Ph.D. degree (1984), all in Computer Engineering from National Chiao Tung University. He is currently a professor in Computer Science at National Chiao Tung University, Taiwan, R.O.C. He is also an Associate Research Fellow at the Institute of Information Science, Academia Sinica, Taipei. His research interests include operating systems, wireless communication technologies, and embedded systems.