國 立 交 通 大 學

資訊工程學系

碩 士 論 文

動態置換驅動程式以增進作業系統之可用性

_n_Driver: Online Driver Swapping for Increasing Operating System Availability

研 究 生：江英杰

指導教授：張瑞川　教授

林正中　副教授

中 華 民 國　九 十 三 年 六 月

動態置換驅動程式以增進作業系統之可用性

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

研究生：江英杰　　Student：Ying-Jay Chiang

指導教授：張瑞川教授　　Advisor：Prof. Ruei-Chuan Chang

指導教授：林正中副教授　　Advisor: Prof. Cheng-Chung Lin

國立交通大學

資訊工程系

碩士論文

A Thesis
Submitted to Institute of
Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
In

Computer Science and Information Engineering
June 2004
Hsinchu, Taiwan, Republic of China

中華民國九十三年六月

動態置換驅動程式以增進作業系統之可用性

研究生：江英杰　　指導教授：張瑞川教授

指導教授：林正中副教授

國立交通大學資訊工程所

論　文　摘　要

近年來，作業系統的可靠度逐漸受到重視，因為一些需要高可用性的服務都必須依賴作業系統所提供的功能。然而，驅動程式設計上的缺陷卻容易破壞作業系統的穩定性。研究指出，在作業系統的原始碼中，驅動程式的錯誤比例是其他非驅動部分的 3 倍至 7 倍。因此，我們提出了一個架構，驅動程式設計團隊可以提供多份不同實做方式的驅動程式，透過此架構來避免驅動程式的錯誤設計讓整個系統無法運作。如果驅動程式因設計缺陷而發生錯誤後，我們的架構就會將發生錯誤的驅動程式移除，再使用另外一份的驅動程式。除此之外，我們的架構必須修補系統的狀態以及恢復遺失的系統要求。

我們將這個架構實做在目前相當流行的 Linux 作業系統來驗證我們的方法。根據實驗的結果，我們架構對效能所造成的額外負擔不超過百分之五，且整個修復的時間非常短。証明我們的架構是個可以有效增加作業系統可用性的方法。

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

Student: Ying-Jay Chiang Advisor: Prof. Ruei-Chuan Chang

Advisor: Prof. Cheng-Chung Lin

Department of Computer Science and Information Engineering

National Chiao Tung University

Abstract

The reliability of an operating system is important because all applications must depend on the functionality it provides. However, design defects of device drivers violate the reliability of an operating system. It is showed in [18] that the error rates of device drivers can be three to seven times higher than the rest of the kernel. In this paper, we propose a framework named *n*Driver via which a driver administrator can use multiple implementations to increase the driver robustness. In case there is a fault happening in a driver, *n*Driver can dynamically replace the faulty implementation rather than let it crash the system. In addition, *n*Driver must fix the system state and recover the lost system requests.

We implement *n*Driver in the Linux operating system. According to the evaluation, the overhead of *n*Driver is no more than 5% and the time it takes to recover is very small. This indicates nDriver is a feasible mechanism to increase the availability of operating systems.

Keywords: Recovery, Device Driver, Design Diversity, Reconfiguration

Acknowledgement

I deeply appreciate my respected advisor, Prof. Ruei-Chang, for teaching me in doing research. A special thank to Da-Wei Chang for giving me many advices on revising this thesis. And I also want to thank all the members of the Computer System Lab. for their kindness to help me solve my problems.

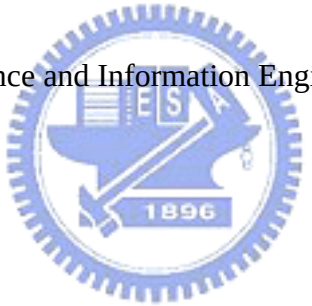
Besides, I am grateful to my family for their encouragement and endless love. Finally, I want to thank all my friends for all the joyous things that inspire my life.

Ying-Jay Chiang

Department of Computer Science and Information Engineering

Nation Chiao-Tung University

2004/6



Index

List of Figures

List of Tables

1. Introduction

With the high reliance of people on computer systems, the availability of a system is becoming more and more important. For a growing number of systems, high availability is no longer optional but mandatory. According to the previous research, [6] shows that 60-90% of current computer faults are software errors instead of hardware faults and [18] shows that hardware failure doesn't take a large part of service failure. Moreover, hardware faults can generally be masked through component redundancy [5][10][11][19]. Therefore, software plays a critical role in system availability.

Since most of the software relies on the underlying operating system, a reliability of a operating system is a key factor to a highly availability computer system. Unfortunately, due to the high complexity of an operating system, it's nearly impossible to make it error-free.

The most error-prone part of an operating system is device drivers. It is showed that the error rates of device drivers can be three to seven times higher than the rest of the kernel [4][13]. The reason is that most of the 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 happening in a driver is a kernel fault, and it results in a kernel panic or a system hang in many operating systems including Linux. This causes the services running on it become unavailable.

However, a faulty driver usually doesn't pollute the other subsystems in the kernel. Therefore, it is possible to recover the system from the driver faults, and hence allow the services running on it remain available. In this paper, we propose a mechanism to survive from the software faults in the drivers. According to the

previous research [4], blocking and exception faults are responsible for the largest portions of faults happening in the kernel. Specifically, blocking faults account for 28.5% of the faults observed in the Linux kernel (version 2.4.1), while exception faults account for 41%. The former lead to kernel hangs, while the latter cause kernel panics. Therefore, we concentrate on detecting and recovering from these faults.

In this thesis, we propose a framework, named *n*Driver, for surviving from these software faults. Based on the design diversity concept [8][21], we use multiple driver implementations for a device. 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 may exist patches for a driver implementation. By downloading the patches and applying them to the driver implementation, another implementation is produced. Second, there may be multiple driver versions for the same device. Because the newer release may be less stable, we can consider the older release as the backup implementation. Third, there may exist a generic but regressive driver for the device. For example, the ne2000 NIC device driver can be used to drive many NICs of different vendors. It is worth to note that, the framework can improve operating system availability even when there is only one driver implementation for each device. By swapping the driver, when a fault occurs in it, with a refresh instance of the same implementation, the problem of transient faults and driver aging [3] can be solved.

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

- Non-stop services: The services or applications running on top of the system should keep on 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, the removing of a driver causes the loss of its internal data, including the requests issued to it. However, to achieve the goal of seamless driver swapping, all the uncompleted requests should be kept and then be 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 requests 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 be crashed due to these dangling references.

In this thesis, we describe the design and implementation of the *n*Driver framework. Specifically, we present how the *n*Driver framework satisfies the above requirements. The framework is implemented in the Linux kernel. Currently, it can survive from faults happening in NIC (Network Interface Card) and NBD (network block device) drivers. However, we consider the mechanisms can be adapted to other module-based device drivers with a little modification. According to the experimental results, nDriver can currently recover a NIC driver fault with only **??**% performance loss under a popular web benchmark, Webstone. Therefore, it is feasible to be applied.

The rest of this thesis is organized as follows. In Chapter 2 we describe our design issues and the flow of the device driver recovery, which is followed by the description of the implementation details in Chapter 3. Chapter 4 presents the experimental results. Chapter 5 shows the related work. And finally, we conclude in Chapter 6.

2. Design

In this section, we will elaborate on what we do to survive from driver faults.

When a fault occurring in the driver is detected, the recovery mechanism will be triggered. Figure 1 shows the overview of the recovery process. Briefly speaking, we remove the faulty driver, undo the changes caused by it, insert the new driver, reconfigure it, and retry the original function in the new driver.

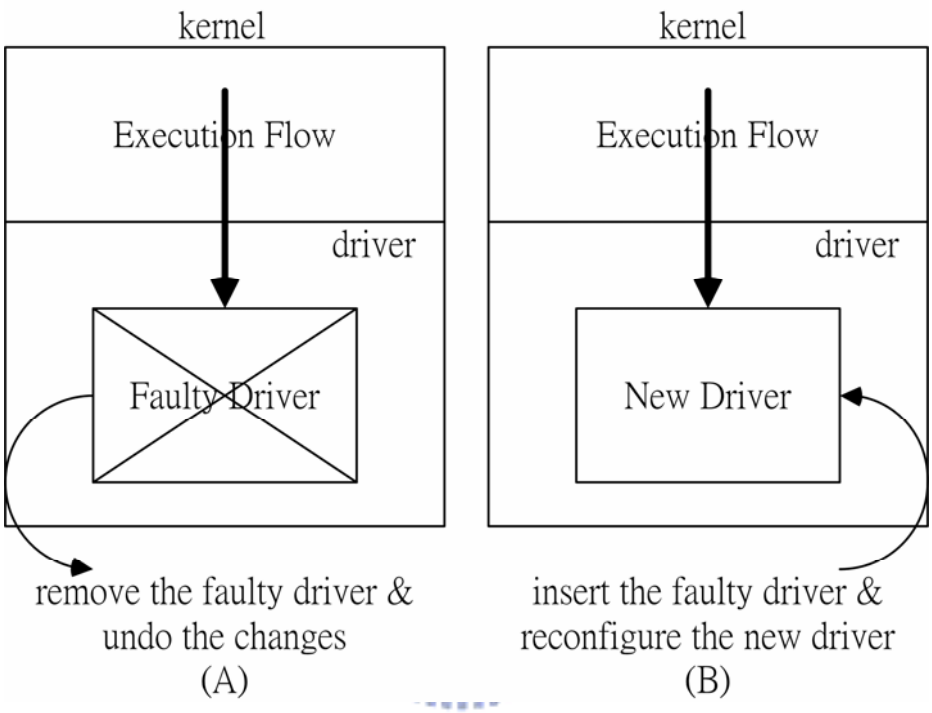


Figure 1. Overview of the Recovery Process

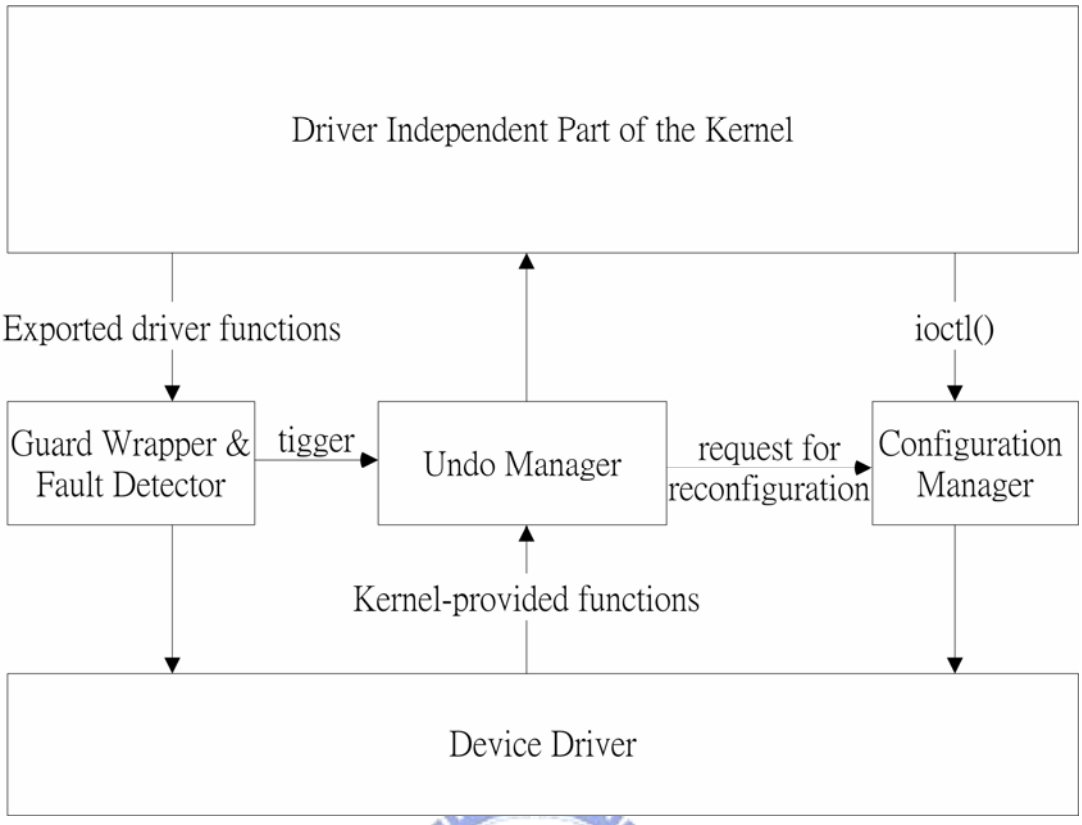


Figure 2. Architecture Overview

Figure 2 illustrates the components of the *n*Driver. If a fault occurs, it will be detected by the fault detector. Since the faulty driver may have changed the system state, we should remove the faulty driver and undo the changes. This is performed by the undo manager which records all the kernel functions invoked by the driver and undoes them when the driver is removed. In addition, it is responsible for inserting the new driver and asking the configuration manager to reconfigure it. After the reconfiguration, all external references to the removed driver must be redirected to the new driver to avoid the problem of dangling references.

In the following sections, we will describe the design of the *n*Driver framework. First, we will present the fault detection approaches, 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 details of the recovery process.

2.1 Fault Detection

The fault detector is responsible for detecting exception and blocking faults. An exception fault occurs due to the reasons such as accessing the NULL page (i.e., the first page of the physical address space), dividing an operand by zero, or executing an invalid opcode. To detect such faults, we replace the kernel exception handlers (such as page-fault and the divide-by-zero handlers) with our own ones. Therefore, the raising of a CPU exception will trigger our exception handler, which will then invoke the undo manager to recover the fault.

Besides exception faults, a faulty driver may cause system hangs (i.e., blocking fault), which make the system become responseless. Blocking faults usually result from careless driver design such as entering an infinite loop or trying to get the 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 setup 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 happen every 10 ms. Although the time-out based approach is straightforward, two issues must be addressed to make it 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’t have a fixed time-out value for all the driver functions. Instead, the time-out value of a driver function should be set to its average execution time plus a guard time. Note that the time-out values are not required to be highly accurate. The 10-ms granularity is accurate enough for detecting blocking faults.

Another issue is how to prevent the software timer approach becoming useless if the driver function disables interrupts in their code. 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() 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.

2.2 State Maintenance

We divide the system state that the driver may modify during its execution into *driver state*, *kernel state*, and *driver requests*. The driver state is the local state of the device driver. The kernel state represents the global kernel state that may be changed by the driver. And the driver requests stands for the requests that are currently processed by the driver and the corresponding device. Because a fault may happen anytime during the execution of the driver code, we must keep the state correct and consistent after recovery. During the recovery period, we undo the changes the driver made to the kernel state. For the driver state, we decide to discard it and rebuild it from scratch. And, for the driver requests, we record them so that they can be re-issued to the new driver implementation after the recovery.

Generally, a driver changes the kernel state only through a few functions provided by the kernel. Such functions may request kernel-managed resources, register a new driver, or exchange information with the kernel. For example, the driver may request IRQs and I/O regions to the kernel. In order to undo the changes, we intercept the kernel functions called from the driver (i.e., callout functions), and

record them in an action list. Each callout function in the list has a corresponding undo routine, which will be invoked during the recovery process, for undoing the changes caused by the function.

It is worth noting that a device driver may invoke only a small subset of kernel-provided functions. This is because the main purpose of a device driver is just to drive the device. For example, a driver usually doesn't perform IPC operations, which are difficult to rollback[1]. Thus, we focus on the set of functions which may be invoked by the driver, and implement their undo functions manually.

As we 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 emerged in the driver code. Second, different driver implementations may use different data structures and thus the old driver state cannot directly be used by the new driver implementation. Therefore, 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 impossible. Moreover, it’s impractical to implement a state transfer function for each pair of driver implementation.

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 request is finished, the request 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 again.

[1] 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.

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. Figure 3 illustrates an example. The structure net_device is used to represent an NIC device driver in Linux. During recovery process, for instance, the faulty driver *Faulty* is removed and the new driver *New* is inserted and initialized. All external references to *Faulty* become dangling pointers.

Soules et al. [23] proposed two approaches (i.e., backward reference and indirection) as shown in Figure 3(a) and 3(b) to solve this problem. In brief, the backward reference approach keeps track of all external references to Faulty, and then updates all of them to point to *New*. 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 Figure 3(b), lets all the external references point to a single indirection pointer. If the target is changed due to the driver swapping, 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 deferencing to access the target.

In *n*Driver, we take another approach to avoid modifying the existing operating system code. Figure 3(c) shows the approach. We add a placeholder for containing the target data. The placeholder is of the same type with the target data, and all the external references point to the placeholder. In the figure, the placeholder is initialized by copying the content of *Faulty* to it. During the recovery process, *Faulty* is removed and the placeholder is updated by copying the content of *New* to it. In this way, neither the maintaining of the backward references nor the modification to the data

type of the external references is needed.

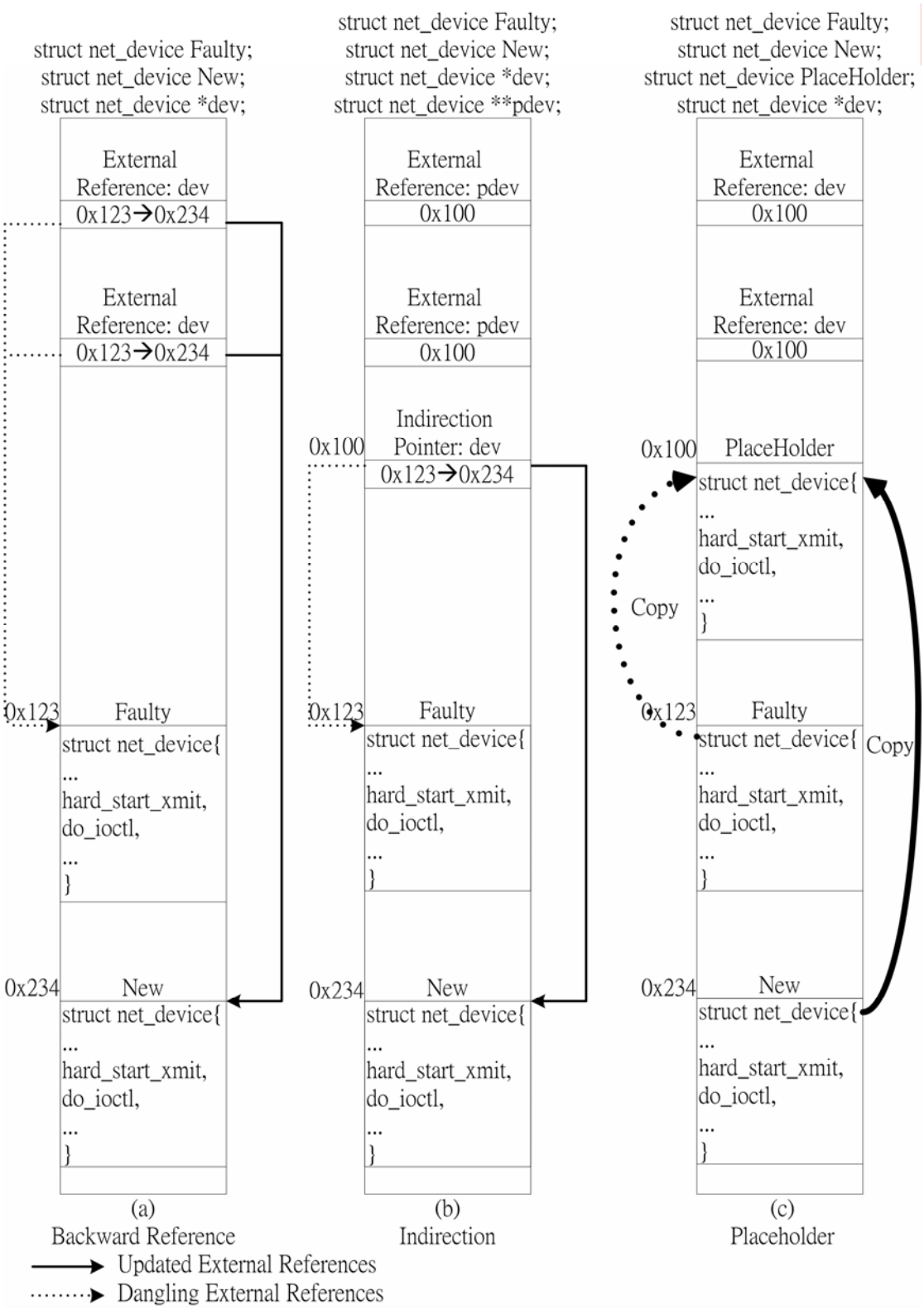


Figure 3. External Reference Redirection

2.4 Detailed Process of Recovery

Before executing a driver function, we initialize the fault detector as well as save the current system context.

During the execution of the function, our recovery mechanism will be triggered if an exception fault or a blocking fault is detected. Figure 4 shows the detailed recovery process. First, we undo the changes the driver has made to the global kernel state. Specifically, we call the undo routine of each entry in the action list to undo the changes. Second, we remove the code and the local state of the faulty driver. Third, we insert the new driver into the kernel and reset the hardware. Fourth, the previously-issued configuration operations are issued again to the new driver in order to rebuild the driver state. This is achievable since all the configuration operations previously issued to the driver were intercepted and logged by the configuration manager. Fifth, we update the external references to point to the new driver by copying the content of the new driver state to the corresponding placeholder. Finally, we restore the system context and retry the originally-failed function in the new driver.

It is worth to note that the new driver may correspond to the same implementation with the old one. In this case, the new driver is just a fresh instance of that implementation. This kind of driver swapping can solve the problem of transient errors and driver aging [3]. The latter problem can be solved because we discard and rebuild the driver state from scratch. If there are multiple driver implementations for the device, the system can choose another implementation if one fails. This allows the system to survive from not only the above two kinds of faults but also the faults caused by driver bugs.

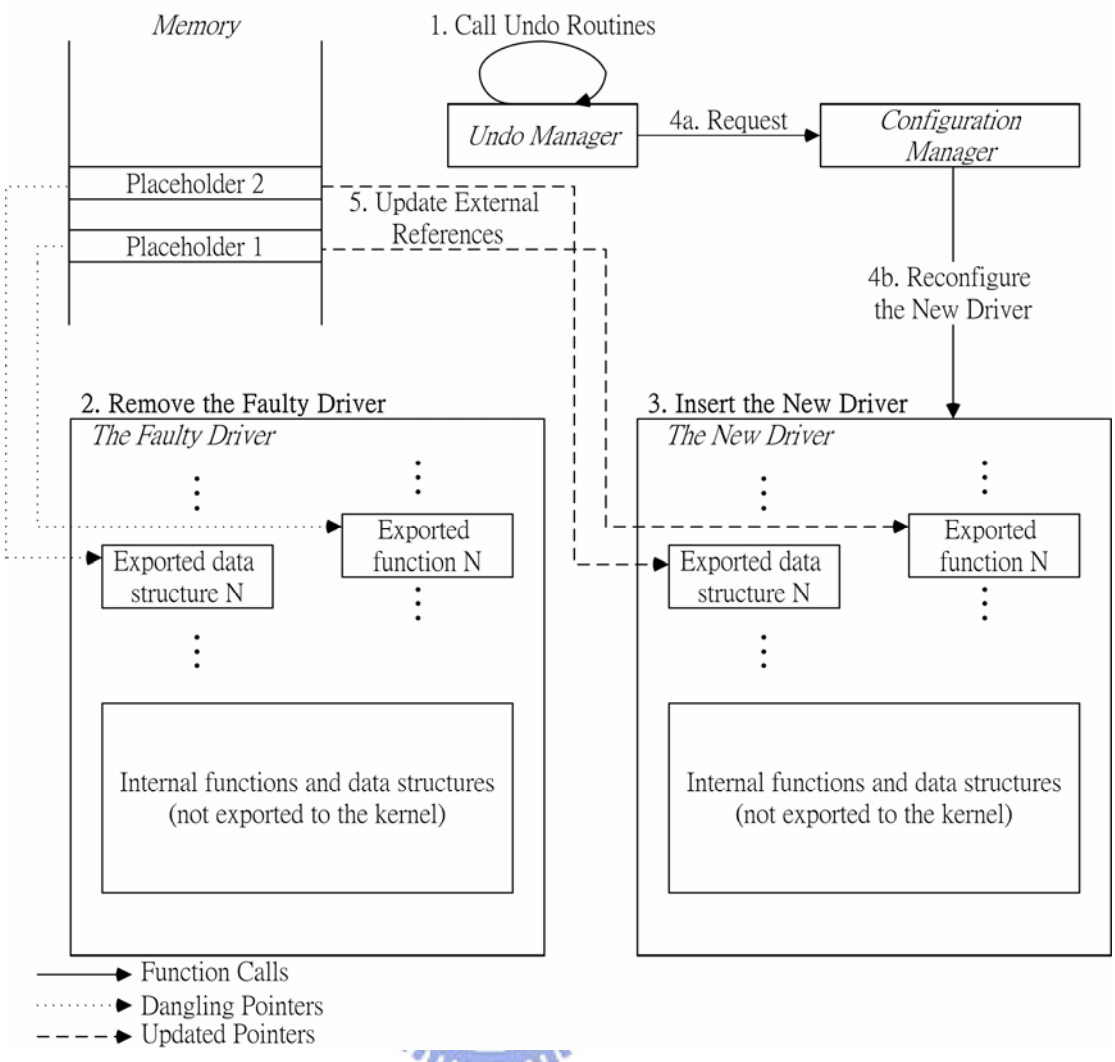


Figure 4. Detailed Process of Recovery

3. Implementation

The *n*Driver framework is implemented as a kernel module in Linux. Based on the framework, we can currently recover the Ethernet driver and network block device driver faults. In the following, we will describe the implementation details of the *n*Driver.

3.1 Fault Detection

3.1.1 Guard Wrapper

Since we regard driver functions as unreliable, we put a guard wrapper on each

function exported by the driver to prevent a driver fault from crashing or halting the kernel. The wrapper takes the following actions.

First, it sets up the fault detection routines. For exception faults, it substitutes our exception handlers with the original exception handlers, such as the divided-by-zero handler or the page-fault handler. For blocking faults, it initiates a software timer to measure time it takes to execute the wrapped function.

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. The recovery process will restore the system context, remove the faulty driver, insert the new driver, and retry the function in the new implementation. Similarly, if the wrapped function doesn't 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 with the original ones and stops the timer.

3.1.2 Software Timer

As we mentioned in Section 2.1, we use an interrupt-based timer to measure the time it takes to execute a driver function. If the timer expires, the function is regarded as failure and the recovery process is triggered.

Before we start to execute the driver function, we initialize a counter to its time-out value. Each time the timer interrupt raises, our software timer decreases the counter by 1. If the counter reaches 0, our software timer will trigger the time-out handler.

Since a driver function may be preempted by other interrupt handlers except for the timer interrupt handler, we should stop counting during the time the function is preempted. However, we don't integrate this technique into *n*Driver. This is because,

according to the experimental result, the time used by ISRs and bottom halves are quite small compared to the 10-ms timer interrupt interval. Therefore, they don't have a visible 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 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 to record the invocations of the callout functions. Figure 5 shows an example of the action list. When an interception wrapper is invoked, we allocate an entry to record the function identifier, the values of the arguments, and the return value. Then, we add this entry to the action list. Keeping the arguments and the return value is necessary to undo 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(), the undo routine of the request_irq(), to release the allocated IRQ resources. Once the driver invokes an 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.

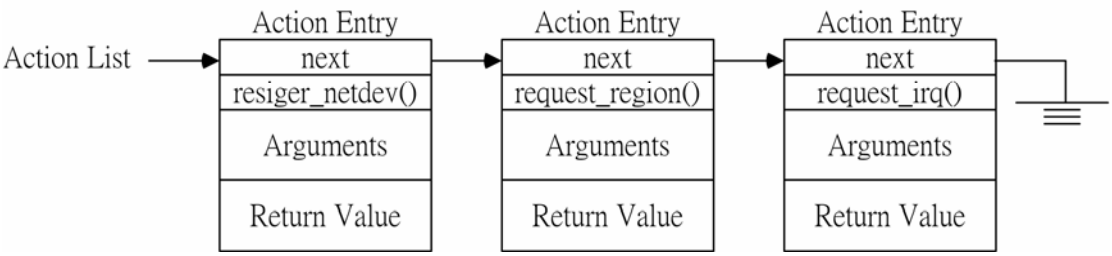


Figure 5. the Structure of the Action List

3.2.1 Preventing Lost of Driver Requests

In this subsection, we describe how the *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 driver for Accton EN1207F Series PCI Fast Ethernet Adapter as an example of illustration.

Figure 6 illustrates how the driver sends and receives packets. For the sending side, the kernel dequeues a packet from the send queue (i.e., qdisc in Figure 6) of the driver and hands the packet to the driver. The job of the driver is to insert the packet into its Tx ring buffer and driving 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. 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 swapping, the packets in the list represent the lost packets and can be re-issued again

to the new driver.

However, this approach cannot be used on the receiving side. The packets in the Rx ring buffer cannot be recovered after the driver swapping. This is because packets are inserted into the Rx ring buffer via the DMA hardware. Without specific hardware support, it is impossible to copy a packet before it enters into the Rx ring buffer. Fortunately, packet lost is not a rarely-happened problem. It can also result from network congestion or the RX ring buffer overrun. (And, it can be resolved by reliable network protocols such as TCP.) Therefore, we consider that losing a small number of Rx packets due to the NIC driver failure is acceptable.

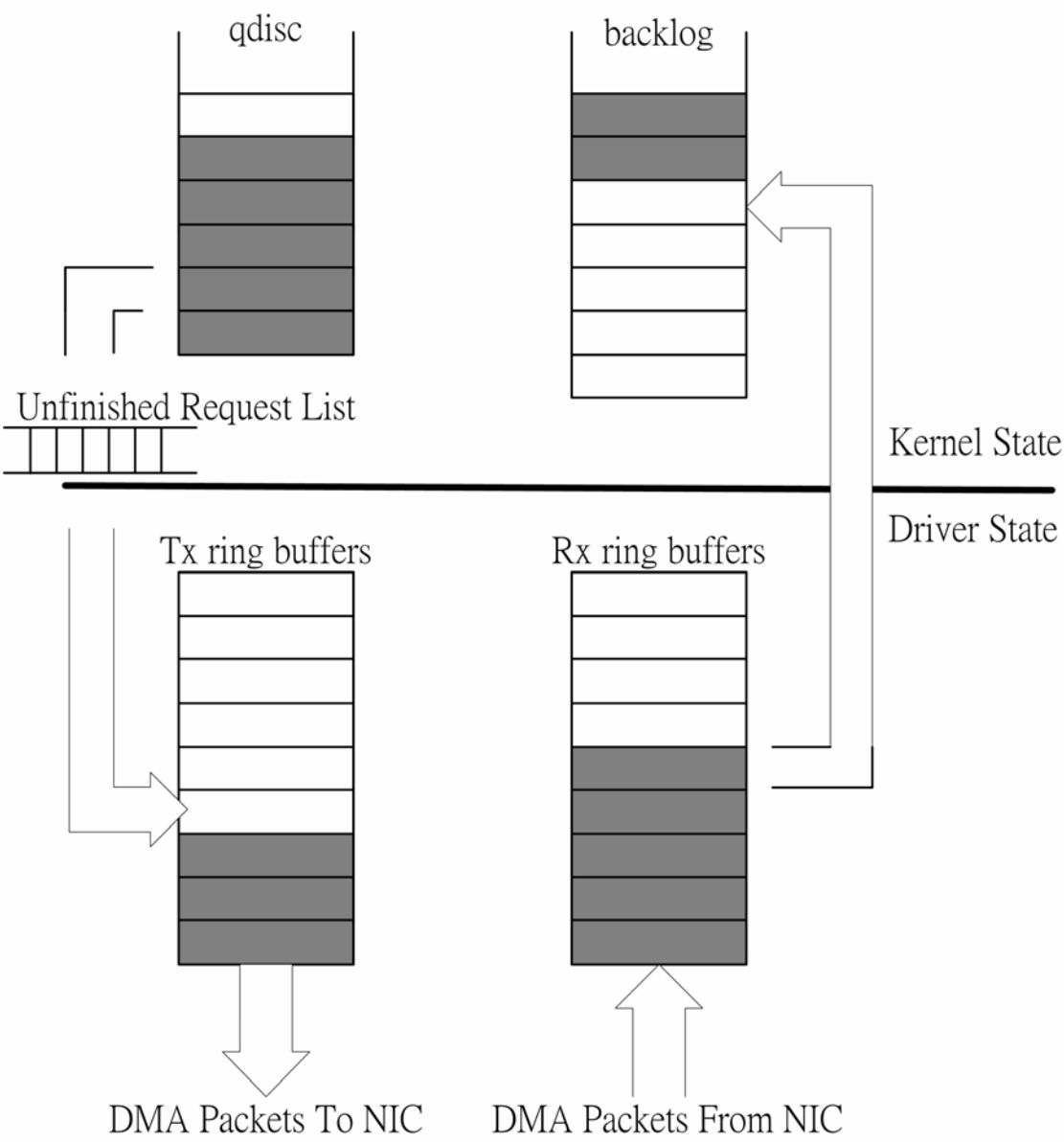


Figure 6. The Data Flow of NIC Device Driver

The Rx data lost problem will not happen for block device drivers. 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 for maintaining the unfinished request list. For character devices, the Rx data may be lost because we can't locate the buffer without digging into the driver code. Although some Rx data may be lost, nDriver can guarantee non-stop services instead of letting the fault driver crash the system.

3.3 Recovery Flow Implementation

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

Before executing the wrapped driver function, the guard wrapper sets up the fault detection routines and saves the system context.

As we mentioned before, 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. It calls the sys_delete_module() function to remove the code and data of the faulty driver module. In addition, it invokes the undo routine for each entry in the action list to undo the kernel state changes and release the resources held by the faulty driver. Although each driver provides functions (i.e. cleanup() and close()) for releasing its resources, we consider that it is unsafe to execute these functions after a fault has happened in that driver.

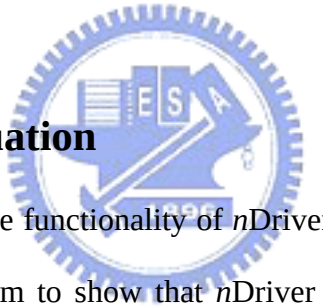
It's worth to note that undoing the kernel changes may result in some events to be sent to other kernel subsystems in order to notify that the status of the driver has been changed. After the subsystems receive the events, they will take some corresponding actions. For examples, if we remove a network device driver, any routing table entries depending on it will be deleted. This situation should be prevented since we don't want the rest of the kernel be aware of the driver swapping. Therefore, we have to block the events.

The second step of the undo manager is to install the code and data of the new driver module into the kernel and call the init() function of that module. The init() function usually resets the hardware as well as causes some initialization events to be sent to other subsystems. Similar to what we have described above, we also have to block these events.

After the new driver module is installed, the configuration manager is asked to

reconfigure the driver. Since it has logged the configuration operations performed on the faulty driver, the reconfiguration can simply be done by performing these operations again to the new driver. After the reconfiguration, the undo manager redirects the external references to the new driver module, and restores the system context. Finally, the undo manager retries the previously-failed function.

We believe that all the mechanisms described above can be adapted to other types of device drivers (e.g., block device drivers) with little modification. The reasons are as follows. First, each driver type provides a standard interface to the kernel, on which we can place the fault detectors. Second, the mechanism for undoing the kernel state, rebuilding the driver state, and preventing the request lost are all independent to the types of the drivers.



4. Performance Evaluation

In this chapter, we test the functionality of *n*Driver and measure its performance overhead. The experiments aim to show that *n*Driver can make the system survive from driver faults with a little performance degradation. The testbed consists of one server and two clients. All the machines are connected to a 1 Gigabit Ethernet switch. Each machine is equipped with Pentium 4 2.0GHz CPU, 256MB DDRAM. The operating system is Linux (kernel version 2.4.20-8).

4.1 Functionality

In this experiment, we initialize a TCP connection from a client to get a file in the server. During the transfer, we use the tcpdump utility to intercept all packets in order to record their ACK sequence number. The ACK sequence numbers means the number of bytes which have been received by the client. We repeat the above

procedure but insert a fault in the NIC device driver. Figure 7 shows the results. The connection named No Fault means there is no fault happening during the connection. In the two other connections, there are an exception fault and a blocking fault individually. We can see that *n*Driver can effectively detect the inserted fault and recover from it without stopping the ongoing connection. After recovery, the slope of the survived connections remains almost the same. It means that the recovery process doesn't incur much degradation to the transfer speed.

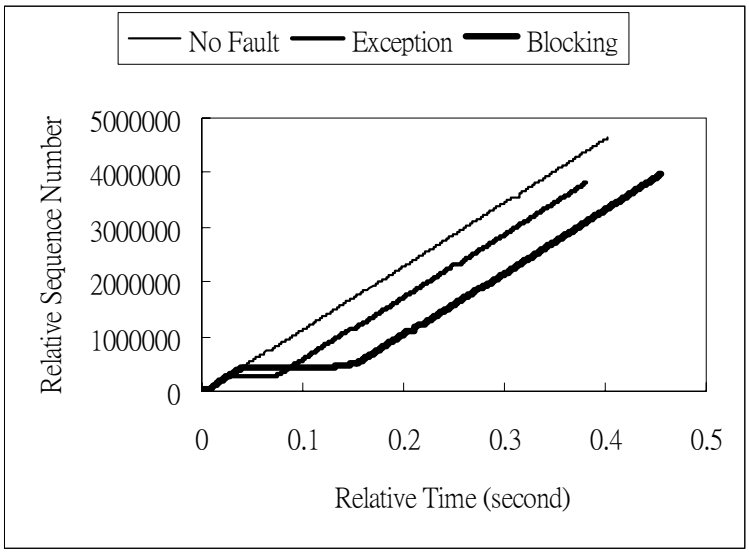


Figure 7. the Functionality of *n*Driver

4.2 Performance Overhead

We use two benchmarks, Netperf [12] and Webstone [1], to measure the overhead of *n*Driver.

4.2.1 Micro Benchmark: Netperf

We use the utility, Netperf, to measure the network throughput and CPU utilization of a machine with *n*Driver in order to compare it with a machine without *n*Driver. During each experiment, it will send as many fixed-size messages as possible.

Figure 8 shows the network throughput of each experiment. When the message size is equal to or more than 16 bytes, the network throughput is limited by the network maximum bandwidth (i.e. 100Mbps) and there is no visible network throughput degradation. But when the message size is under 16 bytes, the average network throughput degradation is 5%. Besides, Figure 9 shows the CPU utilization of each experiment. When the message size is under 16 bytes, the CPU utilization of both conditions is 100%. When the message size is under 16 bytes, the average overhead of *n*Driver is between 3% and 5%. The overhead mainly results from the maintenance of the action list and the software timer.

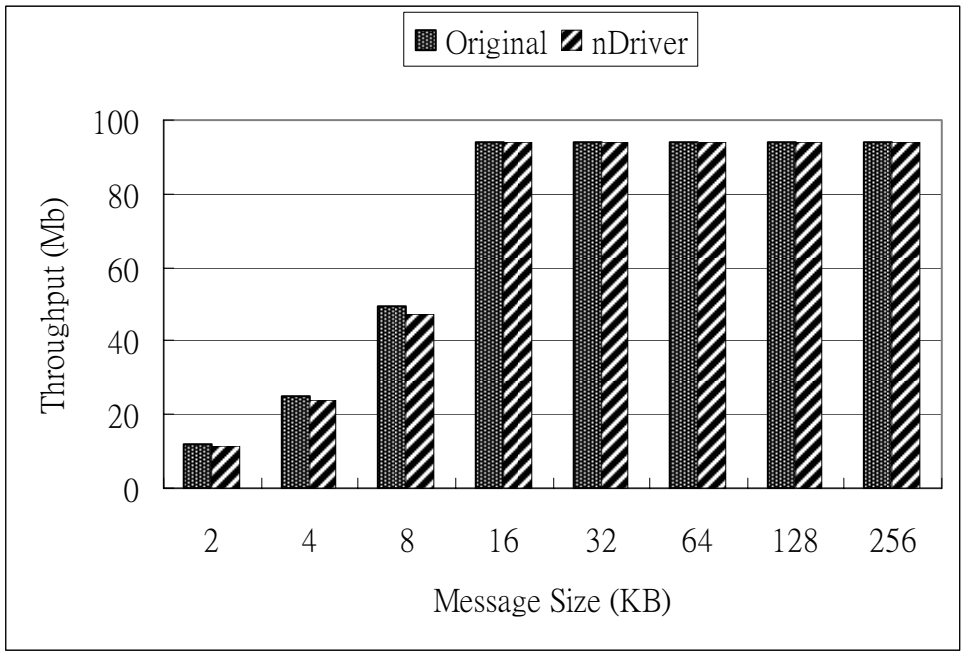


Figure 8. Throughput of a Machine with and without *n*Driver

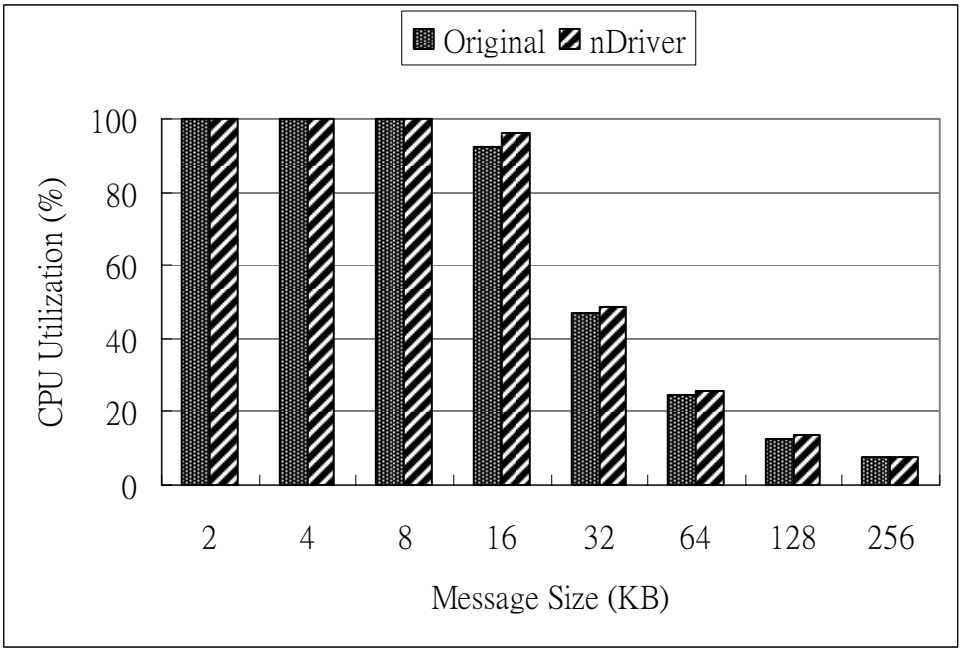


Figure 9. CPU Utilization of a Machine with and without *n*Driver

4.2.2 Macro Benchmark: Webstone

In this subsection, we want to measure the overhead of *n*Driver under realistic workload. We install the Apache (version 2.0.40) Http server in the server machine. The two client machines are used to simulate the web clients. The workload is gotten from the Webstone benchmark, and each round lasts for 10 minutes. We measure the throughput under the two conditions: the server without *n*Driver and the server with *n*Driver. Figure 10 shows the throughput results. The x-axis represents the number of web clients. The more web clients, the more Http requests the web server processes. The y-axis represents the server throughput. From the figure we can see that the performance degradation is between 2.0%~3.5%. In addition, Figure 11 shows the average response time under the same experiments. The y-axis represents the average response time to access a web page. The average response time for the *n*Driver is higher than that for the original server without the *n*Driver by 2% to 3%.

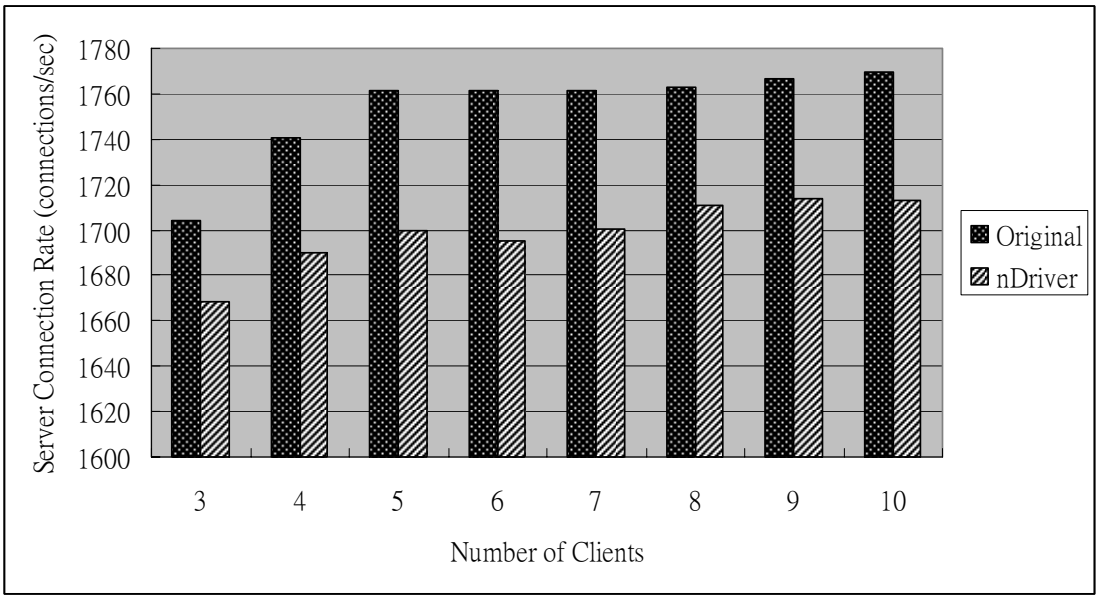


Figure 10. Throughput of the Http Server

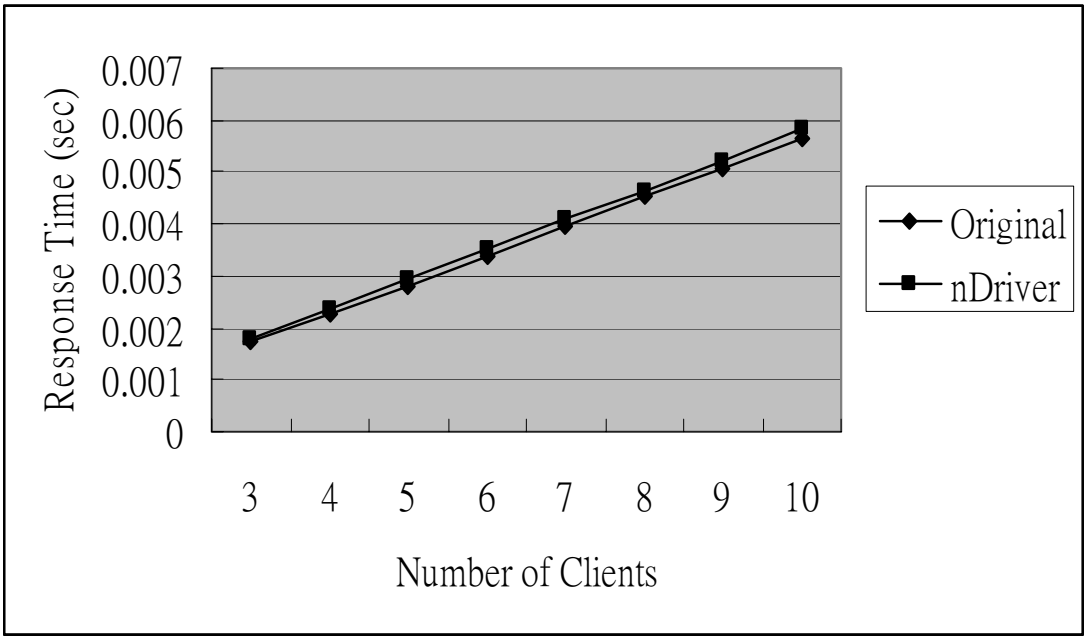


Figure 11. Response Time to Access Web Pages

4.3 Recovery time

In this subsection, we measure the time required by different parts of the recovery process. We manually insert a fault triggering the recovery process into the driver. As shown in figure 12, the recovery time consists of the following components. T*u* is the time that the undo manager spends in invoking the undo routine for each

entry in the action list. Ts is the time that the undo manager spends in swapping the drivers. T*c* is the time that the configuration manager spends in configuring the new driver. T*e* is the time spent in updating the external references. T*r* is the time spent in restoring system context.

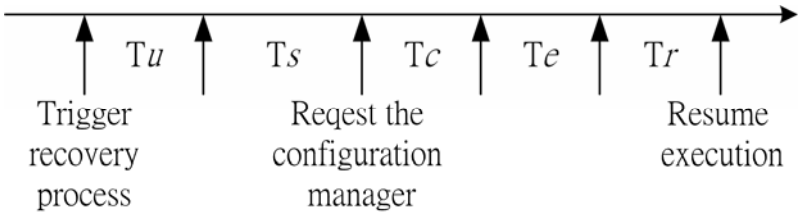


Figure 12. Different Parts of the Recovery Time

Table 1 shows the results, which are measured by using the Pentium Timestamp Counter [22]. From this table we can see that the total recovery time is very small.

Tu (us)	***Ts (us)***	***Tc (us)***	***Te (us)***	***Tr (us)***	***Ttotal (us)***
31.95	145.47	290.33	0.95	0.0092	468.71

Table 1. The Results of Different Parts of the Recovery Time

4.4 Per-Request Overhead

In this subsection, we measure the extra time the nDriver takes to process a request. (i.e., sending or receiving a packet) It consists of three parts: guard wrapper, maintaining the action list, and maintaining the unfinished list. The job of guard wrapper is to setup/stop fault detection and to save system context. Table 2 shows the results, which are also measured by using the Pentium Timestamp Counter. From the figure we can see that the overhead of guard wrapper is large compared to the two others because of the system context checkpointing. Besides, the total per-request overhead is small.

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

Table 2. The Results of Per-Request Overhead

5. Related Work

The related work falls into 3 categories: 1. improving driver quality, 2. dynamic replacement of kernel components, 3. fault tolerance in operating systems.

5.1 Improving the Driver Quality

Some techniques were proposed to help the driver developers to improve the design and reduce the bugs when developing drivers.

Microsoft and Intel [17] [9] provided guidelines for designing, implementing drivers for high availability systems. These guidelines cover the following aspects. First, they presented how to have good design and coding practices. Second, they mentioned that a device driver should provide statistics reporting, diagnosis tests, and event logging. Third, the way to perform lots of testing on a device driver was presented.

In order to improve the robustness of a driver, Lowell [16] 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 the checking, the compiler automatically generates low-level code, which is more error-prone, for driving the device hardware.

Microsoft suggests that hardware vendors should use Driver Verifier [X] to test their device drivers before releasing them. Besides, it can be used to detect the driver faults. Driver Verifier contains the following testing: simulating low resource conditions, verification of I/O, DMA verification, deadlock detection, and the like. However, it doesn't consider the problem how to recover from a driver fault.

5.2 Dynamic Replacement of Kernel Components

In Linux, drivers are usually implemented as modules [7]. And, dynamic module-loading can be used as a basic mechanism for hot-swapping module-based device drivers. However, it is not enough for recovery from a faulty driver. The reasons are as follows. First, a module cannot be removed unless its usage count becomes zero. Linux keeps a usage count for each module to tell the current number of users using the module. When a fault happens in the driver, the driver module is still in use and cannot be removed directly. Second, the goal of seamless driver swapping cannot be achieved by the dynamic module-loading mechanism since it doesn't consider undoing the kernel state changes, reconfiguring the new driver, and solving the problem of dangling references.

These software bugs may be avoided by design diversity to some extent. Design diversity [8][21] 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 while retrying different implementations.

Specifically, recovery block [21] uses a set of alternative implementations for the same application to improve the availability. If an alternative fails, another one will be tried. The *n*Driver framework realizes the concept of recovery block at the device driver layer. However, it is much more challenging to achieve the goal of seamless alternative swapping in the kernel code. Specifically, we have to address the issues that were not mentioned by the authors such as the undoing of the global kernel state changes made by the driver, the keeping of the driver requests, and the updating of the external references.

Soules [23] proposed a mechanism to replace an operating system component at 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, the mechanism is not appropriate for dealing with faults. This is because 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.

5.3 Fault Tolerance in Operating Systems

Process pair is used to implement fault-tolerant processes. There are two processes - primary and backup processes - for the same application. Normally, only the primary process provides services. If the primary process fails, the backup process takes over its services. However, the complexity synchronization between the two processes complicates the implementation. Moreover, the synchronization increases the runtime overhead.

The goal of High Availability Linux is to provide a high-availability clustering solution for Linux. It mainly provides two software packages, Hearbeat and Fake. Hearbeat is used to detect if hosts are available or not. A heartbeat is sent between hosts periodically. If a heartbeat isn't received for a time, the host that doesn't send the heartbeat fails. Fake is used to take over the IP address of the failed host. Hosts in a cluster can use Heartbeat to monitor the availability of hosts that provide services. When hosts that provide services fail, others can take over the services by using Fake.

Swift [24] introduced an approach to enhance operating system reliability by isolating the kernel from extensions (including drivers) failures. It executes each extension in a lightweight kernel protection domain to prevent it from corrupting the kernel. In addition, it keeps track of the kernel resources used by the extension to

perform automatic clean-up during recovery. However, it doesn't try to recover the extension. As a result, the applications or services relying on the driver extension will become unavailable. In *n*Driver, we take a more aggressive approach. The extension can be recovered by either a refresh instance of the same implementation or another implementation. This improves the system availability further.

5.4 Others

Autonomic Computing (Kephart and Chess, 2003) is proposed by IBM, which enables systems to manage themselves according to the administrator's goals. The self-managing means self-configuring, self-healing, self-protecting, and self-optimizing. Especially, the self-healing techniques automatically detect, diagnose, and repair software and hardware problems. Some efforts related to the self-healing are SRIRAM (Verma et al., 2003) which is a method that facilitates instantiating mirroring and replication of services in a network of servers, K42 (Appavoo et al., 2003) which allows software codes including system monitoring and diagnosis functions to be inserted and removed dynamically without shutting down the running system, and Dynamic CPU Sparing (Jann et al., 2003) that predicts the defective of a CPU and replace it with a spare one.

Recovery-Oriented Computing (Patterson et al., 2002) proposed by U. C. Berkeley and Stanford University is a related effort to autonomic computing. It proposes new techniques to deal with hardware faults, software bugs, and operator errors. These techniques include Pinpoint (Chen et al., 2002) which finds the root cause of a system failure in an efficient way, System Undo (Brown and Patterson, 2003) which can perform system recovery from operator errors, and Recursive Restart (Candea et al., 2002) which reduces the service downtime. In addition, they also proposed on-line fault injection and system diagnosis to improve the robustness of the

system.

Checkpointing [15][2][25][20][26] is a common technique for system recovery. It saves system state periodically or before entering critical regions. If a system fails, it can be recovered by restoring the last checkpointed state. The major problem of checkpointing is that it can not make the system survive from 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 vast amounts of state need to be saved.

Lakamraju [14] introduced a low-overhead fault tolerance technique to recover from only network processor hangs in Myrinet. 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 only focuses on hardware failures instead of software errors. The former is easier to handle since it doesn't consider the complex software state maintenance problem such as undoing the kernel state changes, reconfiguring the new driver, and solving the problem of dangling references.

6. Conclusion

In this thesis, we propose the *n*Driver framework, which uses multiple implementations of a device driver to survive from driver faults. It can detect two major types of driver faults, the exception and blocking faults. With the help of *n*Driver, driver faults will not always result in kernel panics or system hangs. Instead, if it a fault is detected, *n*Driver substitutes another driver implementation with the faulty one to make the system continue working. In order to achieve the goal of seamless driver swapping, *n*Driver undoes the kernel state changes made by the faulty driver, keeps the unfinished driver requests, and redirects the external references. In addition, *n*Driver blocks the driver removing and installation events so that the other

kernel subsystems are not aware of the driver swapping.

The major contribution of our work is that nDriver realizes the concept of recovery blocks at the device driver layer. It achieves the goal of seamless driver swapping. However, it improves operating system availability without modifying the existing operating system or driver codes.

We implement *n*Driver as a kernel module in Linux. Currently, it can recover from faults in network and block device drivers. According to the performance evaluation, the overhead of *n*Driver is no more than 5% and the recovery time is very small. This indicates that *n*Driver is an efficient mechanism to increase the availability of operating systems.

References

[1] [Surge] Barford, P., and Crovella, M. E.. “Generating Representative Web Workloads for Network and Server Performance Evaluation”. In: Proceedings of the ACM SIGMETRICS '98, pp. 151-160.

[2] [CP2] Subhachandra Chandra, Peter M. Chen. “Whither Generic Recovery From Application Faults? A Fault Study using Open-Source Software”. In proceedings of the 2000 International Conference on Dependable Systems and Networks / Symposium on Fault-Tolerant Computing (DSN/FTCS), June 2000.

[3] [sw-aging] 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 JRD, Vol. 45, No. 2, March 2001.

[4] [error_distr_0] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler. “An Empirical Study of Operating System Errors”. In proceedings of the 18th ACM symposium on Operating systems principles, pp. 73–88, Banff, Alberta, Canada, 2001.

[5] [bonding] Davis, T.. “Linux Channel Bonding”. Available at http://www.sourceforge.net/projects/bonding/usr/src/linux/Documentation/networking/bonding.txt.

[6] [error_distr_4] Gray, J.; Siewiorek, D.P.; “High-availability computer systems”. Computer, Volume: 24, Issue: 9, pp. 39-48, Sept. 1991.

[7] [LLKM_HOWTO] Bryan Henderson. “Linux Loadable Kernel Module HOWTO”. Available at http://www.tldp.org/HOWTO/Module-HOWTO/.

[8] [design_diversity] Chris Inacio. “Software Fault Tolerance”. Available at http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/index.html.

[9] [Harden] Intel Corporation, IBM Corporation. “Device Driver Hardening”.

Available at http://hardeneddrivers.sourceforge.net/.

[10] [Intel] Intel Corporation, 2003. “Intel Networking Technology – Load Balancing”. Available at http://www.intel.com/network/connectivity/resources/technologies/load_balancing.htm.

[11] [pSeries] Jann, J., Browning, L. M., and Burugula, R. S.. “Dynamic Reconfiguration: Basic Building Blocks for Autonomic Computing on IBM pSeries Servers”, IBM Systems Journal, 42(1): 29–37.

[12] [netperf] Rick Jones. “Netperf benchmark”. Available at http://www.netperf.org/netperf/NetperfPage.html.

[13] [error_distr_2] “Kernel Summit 2003: High Availability”. Available at http://lwn.net/Articles/40620/.

[14] [Myrinet] Vijay Lakamraju, Israel Koren, C.M. Krishna. “Low Overhead Fault Tolerant Networking in Myrinet”. 2003 International Conference on Dependable Systems and Networks (DSN'03), San Francisco, California. June 22-25, 2003.

[15] [CP1] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. “Exploring Failure Transparency and the Limits of Generic Recovery”. In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), October 2000.

[16] [Devil] Fabrice Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, Gilles Muller. “Devil: An IDL for Hardware Programming”. In Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, California, October 2000.

[17] [WinHEC] Microsoft Corporation. “Writing Drivers for Reliability, Robustness and Fault Tolerant Systems”. Microsoft Windows Hardware Engineering Conference (WinHEC), 2002.

[18] [WHY] David Oppenheimer, Archana Ganapathi, and David 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 (USITS '03), 2003.

[19] [RAID] Patterson, D. A., Chen, P., Gibson, G., and Katz, R.H.. “Introduction to Redundant Arrays of Inexpensive Disks (RAID)”. In: Digest of Papers for 34th IEEE Computer Society International Conference (COMPCON Spring '89), pp. 112 -117.

[20] [CP4] James S. Plank, Micah Beck, Gerry Kingsley, Kai Li. “Libckpt: Transparent Checkpointing under Unix”. Usenix Winter 1995 Technical Conference, pp. 213 - 223, New Orleans, LA, January, 1995.

[21] [recovery_blocks] B. Randell and J. Xu. “The Evolution of the Recovery Block Concept”. Software Fault Tolerance, John Wiley & Sons, pages 1-21, New York, 1995.

[22] [TimeStamp] Rubini, A.. “Making System Calls from Kernel Space”. Linux Magazine, Nov. 2000. Available at http://www.linux-mag.com/2000-11/gear_01.html.

[23] [Online] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, Jimi Xenidis. “System Support for Online Reconfiguration”. In Proceedings of the USENIX 2003 Annual Technical Conference, pp. 141-154, San Antonio, June 9-14, 2003.

[24] [Nook_SOSP] Michael Swift, Brian N. Bershad, and Henry M. Levy. “Improving the Reliability of Commodity Operating Systems”. In proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, Oct. 2003.

[25] [CP3] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung and Chandra Kintala. “Checkpointing and Its Applications”. In proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, pp. 22, 1995.

[26] [CP5] Avi Ziv, Jehoshua Bruck. “An On-Line Algorithm for Checkpoint Placement”. Computers, IEEE Transactions on , Volume: 46 , Issue: 9 , pp. 976 - 985 , Sept. 1997.