Improving the Reliability of Commodity Operating Systems

Michael M. Swift

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Michael M. Swift

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

Henry M. Levy

Brian N. Bershad

Reading Committee:

Henry M. Levy

Brian N. Bershad

John Zahorjan

Date:

Signature______________________________

Date__________________________________

University of Washington

Abstract

Improving the Reliability of Commodity Operating Systems

Michael M. Swift

Co-Chairs of Supervisory Committee:

Professor Henry M. Levy
Computer Science and Engineering

Associate Professor Brian N. Bershad
Computer Science and Engineering

This dissertation presents an architecture and mechanism for improving the reliability of commodity operating systems by enabling them to tolerate component failures.

Improving reliability is one of the greatest challenges for commodity operating systems, such as Windows and Linux. System failures are commonplace in all domains: in the home, in the server room, and in embedded systems, where the existence of the OS itself is invisible. At the low end, failures lead to user frustration and lost sales. At the high end, an hour of downtime from a system failure can result in losses in the millions.

Device drivers are a leading cause of failure in commodity operating systems. In Windows XP, for example, device drivers cause 85% of reported failures [160]. In Linux, the frequency of coding errors is up to seven times higher for device drivers than for the rest of the kernel [45]. Despite decades of research in reliability and extensible operating system technology, little of this technology has made it into commodity systems. One major reason for the lack of adoption is that these research systems require that the operating system, drivers, applications, or all three be rewritten to take advantage of the technology.

In this dissertation I present a layered architecture for tolerating the failure of *existing* drivers within *existing* operating system kernels. My solution consists of two new techniques for isolation drivers and then recovering from their failure.

First, I present an architecture, called Nooks, for isolating drivers from the kernel in a new protection mechanism called a *lightweight kernel protection domain*. By executing drivers within a domain, the kernel is protected from their failure and cannot be corrupted.

Second, I present a mechanism, called *shadow drivers*, for recovering from device driver failures. Based on a replica of the driver's state machine, a shadow driver conceals the driver's failure from applications and restores the driver's internal state to a point where it can process requests as if it had never failed. Thus, the entire failure and recovery is transparent to applications.

I also show that Nooks functions as a platform for higher-level reliability services. I leverage the Nooks and shadow driver code to remove another major source of downtime: reboots to update driver code. This service, *dynamic driver update*, replaces drivers online, without restarting the OS or running applications and without changes to driver code. This service requires little additional code, and demonstrates that Nooks and shadow drivers provide useful services for constructing additional reliability mechanisms.

I implemented the Nooks architecture, shadow drivers, and dynamic driver update within the Linux operating system and used them to fault-isolate several device drivers and other kernel extensions. My results show that Nooks offers a substantial increase in the reliability of operating systems, catching and quickly recovering from many faults that would otherwise crash the system. Under a wide range and number of fault conditions, I show that Nooks recovers automatically from 99% of the faults that otherwise cause Linux to crash. I also show that the system can upgrade existing drivers without rebooting. Finally, I show that the system imposes little performance overhead for many drivers.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Number **Page**

LIST OF TABLES

Table Number **Page**

ACKNOWLEDGMENTS

Thank you all!

I would like to thank my advisors, Hank Levy and Brian Bershad, who encouraged me in this endeavor and taught me how to do research. When I entered graduate school, I thought I knew everything. From Hank and Brian I learned patience, as research is not an overnight process. I learned the value of a good question. I learned the value of a good story. And most importantly, I learned the value of a clear presentation, because ideas are useless if nobody understands them.

I would also like to thank all the students who participated in the Nooks project. Steve Martin provided early testing. Doug Buxton and Steve Martin implemented our wrapper generator. Leo Shum implemented sound-card driver support Muthu Annamalai implemented IDE support. Christophe Augier performed the fault injection testing for the SOSP paper and greatly enhanced Nooks recovery ability through a scripting language. Damien Martin-Guillerez improved the wrapper code and implemented dynamic driver update. Brian Milnes performed performance testing and optimization, using super-pages to reduce the number of TLB misses. Micah Brodsky and Eric Kochhar implemented a reliability subsystem in the Windows kernel and provided great feedback on the assumptions of my system.

I would also like to thank Rob Short at Microsoft, who provided valuable information about Windows failures and the problem being faced in its development.

Finally, I would like to thank my parents, Alice and Arthur Swift, who allowed me to choose any career I wanted but are secretly happy that I've followed in their footsteps.

DEDICATION

To my family: Suzanne, Jacob and Lily, who encouraged me in my passions, bravely weathered the resulting absences, and followed me to a land of snow and ice.

Chapter 1

INTRODUCTION

This dissertation presents an architecture and mechanism for improving the reliability of commodity operating systems by enabling them to tolerate component failures.

Improving reliability is one of the greatest challenges for commodity operating systems. System failures are commonplace and costly across all domains: in the home, in the server room, and in embedded systems, where the existence of the OS itself is invisible. At the low end, failures lead to user frustration and lost sales. At the high end, an hour of downtime from a system failure can lead to losses in the millions [77].

1.1 Motivation

Three factors motivated my research. First, computer system reliability remains a crucial but unsolved problem [86, 164]. This problem has been exacerbated by the adoption of commodity operating systems, designed for best-effort operation, in environments that require high availability [187, 20, 4, 56]. While the cost of high-performance computing continues to drop because of commoditization, the cost of failures (e.g., downtime on a stock exchange or e-commerce server, or the manpower required to service a help-desk request in an office environment) continues to rise as our dependence on computers grows. In addition, the growing sector of "unmanaged" systems, such as digital appliances and consumer devices based on commodity hardware and software [102, 194, 180, 184, 179], amplifies the need for reliability.

Second, OS extensions have become increasingly prevalent in commodity systems such as Linux (where they are called *modules* [31]) and Windows (where they are called *drivers* [55]). Extensions are optional components that reside in the kernel address space and typically communicate with the kernel through published interfaces. Common extensions include device drivers, file systems, virus detectors, and network protocols. Drivers now account for over 70% of Linux kernel code [45], and over 35,000 different drivers with over 112,000 versions exist on Windows XP desktops [151, 160]. Unfortunately, most of the programmers writing drivers work for independent hardware vendors and have significantly less experience in kernel organization and programming than the programmers that build the operating system itself.

Third, device drivers are a leading cause of operating system failure. In Windows XP, for example, drivers cause 85% of reported failures [160]. In Linux, the frequency of coding errors is up to seven times higher for device drivers than for the rest of the kernel [45]. While the core operating system kernel can reach high levels of reliability because of longevity and repeated testing, the *extended* operating system cannot be tested completely. With tens of thousands of drivers, operating system vendors cannot even identify them all, let alone test all possible combinations used in the marketplace. In contemporary systems, any fault in a driver can corrupt vital kernel data, causing the system to crash.

1.2 Thesis and Contributions

My thesis is that operating system reliability should not be limited by extension reliability. An operating system should never crash due to a driver failure. Therefore, improving OS reliability will require systems to become highly tolerant of failures in drivers and other kernel extensions. Furthermore, the hundreds of millions of existing systems executing tens of thousands of drivers demand a reliability solution that is both *backward compatible* and *efficient* for common device drivers. Backward compatibility improves the reliability of already deployed systems. Efficiency avoids the classic trade-off between robustness and performance.

To this end, I built a new operating system subsystem, called Nooks, that allows existing device drivers and other extensions to execute safely in commodity kernels. Nooks acts as a layer between extensions and the kernel and provides two key services–isolation and recovery. This thesis describes the architecture, implementation and performance of Nooks.

Nooks allows the operating system to tolerate driver failures by isolating the OS from device drivers. With Nooks, a bug in a driver cannot corrupt or otherwise harm the operating system. Nooks contains driver failures with a new isolation mechanism, called a *lightweight kernel pro-*

tection domain, that is a privileged kernel-mode environment with restricted write access to kernel memory. Nooks interposes on all kernel-driver communication to track and validate all modifications to kernel data structures performed by the driver, thereby trapping bugs as they occur and facilitating subsequent recovery. Unlike previous safe-extension mechanisms, Nooks supports existing, unmodified drivers and requires few changes to existing operating systems.

When a driver failure occurs, Nooks detects the failure with a combination of hardware and software checks and triggers automatic recovery. Nooks restores driver operation by unloading and then reloading the driver. This ensures that the driver services are available to applications after recovery.

With Nooks' recovery mechanism, the kernel and applications that do not use a failed driver continue to execute properly, because they do not depend on the driver's service. Applications that depend on it, however, cannot. These failures occur because the driver loses application state when it restarts, causing applications to receive erroneous results. Most applications are unprepared to cope with this. Rather, they reflect the conventional failure model: drivers and the operating system either fail together or not at all.

To remedy this situation, I introduce a new kernel agent, called a *shadow driver*, that conceals a driver's failure from its clients while recovering from the failure. During normal operation, the shadow tracks the state of the real driver by monitoring all communication between the kernel and the driver. When a failure occurs, the shadow inserts itself *temporarily* in place of the failed driver, servicing requests on its behalf. While shielding the kernel and applications from the failure, the shadow driver restarts the failed driver and restores it to a state where it can resume processing requests as if it had never failed.

The two services provided by Nooks and shadow drivers, transparent isolation and recovery, provide a platform for constructing additional reliability services. I built a third service on top of these components that enables device drivers to be updated on-line without rebooting the operating system. This new mechanism, called *dynamic driver update*, relies upon shadow drivers both to conceal the update and to transfer the state of the old driver to the new driver automatically, ensuring that driver requests will continue to execute correctly after the update. Dynamic driver update improves availability by removing the lengthy reboot currently required to apply critical driver patches.

My focus on reliability is not new. The last forty years has produced a substantial amount of research on improving reliability for kernel extensions by new kernel architectures [51, 212, 71, 97], new driver architectures [169, 83], user-level extensions [82, 132, 212], new hardware [74, 209], and type-safe languages [24, 87, 110].

While previous systems have used many of the underlying techniques in Nooks, my system differs from earlier efforts in three key ways. First, I target an existing set of extensions, device drivers, for existing commodity operating systems instead of proposing a new extension architecture. I want today's drivers to execute on today's platforms without change if possible. Second, I use C, a conventional programming language. I do not ask developers to change languages, development environments, or, most importantly, perspective. Third, unlike earlier safe-extension systems that were only designed to isolate extensions, Nooks also recovers by restarting drivers and concealing failures from applications. Nooks does not require modifications to applications to tolerate driver failures. Overall, I focus on a single and very serious problem – reducing the huge number of crashes resulting from drivers and other extensions – instead of a more general-purpose but higher-cost solution.

I implemented a prototype of Nooks in the Linux operating system and experimented with a variety of device drivers and other kernel extensions, including a file system and a kernel-mode Web server. Using synthetic fault injection [108], I show that when I inject bugs into running extensions, Nooks can gracefully recover and restart the extension in 99% of the cases that cause Linux to crash. Recovery occurs quickly, as compared to a full system reboot, and is unnoticed by running applications in 98% of tests. The performance overhead is low to moderate, depending on the frequency of communication between the driver and the kernel. I also show that dynamic driver update allows transparent, on-line upgrade between a wide range of driver versions. Finally, of the 14 drivers I isolated with Nooks, none require code changes. Of the two kernel extensions, one required no changes and only 13 lines changed in the other.

Although I built the prototype on the Linux kernel, I expect that the architecture and many implementation features would port readily to other commodity operating systems, such as BSD, Solaris, or Windows.

1.3 Dissertation Organization

In this dissertation, I describe the architecture and implementation of Nooks and evaluate its effectiveness and performance cost. Chapter 2 provides background material on reliability, failure models, and the causes of operating system failure. I also discuss previous work on operating system reliability, including techniques for avoiding bugs from OS code and for tolerating faults at run-time. As this dissertation concerns device drivers, Chapter 3 presents background material on driver architectures and operations.

Chapter 4 describes the Nooks Reliability Layer, which prevents drivers from corrupting the OS by isolating drivers from the operating system kernel. After a failure, Nooks recovers by unloading and reloading the failed driver. I first describe the architecture of Nooks at a high level and then discuss the details of the Nooks implementation in Linux.

Chapter 5 describes shadow drivers, a replacement recovery mechanism for Nooks that conceals the failure from applications and the OS. This allows applications that depend on a driver's service to execute correctly in the presence of its failure. Again, I provide a high-level view of shadow driver operation and then discuss the implementation details.

Following the chapters on Nooks and shadow drivers, Chapter 6 analyzes the effectiveness of Nooks and shadow drivers at (1) preventing drivers from crashing the OS and (2) recovering from driver failures. I also analyze the performance cost of using Nooks with common drivers and applications.

Chapter 7 describes dynamic driver update, a mechanism based on shadow drivers to update device drivers on-line. I present an evaluation showing that dynamic update works for most common driver updates and imposes a minimal performance overhead.

In Chapter 8 I discuss the lessons learned in implementing Nooks. I evaluate the impact of operating system architecture on the Nooks architecture by comparing the implementation on Linux to a partial implementation on Windows. I discuss opportunities for future research in Chapter 9.

Finally, in Chapter 10 I summarize and offer conclusions.

Chapter 2

BACKGROUND AND RELATED WORK

The goal of my work is to improve the reliability of commodity operating systems, such as Linux and Windows. Reliability has been a goal of computer systems designers since the first days, when hardware failures were orders of magnitude more common then they are today [204, 63]. As hardware reliability has improved, software has become the dominant source of system failures [91, 150]. My work therefore focuses on improving operating system reliability in the presence of software failures.

In this section, I present background material on reliable operating systems. First, I define the terms used to discuss reliable operation. I then discuss software failures and their manifestations. Finally, I describe previous approaches to building reliable systems and their application to commodity operating systems.

2.1 Definitions

Reliability is defined as the expected time of correct execution before a failure and is measured as the mean time to failure (MTTF) [93]. Availability is the proportion of time that the system is available to do useful work. Availability is calculated from the MTTF and the duration of outages (the mean time to repair, or MTTR) as MTTF/(MTTF+MTTR) [93]. My work seeks to improve reliability by reducing the frequency of operating system failures, and to improve availability by reducing the time to repair, or recover from a failure.

The events that cause a system to be unreliable can also be precisely defined [93, 10]. A *fault* is a flaw in the software or hardware. When this fault is executed or activated, it may corrupt the system state, leading to an *error*. If the error causes the system to behave incorrectly, a *failure* occurs. An error is termed *latent* until it causes a failure, at which point it becomes *effective*.

As an example, a bug in a device driver that corrupts memory is a fault. When it executes and actually corrupts memory, the resulting corruption is an error. Finally, a failure occurs when the kernel reads the corrupt memory and crashes the system.

2.2 System Models

There exist a wide variety of operating systems that target different environments. For example, the QNX operating system targets real-time services on embedded devices [104]. Unicos targets high-performance applications on supercomputers [29], and the Tandem Guardian operating system targets application with extreme reliability needs, such as stock exchanges [128]. These *special-purpose* operating systems are optimized for the unique properties of their operating environment and workload, for example by trading peak performance for predictable scheduling.

In contrast, *commodity* operating systems, such as Linux [31] and Windows [173], are sold in high volumes and used in a wide variety of environments. For example, in addition to use on desktops and in data centers, Linux has been used on wristwatches [156] and in planetary exploration [4]. As a result, commodity operating systems cannot be over-optimized for a single target environment. Instead, they are designed with a variety of competing goals, such as cost, size, peak performance, features, and time-to-market. Furthermore, the huge installed base of these operating systems forces compatibility with existing code to be a priority. As a result, any attempt to improve the reliability of commodity operating systems must take into account the wide variety of demands placed on the system.

2.3 Failure Models

Failures occur when a fault is activated and causes an effective error. The circumstances of *when* and *how* a fault causes a failure, and the failure's behavior, determine what a system must do to tolerate the failure. As a result, designers make many assumptions about how failures occur in order to build a cost-effective system that prioritizes frequent and easy-to-handle failures. For example, most systems are built with the assumption that memory will not become corrupted silently. As a result, these systems avoid the cost of error checking every memory operation. If silent memory corruption occurs, the system may cease functioning. The same system, though, may also assume that a network can corrupt data in transmission and, as a result, checksums packet contents to detect corruption. Thus, the assumptions about what failure occurs and their frequency drive system design in terms of what mechanisms are used to detect and tolerate failures.

A system's assumptions about failures are called its *failure model*. This model specifies any dependency that a system has on the properties of failures. In the context of software reliability, failure models commonly specify two major properties of failures: how the system behaves when it fails and how failures occur. How the system behaves determines what mechanisms are required to detect failures. How failures occur determines whether a failure can be masked from other system components.

A failure model may also incorporate assumptions about the system's response to a component failure. Rarely do entire systems fail at once. Generally, a single component fails, which if not handled, leads to a system failure. To simplify failure handling, a system designer may assume that failures are contained to a single component and detected before any other component has been corrupted. In contrast to the previous assumptions, this is an assumption about the *system* and not the failure, and must therefore be enforced by the implementation.

2.3.1 How Failures Behave

The failure behavior of a component determines how a system contains and detects failures. For example, a system can detect component crashes with the processor's exception mechanism. To contain a component that fails by producing incorrect outputs, the system must check all outputs for correctness before other components observe the output and become corrupted. In contrast, if a component fails by stopping, then no work may be needed to contain its failure.

We can categorize failures by their behavior in order to detect and tolerate different behaviors with different mechanisms. Previous studies have identified five major categories of failure behavior [54]:

1. *Timing* failures occur when a component violates timing constraints.
2. *Output* or *response* failures occur when a component outputs an incorrect value.
3. *Omission* failures occur when a component fails to produce an expected output.
4. *Crash* failures occur when the component stops producing any outputs.
5. *Byzantine* or *arbitrary* failures occur when any other behavior, including malicious behavior,

occurs.

Each failure category requires its own detection and containment mechanisms. For example, omission failures require that the system predict the outputs of a component to detect when one is missing, while timing failures require timers that detect when an output is delayed.

For simplicity, reliable systems commonly assume that only a subset of these behaviors occurs and remove from consideration the other failure modes. For example, Backdoors [27] assumes that only timing and crash failures occur. Fortunately, many failures are of these types, although the more general arbitrary failure has been studied in context of some byzantine-tolerant research operating systems [177, 37].

2.3.2 How Failures Occur

How a fault is activated and causes a failure determines what strategies are possible for tolerating the failure. A fault that is activated by a single, regularly occurring event is called a *deterministic* failure, or a *bohrbug* [91]. Any time that event occurs, the failure occurs. As a result, a recovery strategy for deterministic failures must ensure that the triggering event does not occur again during recovery. If the triggering event is a specific service request, that request must be canceled, because it can never be completely correctly.

In contrast, a fault that is activated by a combination of events that occur irregularly is called a *transient* failure, or a *heisenbug*. These failures are triggered by a combination of inputs, such as the exact interleaving of requests and other properties of the system environment that do not recur often [91]. As a result, the recovery strategy need not prevent the failure from recurring–it is probabilistically unlikely to happen. Requests being processed at the time of failure can be retried automatically, which conceals the failure from the component's clients. Empirical evidence suggests that many failures are transient [91] and as a result many fault-tolerant systems ignore deterministic failures. For example, the Phoenix project [15], the TARGON/32 fault-tolerant Unix system [28] and Shah et al's work on fault-tolerant databases [183] all assume that all failures are transient.

2.3.3 System Response to Failure

When a component failure occurs, the system may *translate* the failure from one category to another. For example, Windows XP crashes when it detects that a device driver passes a bad parameter to a kernel function. Thus, the OS translates an output failure of a component into a crash failure of the system. Failure translation allows higher levels of software to process a single category of failure rather than manage the variety of component failures that may occur.

Many fault-tolerant systems take this approach with the *fail-stop processor* model [175], whereby the system halts a component when it detects a failure. By halting, the component cannot further spread corruption. A system must meet three conditions to be fail-stop:

1. *Halt on failure*: the processor halts before performing an erroneous state transformation.
2. *Failure status*: the failure of a processor can be detected.
3. *Stable storage*: the processor state can be separated into volatile storage, which is lost after a failure, and stable storage, which is preserved, unaffected by the failure.

The benefit of the fail-stop model is that it greatly simplifies construction of a reliable system: it is simple to distinguish between correct and failed states and memory is similarly partitioned into correct and corrupted storage. Thus, the system and its components are only concerned with these two states and not with other failure modes. Many fault-tolerant systems, such as Phoenix [15] assume that failures are fail stop. Others, such as Hive [39] incorporate failure detectors that translate output failures into fail-stop failures.

The drawback of the fail-stop model is that it is difficult to achieve in practice and does not adequately capture all the properties of the system, such as intermittent failures [8]. Furthermore, without strong isolation that contains faults to a single component and good failure detectors, it can be difficult to ensure fail-stop behavior. For example, a study by Chandra and Chen shows that application failures are often detected *after* corrupt data is written to disk [38], violating the fail-stop conditions. Additional error checking internal to the application may be able to reduce the frequency of the violations.

2.3.4 Summary

How failures occur and how failures behave determine the possible strategies for tolerating failures. If failures are transient, then a recovery strategy may retry operations in progress at the time of failure. In contrast, if failures are deterministic, care must be taken to avoid repeating the triggering event. Similarly, if failures have easily detected behavior, such as crashes, or are translated into fail-stop failures, then the system can avoid managing complex error states.

2.4 Properties of OS Failures

Within the OS, many failures can be traced to operating system extensions. For example, recent data on the Windows NT family of operating systems from Microsoft implicates operating system extensions as the major source of failures, particularly as the core operating system itself becomes more reliable [150]. According to this study, the core operating system in Windows NT 4 was responsible for 43% of system failures, while drivers and other kernel extensions were responsible for 32%. In Windows 2000, the follow-on to NT 4, the core operating system was responsible for only 2% of failures, while drivers grew to 38% (the remaining failures were caused by faulty hardware and system configuration errors). More recently, drivers caused over 85% of Windows XP crashes [160]. Hardware failures remain a significant, but not dominant, source of system crashes: for Windows Server 2003, 10% of failures are due to hardware problems [140]. However, approximately half of the crashes from hardware faults manifest as driver failures. These driver failures could be prevented with a “hardened” driver that is able to mask the hardware fault [9].

While there are no published crash-reports for the Linux operating system, an automated study of the Linux 2.4 kernel source code showed similar trends. Device drivers, which comprised 70% of the source code, had between two and seven times the frequency of coding errors (errors per line of code) as other kernel code [45]. Thus, drivers are likely a major source of failures in Linux as well.

Many studies of operating system failures have shown that these failures are often transient, i.e., they do not recur when the same operation is repeated [91, 92]. One reason is that kernel code is highly concurrent and may run simultaneously on multiple threads or processors. As a result, the exact ordering of events changes frequently. Other frequent changes in the kernel environment, such as the availability of memory and the arrival of interrupts, can also cause transient failures. In addition, the dominant manifestation of faults in operating systems is memory corruption [191, 101]. Thus, changes in memory-allocation patterns may also cause failures to be transient.

2.5 Previous Approaches to Building Highly Reliable Systems

Due to the need for reliable systems, many approaches have been tried in the past. At a high level, past systems have improved reliability with two general techniques: fault avoidance and fault tolerance. Fault avoidance seeks to avoid executing faulty code, whereas fault tolerance seeks to continue executing after a fault is activated.

2.5.1 Fault Avoidance

The goal of fault avoidance is to improve reliability by executing only correct code. The strategy can applied at all points during the lifecycle of software: during the design and coding process, when it is called *fault prevention*, after code is written, called *fault removal*, or at run time, called *fault work-arounds*.

Fault Prevention

The goal of fault prevention is to ensure that faults never make it into the source code. There are two major approaches to fault prevention: software engineering techniques and programming languages. Software engineering specifies how to write software. For example, modular software structure reduces faults by clarifying the interfaces between components [162, 163]. More recently, aspect-oriented software development (AOSD) systems promise to reduce software faults in operating systems by avoiding common cut-and-paste errors [47]. Software engineering processes, such as RTCA DO178B for aviation software [111] and the Capability Maturity Model's Level 5 [165], address not only coding but also planning, documentation, and testing practices. These processes seek to turn the craft of programming into a rigorous engineering discipline.

Programming languages prevent faults by preventing compilation of faulty code. Type-safe programming languages and run-time systems used for operating systems, such as Modula 2+ [141], Modula 3 [141, 24], Java [87], Cyclone [114], C♯ [110], Vault [58], and Ivy [34], prevent unsafe memory accesses and thereby avoid the major cause of failures. In addition to checking memory accesses, languages can provide compile-time checking of other properties. For example, the Vault language provides an extended type system that enforces ordering and data-flow constraints, such as the order of operations on a network socket [58]. To date, however, OS suppliers have been unwilling to implement system code in type-safe, high-level languages. Moreover, the type-safe language approach makes it impossible to leverage the enormous existing code base. Recent approaches that add type-safety to C, such as CCured, remove the cost of porting but increase run-time costs substantially because of the need to track the types of pointers [50].

Other language-based approaches seek to improve reliability by automating error-prone manual tasks. Aspect-oriented software development (AOSD) automates the application of changes that cut across a large body of code [118] and has been applied to operating system kernels [47]. Tarantula applies AOSD to automate changes to the kernel-driver interface [127], while c4 uses aspect-oriented techniques to create a new programming model for cross-cutting kernel patches [80]. Compared to other language-based techniques, these approaches have the benefit that they are fully compatible with existing code in that they apply changes to code written in C.

Domain-specific languages can similarly improve reliability by simplifying complex and error-prone code. The Bossa project, for example, provides a language and run-time system for writing Linux schedulers that remove much of the complexity in current schedulers [30]. The Devil project simplifies writing driver code that interfaces with devices [142]. In most operating systems, the device interface is specified in a document describing the state machine of the device and the organization of registers exported by the device for communication. Based on this description, the driver writer must then implement code in C or another low-level language that interacts with the device. With Devil, a device vendor specifies the device-software interface in a domain-specific language. The Devil compiler then uses that specification to generate an API (i.e., C-language stubs) for the device. Driver writers call these functions to access the device. Devil prevents many of the bugs associated with drivers by abstracting away the complexities of communicating through I/O ports and memory-mapped device registers. This approach requires additional work by device manufacturers to produce the specification, but reduces the effort of driver programmers. However, it requires nearly complete rewriting of existing drivers, and hence is only applicable to new drivers. Devil complements the other language techniques, in that it addresses the hardware interface instead of the system interface to drivers.

While fault prevention is important to the writing of new code, it is of limited use for the millions of lines of existing code in operating systems. Fault removal is a more appropriate technique for this code.

Fault Removal

In contrast to fault prevention, which seeks to prevent faults from being written, fault removal seeks to detect faults in existing code. There are both process-oriented approaches to fault removal, such as code reviews [76], and technology approaches, such as static analysis. While programming practices are important and can have a huge impact on program quality [165], technology for automatically finding bugs is of more interest given the huge base of existing operating system code. For example, Microsoft Windows Server 2003 has over 50 million lines of code [193].

There are two major approaches to analyzing source code for faults: theorem proving and model checking [208]. Theorem proving works by converting the program code into logic statements that are then used to prove that the program never enters an illegal state. In contrast, model checking enumerates the program states along possible execution paths to search for an illegal state. Theorem proving generally scales better, because it manipulates formulas rather than program states. However, theorem provers often require human input to guide their execution towards a proof [208]. For this reason, model checkers have proven more popular as tools for a general audience [126].

Model checkers evaluate an abstraction of the code along all paths to ensure that it meets certain invariants, e.g., that memory that has been released is never accessed again. The output of a model checker may be: (1) an example path through the code that violates an invariant, (2) termination with the guarantee that the invariants are never violated, or (3) continued execution as the model checker works on ever-longer paths. To prove correctness, a model checker must check *every possible* state of a program, which is exponential in program size. Hence, model checkers have difficulty scaling because they must exhaustively enumerate the state space of a program. However, model checking can find bugs (but not prove correctness) by checking a much smaller number of states [211]. For example, Microsoft's PREfix and PREfast tools, which create a simplified abstraction of the source code to speed checking, do not prove that code is correct but are nonetheless useful for finding common programming errors [126].

Traditionally, model checking could not be applied to program code because of the complexity of evaluating the code directly. Instead, a programmer had to translate the code into a specification language such as Promela [67, 106], which is then checked. Recent language designs, such as ESP [124] and Spec♯ [18], allow model checking of executable code. The benefit of this approach is that no separate specification of the code is needed; instead the code itself is directly checked. The drawback is that it requires writing code in a new language and hence is not compatible with existing code or programming practices.

Several recent systems automatically compute a model from the existing source code, removing the need for new languages. The SLAM project from Microsoft creates binary programs from device drivers that can be checked [14]. Both SLAM and the similar Blast project [100] improve performance by only refining the models with additional detail when necessary. In addition, recent work has shown that it is possible to directly model check C code by executing it in a virtual machine [153, 152]. The virtual machine creates snapshots of each program state. However, model checking by direct execution is often slow, and it is difficult to isolate the state of the component being tested from the rest of the system [153]. As a result, this technique is not yet widely used.

Recently, model checking and theorem proving tools have been applied to existing operating system code [68, 13, 100]. The output of these tools is a list of sites in or paths through the code that may be faulty. However, finding bugs does not necessarily translate directly into more reliable software. First, even though a small fraction of faults cause the majority of failures [1], tools are also unable to determine statically whether a fault is a significant cause of failure [70]. Second, a programmer must still review the faults and manually correct the code–current tools cannot fix the code themselves [70]. Microsoft, for example, runs static analysis tools regularly and forces developers to fix the bugs they find [154]. However, Microsoft cannot force third-party developers to run the tools over extension code. Coverity runs similar tools over the Linux source tree and posts bug reports to the Linux Kernel Mailing List [53]. Again, Coverity only examines the code checked into the Linux kernel source tree. Many kernel extensions are maintained separately and released as patches to the kernel, and hence do not benefit from the analysis [138]. As a result, these tools often help the base operating system become more reliable but have less impact on extension reliability.

Fault Work-Arounds

When faulty code has been deployed and the resulting failures cannot be tolerated, fault work-arounds are the remaining possibility for attaining high reliability. Work-arounds prevent unsafe inputs from reaching a system. For example, a firewall is a fault work-around technique that prevents malicious requests from arriving at a network service.

In general, it is impossible to predict in advance the exact inputs that must be filtered to avoid failures [199, 134]. Thus, work-arounds are an ad-hoc technique for preventing failures with known causes when the software cannot be fixed. Given the wide variety of environments and inputs to commodity operating systems, this approach is therefore impractical as a general-purpose strategy for improving OS reliability.

Obtaining Specifications

A common problem for all fault-avoidance tools is the definition of correctness. Most software is written without a formal specification of its behavior. Indeed, writing a formal specification is prohibitively expensive: although the space shuttle control software contains only 425,000 lines of code, the specifications constitute 40,000 pages [79]. Therefore, many static analysis tools verify *partial* specifications, e.g., proving that invariants are not violated instead of complete correctness. The Static Driver Verifier from Microsoft [14], the Vault programming language for device drivers [58], and the CMC tool for checking network protocols and file systems [153, 152, 211] use this approach.

An existing body of code can be another source of specifications. If there are many implementations of an interface, then the code that deviates in its usage of the interface is likely to be incorrect. For example, if 99% of the implementers of an interface lock a parameter before accessing it, the remaining 1% of implementations that do not are probably faulty. This approach has been applied to the Linux kernel to identify both extension interface rules and violations of those rules [69].

A third approach to obtaining specifications for system software is to write them by hand. Formally verified operating system code [67, 198] and high-assurance systems [26] take this approach. However, only subsystems or small operating systems have been verified to date. Given the size of commodity operating systems, it is impractical to write a complete specification by hand.

Fault Avoidance Summary

Fault avoidance improves reliability by not executing faults. The choice between prevention, removal, and work around, depends on a projects stage of development. Fault prevention applies to systems in the design stage, when the system is being architected and implemented. Fault removal applies to existing code bases. Fault work-arounds apply to deployed systems and provide a stopgap solution until the system can be patched.

2.5.2 Fault tolerance

The previous section described methods for avoiding bugs by preventing faulty code from being executed. In contrast, fault tolerance seeks to continue correct execution in the presence of faulty code.

The core technique for tolerating failures is redundancy, so that a single failed computation does not terminate execution. The redundancy may come physically, by performing the computation on multiple processors, temporally, by re-executing the computation at a later time, or logically, by executing on a different copy of the data. A system may also use multiple implementations of a component to increase the probability that *some* computation will complete successfully. The core assumption behind redundancy is *failure independence*: the different implementations or executions will not all fail simultaneously.

At the program design level, there are two major categories of fault-tolerant designs: multi-version and single-version [195]. In multi-version designs, code components are implemented multiple times, providing *design diversity* and lessening the probability that all versions fail on the same inputs. When one version does fail, the other versions should (hopefully) continue to execute correctly. Single-version designs, in contrast, execute only one version of the code and instead rely on additional code to detect and recover from failures.

Three major multi-version approaches have been developed to provide both diversity and redundancy: recovery blocks [170], N-version programming [41], and N self-checking [125]. While the details differ, all three techniques require that programmers implement multiple versions of a component and a mechanism, either a checker or a voter, to select the correct output. Even with diversity, though, experimental evidence suggests that the different versions of a component often share failure modes, because the difficulty of producing the correct output varies widely across the space of inputs. Thus, different implementations are likely to fail on the same difficult inputs [121].

Implementing multiple versions of a program is time-consuming and expensive. The BASE technique reduces this cost by allowing off-the-shelf implementations of standard network services, such as a file server, to be configured into an N-version system [171]. An abstraction layer smoothes small differences between implementations, such as the granularity of timestamps and message ordering. This approach is limited to applications with well-specified behavior and clean, documented interfaces. As a result, it is not applicable to whole operating systems, which meet neither condition.

Due to the cost of multi-version approaches, most fault-tolerant systems opt for single-version redundancy and forgo the benefits of design diversity. Single-version fault tolerance depends on *error detection*, to notify the system that an error has occurred, and *recovery*, to transform the incorrect system state into a correct system state [10]. In addition, fault tolerant systems generally rely on isolation or containment to limit the scope of damage from a fault and to prevent errors from becoming failures [59]. In the next sections I further discuss isolation, error detection, and recovery as it applies to operating systems.

Isolation

Isolation, which confines errors to a single component, has long been recognized as the key to fault tolerance because it allows components to fail independently [59, 91].

The coarsest granularity for isolation is the entire computer. Many commercial fault-tolerant systems, such as Stratus Integrity systems [113] and Tandem NonStop systems [19, 22], execute each program on multiple processors. If one processor fails, the others continue uncorrupted. There have been many research systems that take the same approach [28, 11, 33, 177]. The primary benefit of isolating at the machine level is tolerance of hardware failures: when one machine fails, another can continue to provide service. The drawback, however, is the substantial cost and complexity of purchasing and managing multiple computers.

Virtual machine technologies [39, 43, 190, 207, 17] are an alternative solution to isolating unreliable code. They promise the software reliability benefits of multiple physical computers but without the associated costs. In addition, virtual machines reduce the amount of code that can crash the whole machine by running the operating system in user mode and only the virtual machine monitor in privileged mode. Virtualization techniques typically run several entire operating systems on top of a virtual machine, so faulty extensions in one operating system cause only that OS instance and its applications to fail. While applications can be partitioned among virtual machines to limit the scope of failure, doing so removes the benefits of sharing within an operating system, such as fast IPC and intelligent scheduling.

However, if the extension executes inside the virtual machine monitor, such as device drivers for physical devices, a driver fault causes all virtual machines *and* their applications to fail. Executing extensions in their own virtual machines solves this problem [83, 130, 72], but raises compatibility problems. All communication between applications and drivers must then transit an IPC channel. If the contents of driver requests are unknown, they cannot be passed automatically between virtual machines.

Within a single operating system, user-mode processes provide isolation for unreliable code [186]. Traditionally, operating systems isolate themselves from applications by executing applications in a separate user-mode address space. Several systems have moved a subset of device drivers into user-mode processes to simplify development and improve reliability [109, 117, 146]. Microkernels [210, 132, 212] and their derivatives [71, 81, 96] extend the process abstraction to apply to other kernel extensions as well. Extensions execute in separate user-mode address spaces that interact with the OS through a kernel communication service, such as remote procedure call [23]. Therefore, the failure of an extension within an address space does not necessarily crash the system. Despite much research in fast inter-process communication (IPC) [23, 132], the reliance on separate address spaces raises performance concerns that have prevented adoption in commodity systems. An attempt to execute drivers in user mode in the Fluke microkernel doubled the cost of device use [201]. In addition, there is the high price of porting or reimplementing required to move extensions into user mode. Microkernel/monolithic hybrids, such as L^{4}Linux [97], provide a transition path to microkernels by executing a monolithic kernel on top of a microkernel.

Protecting individual pages within an address space provides fine-grained isolation and efficient sharing, because data structures need not be marshaled to be passed between protection domains. Page-level protection has been used in single-address-space operating systems [40] and to isolate specific components or data from corruption, e.g., in a database [192] or in the file system cache [157].

There have been several major hardware approaches to providing sub-page memory isolation. Capability-based architectures [107, 159, 131] and ring and segment architectures [112, 174] both enable fine-grained protection. With this hardware, the OS is extended not by linking code into the kernel but by adding new privileged subsystems that exist in new domains or segments. Several projects also provide mechanisms for new extensions, written to a new programming model, to execute safely with segment-based protection [44, 185] on x86 hardware. In practice, segmented architectures have often been difficult to program and plagued by poor performance [48]. These problems have been addressed recently with the Mondrian protection model that uses segments only for protection and not for addressing [209].

To remove the reliance on new (and unavailable) or existing (and slow) hardware memory protection, software fault isolation (SFI) rewrites binary executables to isolate extension code [205]. The rewriting tool inserts checks before all memory accesses and branches to verify that the target address is valid. The system detects a failure when the extension references memory or branches outside safe regions. While SFI provides isolation without hardware support, it requires contiguous memory regions for efficient address checking, making it inappropriate for existing extensions that access widely dispersed regions of the kernel address space.

Despite decades of research on isolation, commodity operating systems still rely on user-mode processes as their only isolation mechanism. However, processes have proved too expensive to use in many situations, and reliability suffers as a result. Too often, new isolation mechanisms designed to reduce this cost do not consider recovery or backward compatibility, and hence have found little use. As a result, there is still a need for lightweight isolation mechanisms for containing accidental software failures.

Failure detection

Failure detection is a critical piece of reliable execution because it triggers recovery. Prompt failure detection allows the construction of simpler programming models, such as fail-stop processors, from complex failure modes. The difficulty of detecting failures depends on the fault model. For example, timing failures can be detected with a simple timeout mechanism, while detecting output failures may require complete knowledge of the expected output of a component.

Commonly, operating system kernels are implemented with the assumption that faults in kernel code have been avoided and need not be tolerated. Failure-detection code instead targets user-mode processes and external devices rather than kernel components. For example, system calls check their parameters for correctness to ensure that a bad parameter does not crash or compromise the kernel [186]. Network protocols rely on timeouts to detect dropped packets [168]. Some kernels, including Linux [31], also do limited exception handling in kernel mode to continue execution in the presence of non-fatal processor exceptions.

Operating systems that target high reliability add additional checks to ensure that the kernel itself is functioning. For example, the kernel may monitor progress of its components via counters and sensors [27, 202]. In a distributed system, a remote machine or independent hardware with access to the kernel's address space may do the monitoring [27], allowing remote detection of kernel failures at low latency. Failures can also be detected probabilistically based on observed deviations from past behavior [42].

When commodity software is used for applications that need high reliability, *wrappers* interposed between the commodity component and the high-reliability system can provide error detection. For example, Mafalda places wrappers around a commodity microkernel to both check the correctness of the microkernel's actions and recover after a failure [73]. The Healers system detects bad parameters passed to standard library routines to convert crash failures (which may occur when the library operates on bad inputs) into fail-stop failures [78]. Other work has focused on determining where to place wrappers to limit the scope of corruption [115]. One difficulty with wrappers is detecting invalid outputs. Generally, unless the correct output is known (via a redundant computation), then only a subset of invalid outputs can be detected. In the context of device drivers, which interact with agents outside the computer, it is impossible to detect whether data was corrupted as it passed from the device through the driver.

Failure detection accuracy fundamentally limits the availability of a system. When failures are not detected promptly, they persist longer and lower availability [42]. However, simplistic failure detectors, such as exception handlers and timers that are simple to implement and inexpensive to execute, may detect sufficiently many failures to improve the reliability of commodity operating systems.

Recovery

The goal of recovery is to restore the system to correct operation after a failure. The database community has long recognized the importance of recovery and relies on transactions to prevent data corruption and to allow applications to manage failure [90]. Recently, the need for failure recovery has moved from specialized applications to the more general arena of commodity systems and applications [164].

We can classify recovery strategies by the state of the system after recovery and by their transparency to applications and system components. A recovery strategy may move the state of a system *backwards*, to a correct state that existed before the failure, or *forwards*, to a new state [10]. Furthermore, recovery strategies can either be *concealing*, in that they hide failure and recovery from applications, or *revealing*, in that they notify applications of a failure and depend on them to participate in the recovery process.

A general approach to recovery is to run application or operating system replicas on two machines, a primary and a backup. All inputs to the primary are mirrored to the backup. After a failure of the primary, the backup machine takes over to provide service. The replication can be performed by the hardware [113], at the hardware-software interface [33], at the system call interface [11, 28, 32], or at a message passing or application interface [19]. For example, process pairs execute each application on two machines. Before processing a request, the primary forwards the request to the backup. If the primary fails to complete the request, the backup is then responsible for recovering, possibly by retrying the failed request [19]. A problem with primary-backup systems is that extensive failure detection is still required. When the backup and primary processes produce different results, it is impossible to tell which is correct. Triple-modular redundancy (TMR) systems execute three copies of the code to avoid the need for extra failure detection and recovery code [139]. If an output from one replica differs from the other two, it is assumed to be incorrect and that replica is marked as failed. Thus, no extra checks or recovery code are required, as there are still two correctly functioning processes [22].

Primary-backup systems generally take a backwards recovery approach. The backup process begins executing at a point before the error occurred, or, if no error occurred on the backup, at the exact point of failure. These systems can either conceal or reveal errors from applications.

The Tandem NonStop Advanced system, for example, can either expose failures and subsequent recovery to application using transactions [90] or transparently conceal failure and let the backup applications execute without knowledge of the failure [22].

On a single machine, transactions provide a revealing mechanism for backwards recovery. Transactions allow the state of a system to be rolled back to a point before the failure occurred [90]. Upon detecting a failure, applications explicitly abort transactions, which rolls back the system state. While most commonly used in databases and other applications with persistent state, Vino [182] and QuickSilver [176] applied transactions within the OS to improve reliability. In some cases, such as the file system, the approach worked well, while in others it proved awkward and slow [176]. The revealing strategy of transactions, though, makes them difficult to integrate into existing systems, which are unprepared for the independent failure of system components.

Another, simpler, revealing strategy is to restart a failed component and, if necessary, its clients. This strategy, termed "micro-reboots," has been proposed as a general strategy for building high-availability software because it reuses initialization code to recover from a failure rather than relying on separate recovery code [36]. In these systems, the burden of recovery rests on the clients of the restarted component, which must decide what steps to take to continue executing. In the context of operating system component failures, few existing applications do this, since these failures typically crash the system [36]. Those applications that do not may share the fate of the failed component and must be restarted as well.

A common approach to concealing failures is to checkpoint system state periodically and rollback to a checkpoint when a failure occurs [167]. This is a backwards recovery strategy, because the system returns to a previous state. Logging is often used to record state changes between checkpoints [16, 136, 149, 11, 28, 172]. These systems conceal errors from applications by transparently restarting applications from the checkpoint after a failure and then replaying logged requests. Recent work has shown that this approach is limited when recovering from application faults: applications often become corrupted before they fail; hence, their checkpoints and persistent storage may also be corrupted [38, 135].

The fundamental recovery problem for commodity operating systems is keeping applications running. Revealing strategies are unworkable, because they depend on application participation to recover completely. Similarly, forward recovery is also inappropriate, because the state of the system advances beyond what applications may have expected. As a result, improving the reliability of commodity operating systems demands a concealing backwards recovery mechanism.

Fault Tolerance Summary

Tolerating faults requires isolating faulty code, detecting failures, and recovering from a failure. Unlike highly reliable systems, today's commodity operating systems are built with the assumption that the kernel and its extensions cannot fail, and as a result lack all three elements. The challenge for existing operating systems is to find efficient and inexpensive mechanisms to provide these elements while maintaining compatibility with existing code.

2.5.3 Summary

Systems can be made reliable by either avoiding faults and the subsequent failures or by tolerating failures. The choice of strategy depends largely on the goals and properties of the system. Often, a hybrid approach is the most appropriate, for example preventing most failures with compile-time checking and tolerating the remaining ones with isolation and recovery techniques.

In safety critical applications, the code *must* produce the correct result. As a result, fault-tolerant strategies relying on retrying or executing multiple versions of a component are not sufficient if none of the versions produce a correct output. Thus, applications demanding extreme reliability, such as flight-control systems, rely on programming languages, verification tools, and strict engineering practices to avoid faults during the development process [111, 165, 26].

An important limitation of fault-avoidance techniques is that they can only verify what can be specified. For operating system extensions, this has been the interface between the kernel and the extension. Thus, static analysis tools for drivers only address the correctness of software interfaces and not the interface to the device hardware. For example, a driver may obey the kernel interface but pass bad commands to the device, causing the device to perform the wrong action. Hence, static analysis tools can replace or augment some fault-tolerance techniques, but cannot obviate the need for fault-tolerant systems as a whole.

2.6 Fault Tolerance in Nooks

My work aims to improve the reliability of commodity operating systems with as little mechanism as possible. I therefore focus on the dominant cause of failures, faulty device drivers. As these drivers are not under the control of any single entity, I take a fault-tolerance approach that allows operating systems and applications to continue executing when drivers fail. The paramount goal is transparency to drivers and applications, neither of which should be modified.

2.6.1 Nooks' Failure Model

In designing Nooks, I made several assumptions about driver failures. First, I assume that most driver failures are transient. Thus, a reasonable recovery strategy is to reload the failed driver and retry requests in progress at the time of failure, because the failure is unlikely to recur. With this mechanism, it is possible to conceal recovery, because all requests are eventually processed by the driver. Based on anecdotal evidence, I believe this to be a reasonable assumption. However, if driver failures are deterministic, concealing the recovery process from applications may not be possible because requests that trigger the failure can never be successfully completed. Recovery is still possible, but applications must participate by handing failed requests..

Second, I assume that most driver failures are crash failures, output failures, or timing failures. I further assume that these failures are fail-stop and can be detected and stopped before the kernel or other drivers are corrupted. I assume that drivers are not malicious and do not suffer from arbitrary failures. Thus, Nooks includes services to detect driver failures and prevent a failed driver from corrupting other system components. The ability of Nooks to make driver failures fail-stop depends both on the quality of isolation and on the state maintained by the device. For example, if a driver corrupts persistent device state before the failure is stopped, then the stable-storage property is violated and Nooks cannot recover. As a result, Nooks must detect failures before persistent device state is corrupted. If this is not possible, then additional recovery strategies, beyond Nooks' current abilities, may be required to repair corrupt state as part of recovery.

2.6.2 Nooks' Design

Nooks uses *lightweight kernel protection domains* to isolate drivers. These domains are in the kernel address space, allowing fast sharing of read-only data, and rely on page-level protection to support efficient protected access to non-contiguous data regions. I show that this is sufficient to isolate unmodified device drivers and other kernel extensions.

Nooks detects driver failures with simple mechanisms. The kernel exception handlers notify Nooks when a driver generates an exception, such as a protection violation fault. Nooks places wrappers around drivers to detect certain bad outputs, such as pointers to invalid addresses, and relies on timers to detect timing failures when a driver is not making progress. Finally, Nooks detects an output failure when a driver consumes too many kernel resources.

Shadow drivers, my recovery mechanism for Nooks, provides transparent backwards recovery by concealing failure from applications while resetting the driver to its state before the failure. Shadow drivers are similar to primary-backup systems in that they replicate all communication between the kernel and device driver (the primary), sending copies to the shadow driver (the backup). If the driver fails, the shadow takes over temporarily until the driver recovers. However, shadows differ from typical replication schemes in several ways. First, because my goal is to tolerate only driver failures, not hardware failures, both the shadow and the "real" driver run on the same machine. Second, and more importantly, the shadow is *not* a replica of the device driver: it implements only the services needed to manage recovery of the failed driver and to shield applications from the recovery. For this reason, the shadow can be much simpler than the driver it shadows.

Shadow drivers log and replay requests similarly to checkpoint/replay recovery systems. However, shadow drivers rely on the shared interface to drivers to log only those inputs that impact driver behavior. Compared to checkpointing the complete driver state or logging all inputs, the shadow driver approach reduces the amount of state in the log. Hence, the likelihood of storing corrupt state, which is possible with full logging or checkpointing, is also reduced.

2.7 Summary

Table 2.1 lists techniques for improving operating system reliability, the approach they take, and the changes required to use the approach. Nooks is the only fault-tolerance technique that provides

Table 2.1: Comparison of reliability techniques.

Technique	**Provides**	**Approach**	**Required Changes to Applications and Critical Components**
Software engineering	N/A	avoidance	OS, extensions, programmers
Languages	Isolation / error detection	avoidance	OS, extensions, Programmers
Static analysis	Isolation	avoidance	none
Redundant hardware	Isolation / error detection / recovery	tolerance	HW, OS
Virtual machines	Isolation	tolerance	none
OS Structure	Isolation	tolerance	OS, extensions
Capabilities / ring-segment	Isolation	tolerance	HW, OS, extensions
Mondrian Memory	Isolation	tolerance	HW, OS
Checkpoint/Log	Recovery	tolerance	none
Transactions	Isolation / recovery	tolerance	OS, extensions, programmers, applications
Micro-reboots	Recovery	tolerance	OS, applications
Wrappers	Isolation / error detection	tolerance	none
Nooks	**Isolation / error detection / recovery**	**tolerance**	**none**

isolation, failure detection, and recovery without requiring changes to hardware, extensions, applications, or programming practices. The OS requires only minor changes for integrating the new code. Nooks relies on a conventional processor architecture, a conventional programming language, a conventional operating system architecture, and existing extensions. It is designed to be transparent to the extensions themselves, to support recovery, and to impose only a modest performance penalty.

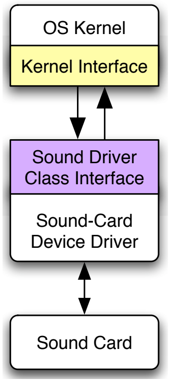

Figure 3.1: A sample device driver.

Chapter 3

DEVICE DRIVER OVERVIEW

A device driver is a kernel-mode software component that provides an interface between the OS and a hardware device.[1] The driver converts requests from the kernel into requests to the hardware. Drivers rely on two interfaces: the interface that drivers *export* to the kernel, which provides access to the device, and the kernel interface that drivers *import* from the operating system. For example, Figure 3.1 shows the kernel calling into a sound-card driver to play a tone; in response, the sound driver converts the request into a sequence of I/O instructions that direct the sound card to emit a sound.

In addition to processing I/O requests, drivers also handle configuration requests. Configuration requests can change both driver and device behavior for future I/O requests. As examples, applications may configure the bandwidth of a network card or the volume of a sound card.

In practice, most device drivers are members of a *class*, which is defined by its interface. Code that can invoke one driver in the class can invoke any driver in the class. For example, all network

[1] This thesis uses the terms “device driver” and “driver” interchangeably.

drivers obey the same kernel-driver interface, and all sound-card drivers obey the same kernel-driver interface, so no new kernel or application code is needed to invoke new drivers in these classes. This class orientation allows the OS and applications to be device-independent, as the details of a specific device are hidden from view in the driver.

In Linux, there are approximately 20 common classes of drivers. However, not all drivers fit into classes; a driver may extend the interface for a class with proprietary functions, in effect creating a new sub-class of drivers. Drivers may also define their own semantics for standard interface functions, known only to applications written specifically for the driver. In this case, the driver is in a class by itself. In practice, most common drivers, such as network, sound, and storage drivers, implement only the standard interfaces.

Device drivers are either *request-oriented* or *connection-oriented*. Request-oriented drivers, such as network drivers and block storage drivers, maintain a single hardware configuration and process each request independently. In contrast, connection-oriented drivers maintain separate hardware and software configurations for each user of the device. Furthermore, requests on a single connection may depend on past requests that changed the connection configuration.

We can model device drivers as abstract state machines; each input to the driver from the kernel or output from the driver reflects a potential state change in the driver. For example, the left side of Figure 3.2 shows a state machine for a network driver as it sends packets. The driver begins in state S0, before the driver has been loaded. Once the driver is loaded and initialized, the driver enters state S1. When the driver receives a request to send packets, it enters state S2, where there is a packet outstanding. When the driver notifies the kernel that the send is complete, it returns to state S1. The right side of Figure 3.2 shows a similar state machine for a sound-card driver. This driver may be opened, configured between multiple states, and closed. The state-machine model aids in designing and understanding a recovery process that seeks to restore the driver state by clarifying the state to which the driver is recovering. For example, a mechanism that unloads a driver after a failure returns the driver to state S0, while one that also reloads the driver returns it to state S1.

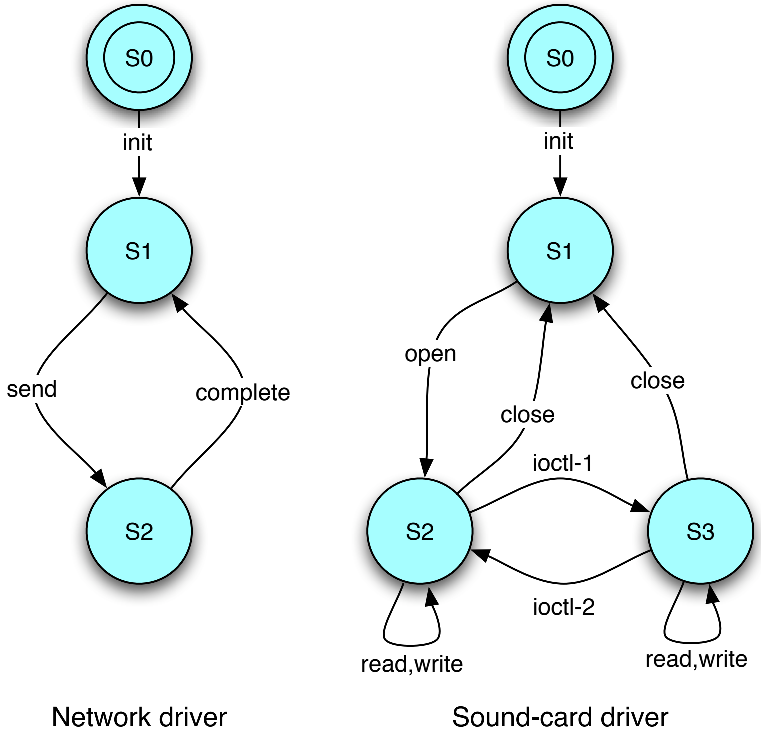

Figure 3.2: Examples of driver state machines.

Chapter 4

THE NOOKS RELIABILITY LAYER

4.1 Introduction

Device drivers are a major cause of crashes in commodity operating systems, such as Windows and Linux, because a fault in a device driver can cause the entire OS to crash. While the core operating system can be tested broadly, drivers are used by a smaller population and are frequently updated as new hardware is introduced. Thus, drivers do not achieve the same level of reliability as core operating system code. Given the vast number of existing device drivers (over 35,000 for Windows XP), it is not practical to remove bugs from driver code. Instead, the driver reliability problem demands a structural solution that severs the dependency of operating systems on driver correctness, so that a driver fault can no longer bring the system down. My work seeks to improve operating system reliability by allowing the OS to safely execute faulty device drivers. Thus, even when a driver fails, the system as a whole remains functional.

To this end, I designed and implemented a system called *Nooks* that isolates device drivers and then recovers when they fail. Isolation ensures that failed drivers are contained and do not cause other drivers or the OS itself to fail. Recovery ensures that the system is restored to a functioning state after a failure. I designed Nooks to be practical, in that supports existing drivers and operating systems at minimal performance overhead.

Nooks is based on two core design principles:

1. *Tolerate with best effort.* The system must prevent and recover from most, but not necessarily all, driver failures.
2. *Tolerate mistakes, not abuse.* Drivers are generally well behaved but may fail due to errors in design or implementation.

From the first principle, I am not seeking a complete solution for all possible driver failures. However, since drivers cause the majority of system failures, tolerating most driver failures will substantially improve system reliability. From the second principle, I have chosen to occupy the design space somewhere between "unprotected" and "safe." That is, the driver architecture for conventional operating systems (such as Linux or Windows) is unprotected: nearly any bug within the driver can corrupt or crash the rest of the system. In contrast, safe systems (such as SPIN [24] or the Java Virtual Machine [88]) strictly limit extension behavior and thus make no distinction between buggy and malicious code. I trust drivers not to be malicious, but I do not trust them not to be buggy.

The practical impact of these principles is substantial, both positively and negatively. On the positive side, it allows me to define an architecture that directly supports existing driver code with only moderate performance costs. On the negative side, my solution does not detect or recover from 100% of all possible failures and can be circumvented easily by malicious code acting within the kernel. As examples, consider a malfunctioning driver that does not corrupt kernel data, but returns a packet that is one byte short, or a malicious driver that explicitly corrupts the system page table.

Similarly, Nooks cannot safely execute 100% of drivers, because it cannot always distinguish between correct and faulty behavior. For example, a driver that dereferences and writes to a hard-coded address containing a kernel data structure may be correct, or it may be accidentally corrupting the kernel. Nooks cannot tell which is the case. On both dimensions, detecting faulty code and executing correct code, I strive to maximize Nooks' abilities while minimizing development and run-time cost.

Among failures that can crash the system, a spectrum of possible defensive approaches exists. These range from the Windows approach (i.e., to preemptively crash to avoid data corruption) to the full virtual machine approach (i.e., to virtualize the entire architecture and provide total isolation). My approach lies in the middle. Like all possible approaches, it reflects trade-offs among performance, compatibility, complexity, and completeness. Section 4.3.7 describes Nooks' current limitations. Some are a result of the current hardware or software implementation while others are architectural. These limitations have two effects: the inability to tolerate certain kinds of faulty behavior and the inability to safely execute certain kinds of correct driver code. Despite these limitations, given the tens of thousands of existing drivers, and the millions of failures they cause, a best-effort solution has great practical value.

At a high level, Nooks is a layer between the operating system kernel and drivers. As shown in Figure 4.1, the layer is composed of four core services and a controller. In the remainder of this chapter, I discuss the architecture and implementation of the Nooks reliability layer.

4.2 Nooks Architecture

The Nooks architecture seeks to achieve three major goals:

1. *Isolation*. The architecture must isolate the kernel from driver failures. Consequently, it must detect these failures before they infect other parts of the kernel.
2. *Recovery*. The architecture must enable automatic recovery to permit applications that depend on a failed driver to run after a failure.
3. *Backward Compatibility*. The architecture must apply to existing systems and existing drivers, with minimal changes to either.

Achieving all three goals in an existing operating system is challenging. In particular, the need for backward compatibility rules out certain otherwise appealing technologies, such as type-safe languages and capability-based hardware. Furthermore, backward compatibility implies that the performance of a system using Nooks should not be significantly worse than a system without it.

I achieve the preceding goals by creating a new operating system *reliability layer* that sits between drivers and the OS kernel. The reliability layer intercepts all interactions between drivers and the kernel to provide isolation and recovery. A crucial property of this layer is *transparency*, i.e., to be backward compatible, it must be largely invisible to existing components. Furthermore, while designed for device drivers, it also applies to other kernel extensions, such as network protocols or file systems.

Figure 4.1 shows this new layer, which I call the *Nooks Reliability Layer* (NRL). Above the NRL is the operating system kernel. While the NRL is pictured as a layer, it may require a small set of modifications to kernel code to be integrated into a particular OS. These modifications need only be made once. Underneath the NRL is the set of isolated drivers. In general, *no* modifications should be required to drivers, since transparency for existing drivers is my major objective.

To support the goals outlined above, the Nooks Reliability Layer provides five major architectural functions: isolation, interposition, object tracking, recovery and control. Isolation ensures that faulty drivers do not corrupt the kernel. Interposition provides backward compatibility for existing

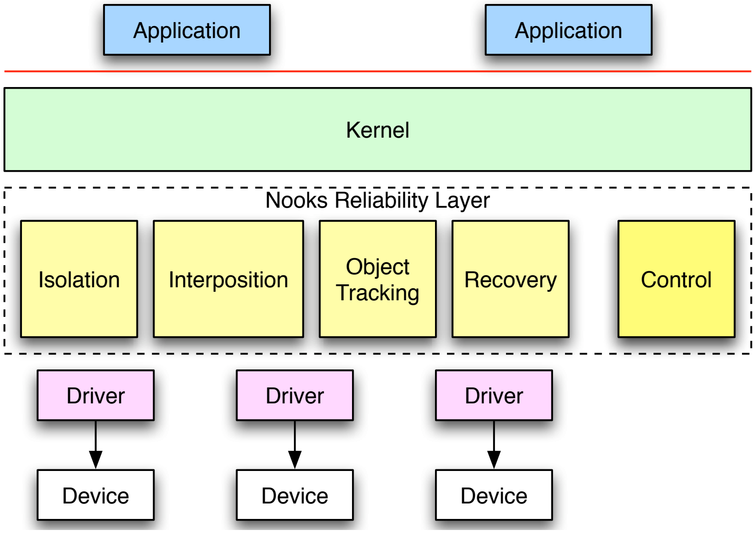

Figure 4.1: The architecture of Nooks.

drivers by injecting code at the existing kernel-driver interface. Object tracking supports recovery, by reclaiming those kernel objects in use by a driver after a failure. Recovery ensures that the system keeps functioning after a driver failure, and control manages the entire layer. I now describe these functions in more detail.

4.2.1 Isolation

Nooks' *isolation mechanisms* prevent driver failures from damaging the kernel (or other isolated drivers). Every driver in Nooks executes within its own *lightweight kernel protection domain*. This domain is an execution context with the same processor privilege as the kernel but with write access to only a limited portion of the kernel's address space.

The major task of the isolation mechanism is protection-domain management. This involves the creation, manipulation, and maintenance of lightweight protection domains. The secondary task is inter-domain control transfer. Isolation services support the control flow in both directions between driver domains and the kernel domain.

Unlike system calls, which are always initiated by an application, the kernel frequently calls into drivers. These calls may generate callbacks into the kernel, which may then generate a call into the driver, and so on. To handle this complex communication, I created a new kernel service, called *Extension Procedure Call* (XPC), that is a control transfer mechanism specifically tailored to isolating extensions within the kernel. This mechanism resembles Lightweight Remote Procedure Call (LRPC) [23] and Protected Procedure Call (PPC) in capability systems [60]. However, LRPC and PPC handle control and data transfer between mutually distrustful peers. XPC occurs between trusted domains but is asymmetric (i.e., the kernel has more rights to the driver's domain than vice versa).

4.2.2 Interposition

Nooks' *interposition mechanisms* transparently integrate existing drivers into the Nooks environment. Interposition code ensures that: (1) all driver-to-kernel and kernel-to-driver control flow occurs through the XPC mechanism, and (2) all data transfer between the kernel and driver is managed by Nooks' object-tracking code (described below).

The interface between the driver, the reliability layer, and the kernel is provided by a set of *wrapper stubs* that are part of the interposition mechanism. Nooks' stubs provide transparent control and data transfer between the kernel domain and driver domains. Thus, from the driver's viewpoint, the stubs appear to be the kernel's extension API. From the kernel's point of view, the stubs appear to be the driver's function entry points. Wrappers resemble the stubs in RPC systems [25] that provide transparent control and data transfer across address space boundaries.

4.2.3 Object Tracking

Nooks' *object-tracking functions* oversee all kernel resources used by drivers. In particular, object-tracking code: (1) maintains a list of kernel data structures a driver may manipulate, (2) controls all modifications to those structures, and (3) provides object information for cleanup when a driver fails. Protection domains prevent drivers from directly modifying kernel data structures. Therefore, object-tracking code must copy kernel objects into a driver domain so they can be modified and then copy them back after changes have been applied. When possible, object-tracking code verifies the type and accessibility of each parameter that passes between the driver and kernel. Kernel routines can then avoid scrutinizing parameters, because wrappers execute checks on behalf of unreliable drivers.

4.2.4 Recovery

Nooks' *recovery functions* detect and initiate recovery from a variety of driver failures. Nooks detects a *software failure* when a driver invokes a kernel service improperly (e.g., with invalid arguments) or when a driver consumes too many resources. Nooks detects a *hardware failure* when the processor raises an exception during driver execution, e.g., when a driver attempts to read unmapped memory or to write memory outside of its protection domain. A user or a program may also detect faulty behavior and trigger recovery explicitly.

Unmodified drivers cannot handle their own failure, so Nooks triggers a *recovery manager* to restore the system to a functioning state. Recovery managers, which may be specific to a driver, provide an extensible mechanism for restoring system operation following a failure. Recovery managers are assisted by the isolation and object-tracking services. These services provide functions for cleanly releasing resources in use by a driver at the time of failure.

4.2.5 Control

Nooks' *control functions* are invoked to change the state of a protection domain. Nooks supports four major control operations: create a protection domain, attach a driver to a protection domain when the driver is loaded, notify a protection domain of a failure, and destroy a protection domain.

These functions may be accessed externally, through tools such as the system module loader, or internally, when invoked by Nooks to recover after a failure.

4.2.6 Summary

The Nooks Reliability Layer improves operating system reliability by isolating and recovering from the failure of existing, unmodified device drivers. The layer is composed of a controller and four core services: isolation, interposition, object tracking, and recovery. These services prevent device drivers from corrupting the kernel, detect driver failures, and restore system functionality after a failure.

4.3 Implementation

I implemented Nooks within the Linux 2.4.18 kernel on the Intel x86 architecture. I chose Linux as a platform because of its popularity and its wide support for kernel extensions in the form of loadable modules. Linux proved to be a challenging platform, because the kernel provides over 700 functions callable by extensions and more than 650 extension-entry functions callable by the kernel. Moreover, few data types are abstracted, and drivers directly access fields in many kernel data structures. Despite these challenges, I brought the system from concept to function in about 18 months. The implementation supports both device drivers and other kernel extensions.

The Linux kernel supports standard interfaces for many extension classes. For example, there is a generic interface for block, character, and network device drivers, and another one for file systems. The interfaces are implemented as C language structures containing a set of function pointers.

Most interactions between the kernel and drivers take place through function calls, either from the kernel into drivers or from drivers into exported kernel routines. Drivers directly access several global data structures, such as the current task structure and the `jiffies` time counter. Fortunately, they modify few of these structures, and frequently do so through preprocessor macros and inline functions. As a result, Nooks can interpose on most kernel-driver interactions by intercepting the function calls between the driver and kernel.

Figure 4.2 shows the Nooks layer in Linux. Under the Nooks controller and object tracker are isolated kernel extensions: a single device driver, three cooperating drivers, and a kernel service. These extensions are *wrapped* by Nooks' wrapper stubs, as indicated by the shaded boxes surrounding them. Each wrapped box, containing one or more extensions, represents a single Nooks protection domain. Shown at the top, in user mode, is a recovery manager, which is responsible for restarting extensions after a failure. Figure 4.2 also shows unwrapped drivers and kernel services that continue to interface directly to the Linux kernel.

To facilitate portability, Nooks does not use the Intel x86 protection rings or memory segmentation mechanisms. Instead, drivers execute at the same privilege level as the rest of the kernel (ring 0). A conventional page table architecture provides memory protection, which can be implemented both with hardware- and software-filled TLBs.

Table 4.1 shows the size of the Nooks implementation. As discussed in Chapter 6, this code can

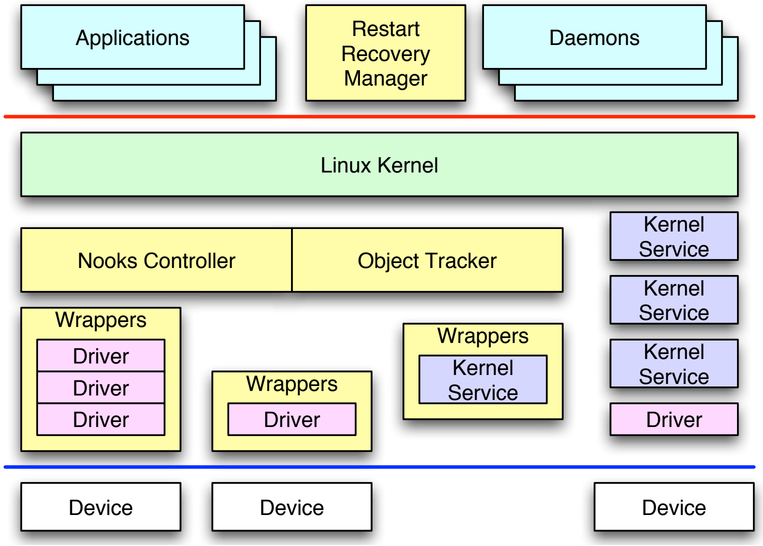

Figure 4.2: The Nooks Reliabilty Layer inside the Linux OS.

Table 4.1: The number of non-comment lines of source code in Nooks.

Source Components	**# Lines**
Domain Management	2,391
Object Tracking	1,498
Extension Procedure Call	928
Wrappers	14,484
Recovery	1,849
Build tools	1,762
Linux Kernel Changes	924
Miscellaneous	1,629
Total number of lines of code	25,465

tolerate the failure of fourteen device drivers and two other kernel extensions. The Nooks reliability layer comprises less than 26,000 lines of code. In contrast, the kernel itself has 2.4 million lines, and the Linux 2.4 distribution has about 30 million [206]. Other commodity systems are of similar size. For example, Microsoft Windows Server 2003 operating system contains over 50 million lines of code [193]. Clearly, relative to a base kernel, its drivers, and other extensions, the Nooks reliability layer introduces only a modest amount of additional system complexity.

In the following subsections I discuss the implementation of Nooks' major components: isola-

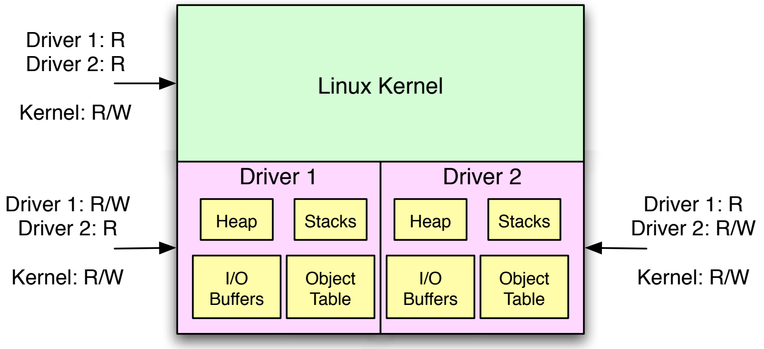

Figure 4.3: Protection of the kernel address space.

tion, interposition, wrappers, object tracking, and recovery. I describe wrappers separately because they comprise the bulk of Nooks' code and complexity. Finally, I describe limitations of the Nooks implementation.

4.3.1 Isolation

The isolation component of Nooks consists of two parts: (1) *memory management* to implement lightweight kernel protection domains with virtual memory protection, and (2) *Extension Procedure Call* (XPC) to transfer control safely between extensions and the kernel.

Figure 4.3 shows the Linux kernel with two lightweight kernel protection domains, each containing a single driver. All components exist in the kernel's address space. However, memory access rights differ for each component: e.g., the kernel has read-write access to the entire address space, while each driver is restricted to read-only kernel access and read-write access to its local domain. This is similar to the management of address space in single-address-space operating systems, where all applications share the virtual-to-physical mapping of addresses but differ in their access rights to memory [40].

To provide drivers with read access to the kernel, Nooks' memory management code maintains a synchronized copy of the kernel page table for each domain. Each lightweight protection domain has private structures, including a domain-local heap, a pool of stacks for use by the driver, memory-mapped physical I/O regions, and kernel memory buffers, such as socket buffers or I/O blocks, that are currently in use by the driver.

Nooks uses the extension procedure call (XPC) mechanism to transfer control between driver and kernel domains. The wrapper mechanism, described in Section 4.3.3, makes the XPC mechanism invisible to both the kernel and drivers, which continue to interact through their original procedural interfaces.

Two functions internal to Nooks manage XPC control transfer: (1) `nooks_driver_call` transfers from the kernel into a driver, and (2) `nooks_kernel_call` transfers from drivers into the kernel. These functions take a function pointer, an argument list, and a protection domain. They execute the function with its arguments in the specified domain. The transfer routines save the caller's context on the stack, find a stack for the calling domain (which may be newly allocated or reused when calls are nested), change page tables to the target domain, and then call the function. XPC performs the reverse operations when the call returns.

Changing protection domains requires a change of page tables. The Intel x86 architecture flushes the TLB on such a change, hence, there is a substantial cost to entering a lightweight protection domain, both from the flush and from subsequent TLB misses. This cost could be mitigated in a processor architecture with a tagged TLB, such as the MIPS or Alpha, or with single-address-space protection support [123], such as the IA-64 or PA-RISC. However, because Nooks' lightweight protection domains execute on kernel threads that share the kernel address space, they reduce the costs of scheduling and data copying on a domain change when compared to normal cross-address space or kernel-user RPCs.

To reduce the performance cost of XPC, Nooks supports *deferred calls*, which batch many calls into a single domain crossing. Nooks can defer function calls that have no visible side effects to the call. Wrappers queue deferred function calls for later execution, either at the entry or exit of a future XPC. Each domain maintains two queues: a driver-domain queue holds delayed kernel calls, and a kernel-domain queue holds delayed driver calls. As an example, I changed the packet-delivery routine used by the network driver to batch the transfer of message packets from the driver to the kernel. When a packet arrives, the driver calls a wrapper to pass the packet to the kernel. The wrapper queues a deferred XPC to deliver the packet after the driver completes interrupt processing.

In addition to deferring calls for performance reasons, Nooks also uses deferred XPC to synchronize some objects shared by the kernel and drivers. In Linux, the kernel often provides a pointer that the driver may modify with no explicit synchronization. The Linux kernel is non-preemptive and assumes, safely, that the modification is atomic and that the driver will update it "in time." In such cases, the wrapper queues a deferred function call to copy the modified object back when the driver returns control to the kernel.

I previously noted that Nooks protects against bugs but not against malicious code. Lightweight protection domains reflect this design. For example, Nooks prevents a driver from writing kernel memory, but it does not prevent a malicious driver from replacing the domain-local page table explicitly by reloading the hardware page table base register.

Furthermore, Nooks currently does not protect the kernel from DMA by a device into the kernel address space. Preventing a rogue DMA requires hardware that is not generally present on x86 computers. However, Nooks tracks the set of pages writable by a driver and could use this information to restrict DMA on a machine with the suitable hardware support.

I made several one-time changes to the Linux kernel to support isolation. First, to maintain coherency between the kernel and driver page tables, I inserted code wherever the Linux kernel modifies the kernel page table. Second, I modified the kernel exception handlers to detect exceptions that occur within Nooks' protection domains. This new code swaps in the kernel's stack pointer and page table pointer for the task. On returning from an exception, the code restores the stack pointer and page table for the driver. Finally, because Linux co-locates the task structure on the kernel stack (which XPC replaces with a driver stack), I had to change the mechanism for locating the current task structure. I currently use a global variable to hold the task pointer, which is sufficient for uniprocessor systems. On a multiprocessor, I would follow the Windows practice of storing the address in an otherwise unused x86 segment register.

4.3.2 Interposition

Interposition allows Nooks to intercept and control communication between drivers and the kernel. Nooks interposes on kernel-driver control transfers with *wrapper stubs*. Wrappers provide transparency by preserving existing kernel-driver procedure-call interfaces while enabling the protection of all control and data transfers in both directions. Control interposition required two changes to Linux kernel code. First, I modified the standard module loader to bind drivers to wrappers instead of kernel functions when loading a driver. As a side benefit, the module loader need not bind drivers

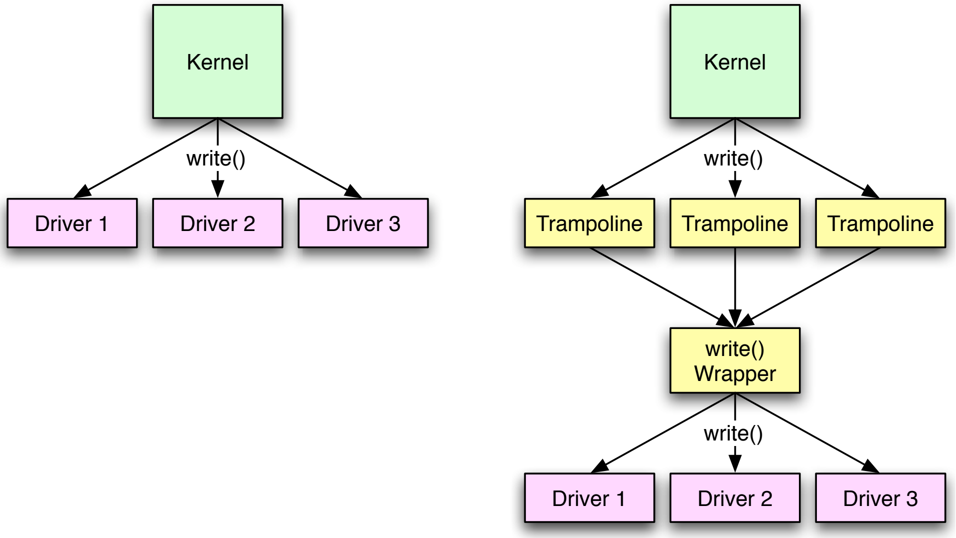

Figure 4.4: Trampoline wrappers for invoking multiple implementations of an interface function.

against wrappers. Thus, an administrator may choose whether or not to isolate each driver. This feature allows Nooks to isolate failure-prone drivers while leaving reliable ones unaffected.

Second, I modified the kernel's module initialization code to explicitly invoke Nooks on the initialization call into a driver, enabling the driver to execute within its lightweight protection domain. Following initialization, wrappers replace all function pointers passed from the driver to the kernel with pointers to wrappers instead. This causes the kernel to call wrapper functions instead of driver functions directly.

Nooks cannot interpose on all functions exported by drivers with a single wrapper stub because the kernel selects the driver it is invoking by the address of function it calls. The left side of Figure 4.4 shows the kernel selecting among multiple instances of a driver function based on the function address. Thus, a unique function pointer is required for each implementation of a function in the kernel-driver interface. As there is only one implementation of kernel functions, a single wrapper for each function suffices. Functions exported by drivers, though, may be implemented multiple times in different drivers or even within the same driver. As a result, a separate function pointer is required for each *instance* of a function in the driver interface.

To address this problem, Nooks generates "trampoline" code at run-time that pushes a pointer

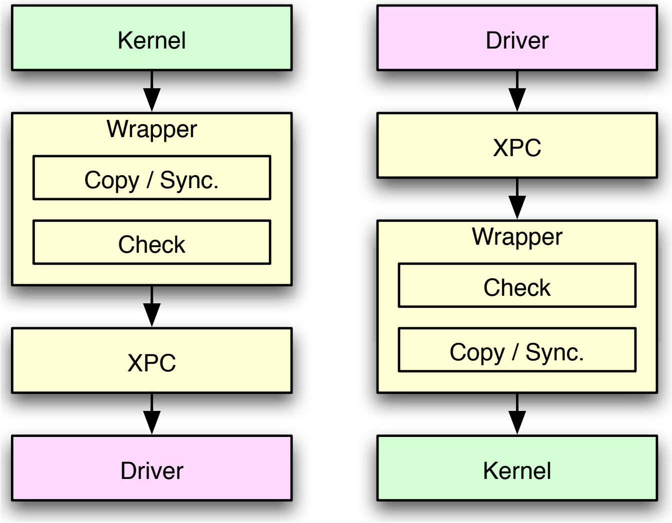

Figure 4.5: Control flow of driver and kernel wrappers.

to the driver instance on the stack and then invokes a wrapper that is common to all instances of the function. Nooks passes the kernel a pointer to the trampoline instead of the common wrapper function. The right side of Figure 4.4 shows the kernel invoking trampolines, which pass control to a common wrapper function. This mechanism ensures that all instances of a driver function have a unique function pointer and that the wrapper logic is centralized in a single implementation.

In addition to interposing on control transfers, Nooks must interpose on some data references. Drivers are linked directly to global kernel variables that they read but do not write (e.g., the current time). For global variables that drivers modify, I create a shadow copy of the kernel data structure within the driver's domain that is synchronized to the kernel's version. For example, Nooks uses this technique for the `softnet_data` structure, which contains a queue of the packets sent and received by a network driver. The object tracker synchronizes the contents of the kernel and driver version of this structure before and after XPCs into a network driver.

4.3.3 Wrappers

As noted above, Nooks inserts wrapper stubs between kernel and driver functions. There are two types of wrappers: *kernel wrappers*, which are called by drivers to execute kernel-supplied functions; *driver wrappers*, which are called by the kernel to execute driver-supplied functions. In both cases, a wrapper functions as an XPC stub that appears to the caller as if it were the target procedure in the called domain.

Both wrapper types perform the body of their work within the kernel's protection domain. Therefore, the domain change occurs at a different point depending on the direction of transfer, as shown in Figure 4.5. When a driver calls a kernel wrapper, the wrapper performs an XPC on entry so that the body of the wrapper (i.e., object checking, copying, etc.) can execute in the kernel's domain. Once the wrapper's work is done, it calls the target kernel function directly with a regular procedure call. In the opposite direction, when the kernel calls a driver wrapper, the wrapper executes within the kernel's domain. When it is done, the wrapper performs an XPC to transfer to the target function within the driver.

Wrappers perform three basic tasks. First, they check parameters for validity by verifying with the object tracker and memory manager that pointers are valid. Second, they implement *call-by-value-result* semantics for XPC, by creating a copy of kernel objects on the local heap or stack within the driver's protection domain. These semantics ensure that updates to kernel objects are transactional, because they are only applied after the driver completes, when the wrappers copy the results back to the kernel. Third, wrappers perform an XPC into the kernel or driver to execute the desired function, as shown in Figure 4.5.

While wrappers must copy data between protection domains, no marshaling or unmarshaling is necessary, because the driver and kernel share the same address space. Instead, wrappers may directly allocate and reference memory in either the kernel or the driver protection domains. The code for synchronizing simple objects is placed directly in the wrappers, while the object tracker provides synchronization routines for complex objects with many pointers. As an optimization, wrappers may pass parameters that are only read but not written by drivers without modification, as any attempt to modify the parameter will cause a memory access fault.

To improve performance, the wrappers rely on several techniques for moving complex objects between protection domains. In some cases, Nooks copies objects into the driver's protection domain, following embedded pointers as appropriate. From an analysis of driver and extension source code, I determined that it is generally unnecessary to copy the complete transitive closure of an object; while drivers read pointers more than one level removed from a parameter, they generally do not write to them. In other cases, Nooks avoids copying entirely by changing the protection on the page containing an object. A "page tracker" mechanism within the object tracker remembers the state of these mapped pages and grants and revokes driver access to the pages. Nooks uses this mechanism to avoid copying network packets and disk blocks.

Opaque pointers, which appear as `void *` in type declarations, pose a challenge for isolation. The contents of these pointers cannot be copied between protection domains because their type is not known. In practice, though, opaque pointers can often be interpreted based on how they are used. There are three classes of opaque pointers:

1. *Private to driver.* These pointers, commonly used as parameters to callback functions, are never dereferenced by any code other than the driver.
2. *Shared with applications.* These pointers are dereferenced by user-mode applications, and hence must point into user address space.
3. *Shared with other drivers.* These pointers are opaque to the kernel but may be dereferenced by other drivers.

Any pointer dereferenced by the kernel is not truly opaque, because Nooks can reuse the kernel's logic to determine its type.

The first class of opaque pointers, private to driver, may be passed through wrappers and the object tracker without any work because the kernel never dereferences the pointer. Pointers in the second class, shared with applications, reference objects from user space. Hence, drivers cannot directly dereference these pointers and must instead use accessor functions, such as `copy_from_user`. Nooks interposes on these function to grant drivers access to user-mode memory.

The third class of opaque pointers, shared with other drivers, poses a problem because drivers may attempt to modify the objects to which they point. As a result, drivers using this type of opaque pointer cannot be isolated from each other; they must all execute in the same protection domain.

Wrappers are relatively straightforward to write and integrate into the kernel. I developed a tool that automatically generates trampoline code and the skeleton of wrapper bodies from Linux kernel header files. To create the wrappers for exported kernel functions, the tool takes a list of kernel function names and generates wrappers that implement function interposition and XPC invocations.

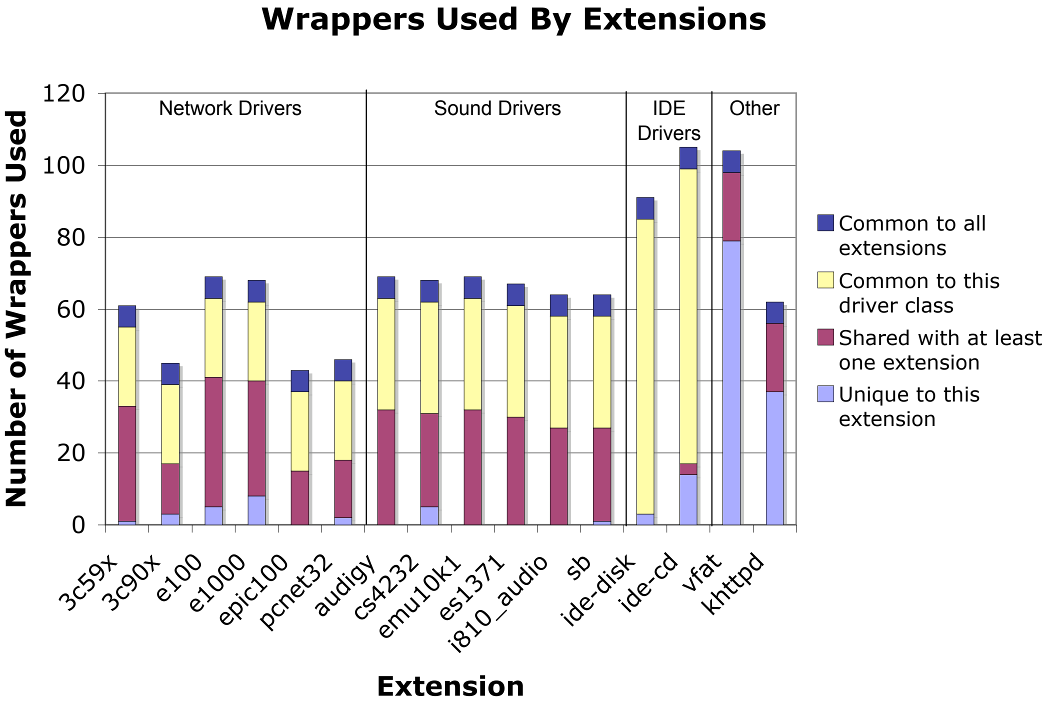

Figure 4.6: Code sharing among wrappers for different extensions.

Similarly, for the kernel-to-driver interface, the tool takes a list of interfaces (C structures containing function pointers) and generates wrappers for the kernel to call.

I wrote the body of wrapper functions manually. This is a one-time task required to support the kernel-driver interface for a specific OS. Once written, wrappers are automatically usable by all drivers that use the kernel's interface. This code checks parameters for simple errors, such as pointing to invalid addresses, and then moves parameters between protection domains.

Writing a wrapper requires knowing how drivers use a parameter: whether it is live across multiple calls to the drivers, whether it can be passed to other threads or back to the kernel, and which fields of the parameter can be modified. While I performed this analysis manually, static analysis tools could determine these properties by analyzing an existing set of drivers [69].

Wrapper Code Sharing

Previously, Table 4.1 showed that the Nooks implementation includes 14K lines of wrapper code, over half of the Nooks code base. Thus, it is important to understand the effort required to write these wrappers when isolating additional drivers or kernel extensions.

Chapter 6 describes in more detail the sixteen extensions I isolated: six sound-card drivers, six Ethernet drivers, two IDE storage drivers, a file system (*VFAT*), and an in-kernel Web server (*kHTTPd*). I only implemented wrappers for functions that were actually used by these extensions. Thus, I wrote wrappers for 126 of the 650 functions imported by the kernel and for 329 of the 700 exported functions, totaling 455 wrappers.

Figure 4.6 shows the total number of wrappers (both kernel and driver wrappers) used by each of these extensions. Each bar gives a breakdown of the number of wrappers unique to that extension and the number of wrappers shared in various ways. Sharing reduces the cost of isolating a given extension. For example, of the 46 wrappers used by the *pcnet32* Ethernet driver (35 kernel wrappers and 11 extension wrappers), 22 are shared among the six network drivers. Similarly, 31 wrappers are shared between the six sound-card drivers. Overall, of the 158 wrappers that are not shared, 116 are in the *VFAT* and *kHTTPd* extensions. While *VFAT* would share many of its wrappers with other file systems, *kHTTPd* is a one-of-a-kind extension and as a result its wrappers are unlikely to be shared. Thus, the class structure of drivers reduces the effort required to isolate a driver. Once Nooks supports a single driver in a class, little work is needed to support additional drivers.

4.3.4 Object Tracking

The object tracker facilitates the recovery of kernel objects following a driver failure. The Nooks object tracker performs two independent tasks. First, it records the *addresses* of all objects in use by a driver in a database. As an optimization, objects used only for the duration of a single XPC call are recorded on the kernel stack. Objects with long lifetimes are recorded in a per-protection-domain hash table. Second, for objects that drivers may modify, the object tracker creates and manages a driver version of the object and records an association between the kernel and driver versions. Wrappers rely on this association to map parameters between the driver's protection domain and the kernel's protection domain.

The Nooks implementation currently supports many kernel object types, such as tasklets, PCI devices, inodes, and memory pages. To determine the set of objects to track, I inspected the interfaces between the kernel and the supported drivers and noted every object type that passed through those interfaces. I then wrote object-tracking procedures for each of the 52 object types that I found. For each object type, there is a unique type identifier and code to release instances of that type during recovery. Complex types also have a routine to copy changes between a kernel and driver instance of the type.

When an object "dies" and is no longer usable by a driver, the object tracker must remove the object from its database. Determining when an object will no longer be used requires a careful examination of the kernel-driver interface. This task is possible because the kernel requires the same information to safely reclaim shared objects. For example, some objects are accessible to the driver only during the lifetime of a single XPC call from the kernel. In this case, I add the object to the tracker's database when the call begins and remove it on return. Other objects are explicitly allocated and deallocated by the driver, in which case I know their lifetimes exactly. In still other cases, I go by the semantics of the object and its use. For example, drivers allocate the `timer` data structure to register for a future callback. I add this object to the object tracker when a driver calls `add_timer` and remove it when the timer fires, at which point I know that it is no longer used. The object-tracking code is conservative, in that it may under-estimate the lifetime of an object and unnecessarily add and remove the same object from the database multiple times. It will not, however, allow a driver to access an object that the kernel has released.

In addition to tracking objects in use by drivers, the tracker must record the status of locks that are shared with the kernel. When a driver fails, Nooks releases all locks acquired by the driver to prevent the system from hanging. As a result, calls to lock kernel data structures require an XPC into the kernel to acquire the lock, synchronize the kernel and driver versions of the data structure, and record that the lock was acquired. Fortunately, the performance impact of this approach is minimal because shared locks are uncommon in the Linux device drivers I examined.

4.3.5 Recovery

The recovery code in Nooks consists of three components. First, the isolation components detect driver failures and notify the controller. Second, the object tracker and protection domains support cleanup operations that release the resources in use by a driver. This functionality is available to the third component, a *recovery manager*, whose job is to recover after a failure. The recovery manager may be customized to a specific driver or class of drivers, as I discuss more in Chapter 5.

Fault Detection

Nooks triggers recovery when it detects a failure through software checks (e.g., parameter validation or livelock detection), processor exceptions, or notification from an external source. Specifically, the wrappers, protection domains, and object tracker notify the controller of a failure when:

- The driver passes a bad parameter to the kernel, such as accessing a resource it had freed or unlocking a lock not held.
- The driver allocates too much memory, such as an amount exceeding the physical memory in the computer.
- The driver executes too frequently without an intervening clock interrupt (implying livelock).
- The driver generates an invalid processor exception, such as an illegal memory access or an invalid instruction.

In addition, it is possible to implement an external failure detector, such as a user- or kernel-mode agent, that notifies the controller of a failure. In all cases, the controller invokes the driver's recovery manager.

Recovery Managers

The recovery manager is tasked with returning the system to a functioning state. Nooks initially supported two simple recovery managers. The *default recovery manager* is a kernel service that simply unloads the failed driver, leaving the system running but without the services of the driver. The *restart recovery manager* is a user-mode agent that similarly unloads the failed driver but then executes a script to reload and restart the driver. In the next chapter, I discuss a third recovery manager that provides additional services. For each protection domain, the Nooks controller records which manager to notify when a failure occurs.

The XPC, object tracking, and protection domain code all provide interfaces to the recovery managers. The XPC service allows a manager to signal all the threads that are currently executing in the driver or have called through the driver and back into the kernel. The signal causes the threads to unwind out of the driver by returning to the point where they invoked the driver without executing any additional driver code. Threads that are in a non-interruptible state may ignore the signal, and hence complete recovery is impossible if a sleeping thread never wakes. It is not safe to wake such a thread, because the condition it is waiting for may not be true, and the thread could then corrupt data outside the driver's protection domain. By ignoring hung threads, partial recovery may still be possible. Uninterruptible sleeps are infrequent in the Linux kernel, however, so I do not believe this to be a significant limitation.

The object tracker provides an interface to recovery managers to enumerate the objects in use by a driver at the time of failure and to garbage collect the objects by releasing them to the kernel. The manager may choose both the set of objects it releases and the order in which to release them. Thus, it may preserve objects for use by the driver after recovery, such as memory-mapped I/O buffers that a hardware device continues to access.

Lightweight kernel protection domains provide similar support for recovery. The domains record the memory regions accessible to a driver and provide interfaces for enumerating the regions and for releasing the regions to the kernel.

Sequence of Recovery Actions

The Nooks recovery managers execute in stages to first disable a failed driver, then unload the driver, and optionally restart the driver. These actions are derived from the state machine model of device drivers in Chapter 3. The first two stages return the system to the initial state, S0, where the driver is not present. The final stage, restarting the driver, returns the driver to state S1, where it is initialized and ready to process requests.

Once a failure has been detected, the first step of recovery is to disable the failed driver. As Nooks isolation is imperfect, this step ensures that the driver does not cause any further corruption.

At this stage, the hardware device must be disabled to prevent it from corrupting system state as well.

The second stage of recovery is to unload the failed driver and return the system (or rather, the driver-related portions of the system) to its state before the driver was loaded. The recovery manager invokes the object tracker to release objects in use by the driver, protection domain code to release memory objects, and the XPC code to signal threads that were executing in the driver. The default recovery manager completes execution after this stage, leaving the OS running but without the services of the failed driver.

To further improve reliability, the restart recovery manager continues on to reload and re-initialize the failed driver. While applications that were actively using the old driver may experience failures when the driver is unloaded, applications not using the driver continue to execute properly and those launched after the failure may use the driver.

Complications

While conceptually simple, the practicalities of dealing with low-level device driver code pose many complications to the recovery process. The recovery manager must ensure that the kernel is returned to a functioning state and that devices managed by a failed driver do not interfere with recovery.

As much as possible, the recovery code attempts to mimic the behavior a driver that is shutting down cleanly. Hence, the object tracker uses the same routines that drivers use to release objects. For example, the object tracker calls the `kfree_skb` function to release network packets in flight, which decrements a reference count and potentially frees the packet. As a result, higher levels of software, such as network protocol code, continue to function correctly during and after recovery.

The recovery manager must ensure that the system is not left in a hung state after recovery. Two factors prevent the system from hanging when an extension fails while holding a lock. First, if the lock is shared with the kernel, the object tracker records the lock status so that a recovery manager can release the lock. Second, if the lock is private to the driver, any threads blocked on the lock will be signaled to unwind by the XPC service.

As part of the unloading process, the recovery manager must ensure that open connections to drivers and other extensions are invalidated, because the connection state is lost when the failed driver is unloaded. However, the driver's clients may still attempt to use these connections. Thus, the object tracker, when releasing connection objects, replaces the function table in the connection with a table of dummy functions. These functions return errors for all calls other than the `release` operation, which closes the connection. As a result, clients can close a connection to a failed driver but will receive errors if they try to request any other services on a connection.

In the period between the detection of a failure and the unloading of the driver, requests may continue to arrive. To ensure that driver code is not invoked after a failure, wrappers check the state of a driver before performing an XPC into a driver. If the driver has failed, the wrapper returns an error code to its caller. After the driver is unloaded, attempts to access the driver will receive an error from the kernel, because the driver no longer exists.

Device hardware may continue to access system resources even after its driver has been halted. For example, a device may continue to generate interrupts or DMA into system. Thus, the recovery manager disables interrupt processing for the device controlled by the driver during the first stage of recovery. This prevents livelock that could occur if device interrupts are not properly dismissed. The recovery manager must also delay releasing resources, such as physical memory, that a device may access until its driver has recovered and is in control of the device. For example, a network device may continue to write to packets in memory buffers; therefore, those buffers cannot be released until the driver has restarted and reinitialized the device. The default recovery manager, because it does not reload drivers, cannot reclaim these resources.

The recovery code must also take care to prevent denial-of-service when multiple drivers share a single interrupt request line. Disabling an interrupt blocks all drivers that share the line. Nooks addresses this problem with a timer that polls shared interrupts during recovery. Under normal conditions, all drivers using the shared interrupt are invoked every time any device raises the interrupt line. Each driver checks whether its device raised the interrupt, and if so, handles it appropriately. Otherwise, the driver ignores the interrupt. Nooks relies on this behavior during recovery by invoking all drivers sharing an interrupt when the timer fires. If a device *did* generate an interrupt during the timer period, its driver will handle it. Otherwise, it will ignore the call. While performance is degraded during recovery because of the extra latency induced by the timer, this approach prevents livelock due to blocking interrupts.

4.3.6 Achieving Transparency

As mentioned previously, a critical goal for Nooks is transparency, meaning that Nooks must be able to isolate existing drivers written with no knowledge of Nooks. This allows drivers to run unchanged and yet still benefit from the system's fault isolation and recovery mechanisms.

Two components of Nooks' facilitate transparency for existing drivers:

1. Nooks provides *wrapper stubs* for every function call in the kernel-driver interface.
2. Nooks provides *object-tracking code* for every object type that passes between the driver and the kernel.

The wrapper stubs interpose on every call between a driver and the kernel and invoke object-tracking code to manage every parameter passed between the caller and callee. The wrappers also invoke the XPC mechanism to transfer control from the caller's domain to the callee's domain. Thus, wrapper stubs allow isolation code to be interposed transparently between drivers and the kernel. The object-tracking code allows drivers to access kernel objects as expected without explicit code in the driver to copy data between protection domains. Thus, driver code and the kernel code invoking the driver need not be changed.

Neither the driver nor the kernel is aware of the existence of the Nooks layer. For the fourteen device drivers I isolated with Nooks, none required source code changes. Of the two other kernel extensions, only *kHTTPd*, the kernel-mode web server, required changes, and then only to 13 lines of code. With these small changes, Nooks can isolate the extension and track all its resources, allowing Nooks to catch errant behavior and to clean up during recovery. Furthermore, of the 924 lines of kernel changes mentioned in Table 4.1, none were to code that invokes drivers; they were primarily changes to memory management and interrupt handling code.

While Nooks does not require changes to driver source code, it does require that drivers be recompiled to convert inline functions and macros into XPC invocations. Because Linux does not support a binary interface to the kernel (extensions must be recompiled for every kernel), this is not a major limitation.

4.3.7 Implementation Limitations

Section 4.1 described the Nooks philosophy of designing for mistakes and for best effort. The Nooks implementation involves many trade-offs. As such, it does not provide complete isolation or fault tolerance for all possible driver failures. As previously discussed, these limitations are either faulty behavior that cannot be contained or correct behavior that cannot be recognized as such.

The faulty behavior that Nooks cannot detect stems from two factors. First, Nooks runs drivers in kernel mode to simplify backward compatibility, so Nooks cannot prevent drivers from deliberately executing privileged instructions that corrupt system state. In addition, Nooks cannot prevent infinite loops within the driver when interrupts are disabled, because it cannot regain control of the processor from the driver. Second, Nooks cannot fully verify the behavior of drivers and other kernel extensions, so Nooks' parameter checks are incomplete. For example, Nooks cannot determine that a network driver has actually sent the packet when it notifies the kernel that it has done so.

Other limitations stem from driver and extension behavior that Nooks cannot execute safely. Nooks, as an architecture, is limited to extension mechanisms with well-typed parameters. Nooks must be able to determine the type of any data passed out by the driver to the kernel or another driver in order to completely interpose on the driver's communication. While this is often the case, if it is not, Nooks cannot ensure that an isolated driver does not pass corrupt data to another driver.

Nooks' restart recovery manager restores system functionality by restarting the failed driver. As a result, it can only recover for drivers that can be killed and restarted safely. This is true for device drivers that are loaded and unloaded by the kernel's hotplug mechanism, but may not be true for all drivers or kernel extensions.

Finally, Nooks interposes on the kernel-driver interface when drivers are loaded into the kernel. Integrated drivers that are compiled into the kernel, such as real-time clock timers and interrupt controllers, require additional changes to the kernel to be initialized into a protection domain. In addition, these drivers' global variables must be placed on pages separate from kernel globals so the driver is granted write access only to its globals but not others that lie nearby in memory. While these changes increase the amount of kernel code that must change to support Nooks, they are not fundamental limitations of the Nooks architecture.

These limitations are not insignificant, and crashes may still occur. Nonetheless, the implementation will allow a kernel to resist many crashes caused by drivers. Given the enormous number of such crashes, a best-effort solution can have a large impact on overall reliability.

4.4 Summary

Device drivers are a major source of failure in modern operating systems. Nooks is a new reliability layer intended to significantly reduce driver-related failures. Nooks isolates drivers in lightweight kernel protection domains and relies on hardware and software checks to detect failures. After a failure, Nooks recovers by unloading and then reloading the failed driver. Nooks focuses on achieving *backward compatibility*, that is, it sacrifices complete isolation and fault tolerance for compatibility and transparency with existing kernels and drivers. As a result, Nooks has the potential to greatly improve the reliability of today's operating systems by removing their dependence on driver correctness.

This chapter presented the details of the Nooks isolation mechanisms. Nooks is composed of a controller and four core services: isolation, interposition, object tracking, recovery. These services allow Nooks to transparently isolate existing drivers from the kernel, detect driver failures, and recover after a driver failure. The effectiveness of Nooks depends on its abilities to isolate and recover from driver failures, which I evaluate in Chapter 6.

Chapter 5

THE SHADOW DRIVER RECOVERY MECHANISM

5.1 Introduction

The Nooks reliability layer recovers from driver failure by unloading and then reloading the failed driver. As I will show in Section 6.3, reloading failed drivers is effective at preventing system crashes. However, users of a computer are not solely interested in whether the *operating system* continues to function. Often, users care more about the *applications* with which they interact. If applications using drivers fail, then I have only partially achieved my goal of improving reliability.

With the restart recovery manager, calls into a driver that fails and subsequently recovers may return error codes because the recovery manager unloads the driver and invalidates open connections to the driver during recovery. As a result, clients of a recovered driver would themselves fail if they depend on the driver during or after recovery. For example, audio players stopped producing sound when a sound-card driver failed and recovered. For the same reason, Nooks cannot restart drivers needed by the kernel, such as disk drivers. Requests to the disk driver fail while the driver is recovering. When the Linux kernel receives multiple errors from a disk driver used for swapping, it assumes that the device is faulty and crashes the system.

In addition, any settings an application or the OS had downloaded into a driver are lost when the driver restarts. Thus, even if the application reconnects to the driver, the driver may not be able to process requests correctly.

These weaknesses highlight a fundamental problem with a recovery strategy that reveals driver failures to their clients: the clients may not be prepared to handle these failures. Rather, they are designed for the more common case that either drivers never fail, or, if they fail, the whole system fails.

To address these problems, I developed a new recovery mechanism, called a *shadow driver*, or more simply, a *shadow*. My design for shadows reflects four principles:

1. *Device driver failures should be concealed from the driver's clients.* If the operating system and applications using a driver cannot detect that it has failed, they are unlikely to fail themselves.

2. *Driver recovery logic should be generic.* Given the huge number and variety of device drivers, it is not practical to implement per-driver recovery code. Therefore, the architecture must enable a single shadow driver to handle recovery for a large number of device drivers.

3. *Recovery logic should be centralized.* The implementation of the recovery system should be simplified by consolidating recovery knowledge within a small number of components.

4. *Recovery services should have low overhead when not needed.* The recovery system should impose relatively little overhead for the common case (that is, when drivers are operating normally).

Overall, these design principles aim to protect applications and the OS from driver failure, while minimizing the cost required to make and use shadow drivers.

Shadow drivers do not apply to all types of kernel extensions or failures. First, shadow drivers only apply to device drivers that belong to a class and share a common calling interface. Second, shadow drivers recover after a failure by restarting the driver and replaying past requests and hence, can only recover from failures that are both transient and fail-stop. Deterministic failures may recur when the driver recovers, again causing a failure. Recoverable failures must be fail-stop, because shadow drivers must detect a failure in order to conceal it from the OS and applications. Hence, shadow drivers require an isolation subsystem to detect and stop failures before they are visible to applications or the operating system.

In the remainder of this chapter, I describe the architecture and implementation of the shadow driver recovery mechanism.

5.2 Shadow Driver Design

Shadow drivers are a new recovery architecture that takes advantage of the shared properties of a class of drivers for recovery. The architecture consists of three components: shadow drivers, taps, and a shadow recovery manager. I now describe these components in more detail.

5.2.1 Shadow Drivers

A *shadow driver* is a piece of software that facilitates recovery for an entire class of device drivers. A shadow driver instance is a running shadow driver that recovers for a single, specific driver. The shadow instance compensates for and recovers from a driver that has failed. When a driver fails, its shadow restores the driver to its pre-failure state. This allows, for example, the recovered driver to complete requests made before the failure.

Shadow drivers rely on the state machine model of drivers discussed in Chapter 3. Whereas the previous Nooks recovery managers seek to restore the driver to its unloaded state or initialized state, shadow drivers seek to restore drivers to their state at the time of failure.

Shadow drivers execute in one of two modes: *passive* or *active*. Passive mode is used during normal (non-faulting) operation, when the shadow driver monitors all communication between the kernel and the device driver it shadows. This monitoring is achieved via replicated procedure calls: a kernel call to a device driver function causes an automatic, identical call to the corresponding shadow driver function. Similarly, a driver call to a kernel function causes an automatic, identical call to a corresponding shadow driver function. These passive-mode calls are transparent to the device driver and the kernel and occur only to track the state of the driver as necessary for recovery. Based on the calls, the shadow tracks the state transitions of the shadowed device driver.

Active mode is used during recovery from a failure. Here, the shadow performs two functions. First, it "impersonates" the failed driver, intercepting and responding to calls for service. Therefore, the kernel and higher-level applications continue operating as though the driver had not failed. Second, the shadow driver restarts the failed driver and brings it back to its pre-failure state. While the driver restarts, the shadow impersonates the kernel to the driver, responding to its requests for service. Together, these two functions hide recovery from the driver, which is unaware that a shadow driver is restarting it after a failure, and from the kernel and applications, which continue to receive service from the shadow.

Once the driver has restarted, the active-mode shadow returns the driver to its pre-failure state. For example, the shadow re-establishes any configuration state and then replays pending requests.

A shadow driver is a “class driver,” aware of the interface to the drivers it shadows but *not* of their implementations. The class orientation has two key implications. First, a single shadow driver implementation can recover from a failure of any driver in its class, meaning that a handful of different shadow drivers can serve a large number of device drivers. As previously mentioned, Linux, for example, has only 20 driver classes. Second, implementing a shadow driver does not require a detailed understanding of the internals of the drivers it shadows. Rather, it requires only an understanding of those drivers’ interactions with the kernel. Thus, they can be implemented by kernel developers with no knowledge of device specifics and have no dependencies on individual drivers. For example, if a new network interface card and driver are inserted into a PC, the existing network shadow driver can shadow the new driver without change. Similarly, drivers can be patched or updated without requiring changes to their shadows.

5.2.2 Taps

As previously described, a shadow driver monitors communication between a functioning driver and the kernel and impersonates one to the other during failure and recovery. This is made possible by a new mechanism, called a *tap*. Conceptually, a tap is a T-junction placed between the kernel and its drivers. During a shadow’s passive-mode operation, the tap: (1) invokes the original driver, and then (2) invokes the corresponding shadow with the parameters and results of the call, as shown in Figure 5.1.

When a failure is detected in a driver, its taps switch to active mode, as shown in Figure 5.2. Here, both the kernel and the recovering device driver interact only with the shadow driver. Communication between the driver and kernel is suspended until recovery is complete and the shadow returns to passive mode, at which point the tap returns to its passive-mode state.

Taps depend on the ability to dynamically dispatch all communication between the driver and the kernel. Consequently, all communication into and out of a shadowed driver must be explicit, such as a procedure call or a message. Although most drivers operate this way, some do not. For example, kernel-mode video drivers often communicate with user-mode applications through shared memory regions [119]. Without additional support from the driver or its applications, these drivers cannot be recovered without impacting applications.

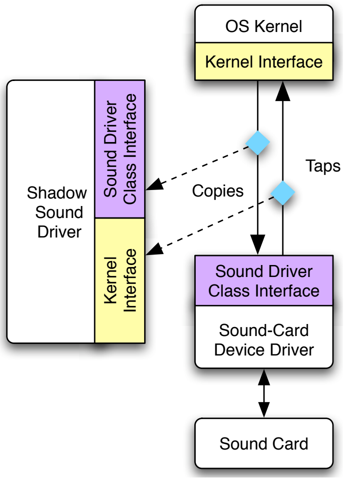

Figure 5.1: A sample shadow driver operating in passive mode.

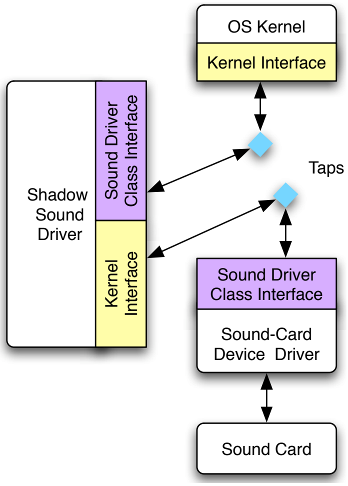

Figure 5.2: A sample shadow driver operating in active mode.

5.2.3 The Shadow Recovery Manager

The *shadow recovery manager* is responsible for coordinating recovery with shadow drivers. The Nooks controller notifies the shadow recovery manager that it has detected a failure in a driver. The shadow recovery manager then transitions the shadow driver to active mode and closes the taps. In this way, requests for the driver's services are redirected to the corresponding shadow driver. The shadow recovery manager then initiates the shadow driver's recovery sequence to restore the driver.

When recovery completes, the shadow recovery manager returns the shadow driver to passive-mode operation and re-opens its taps so that the driver can resume service.

5.2.4 Summary

The shadow driver design simplifies the development and integration of shadow drivers into existing systems. Each shadow driver is a single module written with knowledge of the behavior (interface) of a class of device drivers, allowing it to conceal a driver failure and restart the driver after a failure. A shadow driver, when passive, monitors communication between the kernel and the driver. It becomes active when a driver fails and then both proxies requests for the driver and restores the driver's state.

5.3 Implementation

This section describes the implementation of shadow drivers in the Linux operating system. I have implemented shadow drivers for three classes of device drivers: sound-card drivers, network interface drivers, and IDE storage drivers.

5.3.1 General Infrastructure

Shadow drivers rely on a reliability layer to provide three services within the kernel. An *isolation service* prevents driver errors from corrupting the kernel by stopping a driver upon detection of a failure. An *interposition service* provides the taps required for transparent shadowing and recovery. Lastly, an *object tracking service* tracks kernel resources associated with the driver to facilitate reclamation during the driver's recovery.

The Nooks controller, described in the previous chapter, handles the initial installation of shadow drivers. The controller creates a new shadow driver instance when a driver is loaded into the kernel. Because a single shadow driver services an entire class of device drivers, there may be several instances of a shadow driver executing if there is more than one driver of a class present.

Shadow drivers and Nooks together form a driver recovery subsystem, as shown in Figure 5.3. The Nooks reliability layer contributes lightweight kernel protection domains, wrappers, and the

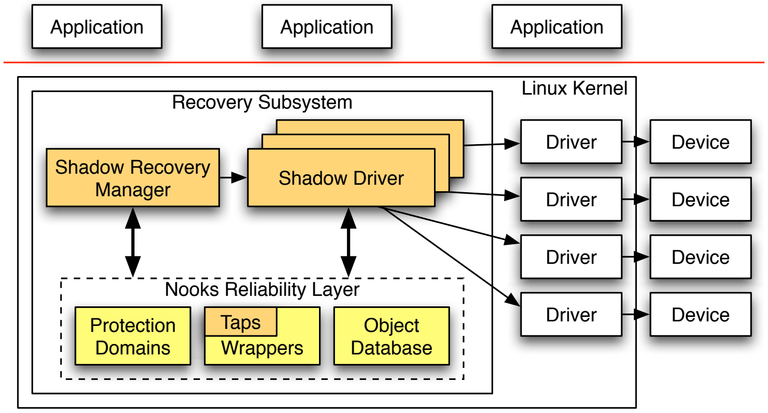

Figure 5.3: The Linux operating system with several device drivers and the shadow driver recovery subsystem.

object tracker's database. The shadow driver recovery mechanism extends Nooks by incorporating taps into the wrapper code, and by adding the shadow recovery manager and a set of shadow drivers.

5.3.2 Passive-Mode Monitoring

In passive mode, a shadow driver monitors the current state of a device driver by observing its communication with the kernel. In order to return the driver to its state at the time of failure, the shadow records the inputs to the driver in a log. These inputs are then replayed during recovery. With no knowledge of how drivers operate, the shadow would have to log all inputs to the driver. However, because the shadow is implemented with knowledge of the driver's interface, and hence its abstract state machine, not all inputs must be logged. Instead, the shadow only records inputs needed to return a driver to its state at the time of failure. The shadow drops requests that do not advance the driver's state or whose impact has been superseded by later inputs, for example transitions on a loop in the abstract state machine.

Figure 5.4 shows an example of the state machine transitions for a sound-card driver. The transitions, made when the kernel issues requests to the driver, are numbered. The final state of the sequence is S4, but there is a loop through state S3. As a result, the shadow may drop requests 2 through 5 from its log, because they do not affect the final state of the driver.

To implement this state machine, the shadow driver maintains a log in which it records several

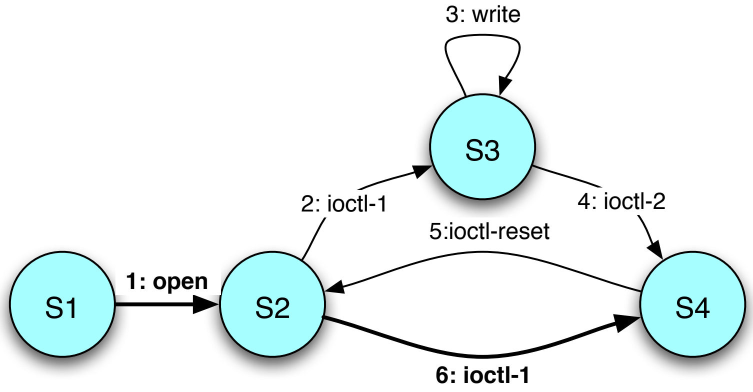

Figure 5.4: The State machine transitions for a sound-card shadow driver. Those recorded for recovery are shown in **boldface**.

types of information. First, it tracks I/O requests made to the driver, enabling pending requests to be re-submitted after recovery. An entry remains in the log until the corresponding request has been handled. In addition, for connection-oriented drivers, the shadow driver records the state of each active connection, such as offset or positioning information.

The shadow driver also records configuration and driver parameters that the kernel passes into the driver. The shadow relies on this information to reconfigure the driver to its pre-failure state during recovery. For example, the shadow sound-card driver logs `ioctl` calls (command numbers and arguments) that configure the driver.

For stateful devices, such as a hard disk, the shadow does not create a copy of the device state. Instead, a shadow driver depends on the fail-stop assumption to preserve persistent state (e.g., on disk) from corruption. In other cases, the shadow may be able to force the device's clients to recreate the state after a failure. For example, a windowing system can recreate the contents of a frame buffer by redrawing the desktop.

In many cases, passive-mode calls do no work and the shadow returns immediately to the caller. For example, the kernel maintains a queue of outstanding requests to a disk driver, and hence the shadow driver for an IDE disk does little in passive mode. For the network shadow driver, too, the Nooks object-tracking system performs much of the work to capture driver state by recording outstanding packets.

Opaque parameters can pose problems for recovery as they did for isolation. However, the class-based approach allows shadow drivers to interpret most opaque pointers. The standardized interface to drivers ensures that a client of the interface that has no knowledge of a driver's implementation can still invoke it. Hence, clients must know the real type of opaque parameters. The shadow implementer can use the same knowledge to interpret them. For example, the Open Sound System interface [197] defines opaque pointer parameters to the `ioctl` call for sound-card drivers. The shadow sound-card driver relies on this standard to interpret and log `ioctl` requests.

5.3.3 Active-Mode Recovery

The shadow enters active mode when a failure is detected in a driver. A driver typically fails by generating an illegal memory reference or passing an invalid parameter across a kernel interface. Nooks' failure detectors notice the failure and notify the controller, which in turn invokes the shadow recovery manager. This manager immediately locates the corresponding shadow driver and directs it to recover the failed driver. The shadow driver's task is to restore the driver to the state it was in at the time of failure, so it can continue processing requests as if it had never failed. The three steps of recovery are: (1) stopping the failed driver, (2) reinitializing the driver from a clean state, and (3) transferring relevant shadow driver state into the new driver. Unlike Nooks' restart recovery manager, a shadow driver does not completely unload the failed driver.

Stopping the Failed Driver

The shadow recovery manager begins recovery by informing the responsible shadow driver that a failure has occurred. It also closes the taps, isolating the kernel and driver from one another's subsequent activity during recovery. After this point, the tap redirects all kernel requests to the shadow until recovery is complete.

Informed of the failure, the shadow driver first invokes the isolation service to preempt threads executing in the failed driver. It also disables the hardware device to prevent it from interfering with the OS while not under driver control. For example, the shadow disables the driver's interrupt request line. Otherwise, the device may continuously interrupt the kernel and prevent recovery. On hardware platforms with I/O memory mapping, the shadow also removes the device's I/O mappings to prevent DMAs into kernel memory.

In preparation for recovery, the shadow garbage collects resources held by the driver. To ensure that the kernel does not see the driver "disappear" as it is restarted, the shadow retains objects that the kernel uses to request driver services. For example, the shadow does not release the `network_device` object for network device drivers. The remaining resources, not needed for recovery, are released.

Reinitializing the Driver

The shadow driver next "boots" the driver from a clean state. Normally, booting a driver requires loading the driver from disk. However, the disk driver may not be functional during recovery. Hence, the driver code and data must already be in memory before a failure occurs. For this reason, the shadow caches a copy of the device driver's initial, clean data section when the driver is first loaded. These data sections tend to be small. The driver's code is already loaded read-only in the kernel, so it can be reused from memory.

The shadow boots the driver by repeating the sequence of calls that the kernel makes to initialize a driver. For some driver classes, such as sound-card drivers, this consists of a single call into the driver's initialization routine. Other drivers, such as network interface drivers, require additional calls to connect the driver into the network stack.

As the driver restarts, the shadow reattaches the driver to the kernel resources it was using before the failure. For example, when the driver calls the kernel to register itself as a driver, the taps redirect these calls to the shadow driver, which reconnects the driver to existing kernel data structures. The shadow reuses the existing driver registration, passing it back to the driver. For requests that generate callbacks, such as a request to register the driver with the PCI subsystem, the shadow emulates the kernel and calls the driver back in the kernel's place. The shadow also provides the driver with its hardware resources, such as interrupt request lines and memory-mapped I/O regions. If the shadow had disabled these resources in the first step of recovery, the shadow re-enables them, e.g., enabling interrupt handling for the device's interrupt line. In essence, the shadow driver initializes the recovering driver by calling and responding as the kernel would when the driver starts normally.

Unfortunately, not all drivers implement the driver interface specification correctly. For example, some drivers depend on the exact timing of the calls made during initialization. While my shadow network driver was able to restart five of the six network drivers that I tested, the *e100* network driver does not fully initialize during the `open` call as the specification requires. Instead, the driver sets a timer that fires between the `open` call and the first call to send a packet. When a user-mode program loads the driver normally, there is ample delay between these two calls for the timer to fire. However, when the shadow driver invokes the driver during recovery, these calls come in rapid succession. As a result, the driver has not finished initializing when the first packet arrives. This causes the recovery process to fail because the shadow cannot replay the log of pending requests.

For testing purposes, I implemented a work-around by inserting a delay to allow any pending timers to fire after calling the `open` function. However, this example raises the larger issue of compatibility. Shadow drivers rely on driver writers to implement to the published interface and not to the specifics of the kernel implementation. The implementer of a shadow driver may choose simplicity, by only supporting those drivers that correctly implement the interface, or coverage, by including extra code to more closely mimic the kernel's behavior.

Transferring State to the New Driver

The final recovery step restores the driver to the state it was in at the time of the failure, permitting it to respond to requests as if it had never failed. Thus, any configuration that either the kernel or an application had downloaded to the driver must be restored. The shadow driver walks its log and issues requests to the driver that to restore its state.

The details of this final state transfer depend on the device driver class. Some drivers are connection oriented. For these, the state consists of the state of the connections before the failure. The shadow re-opens the connections and restores the state of each active connection with configuration calls. Other drivers are request oriented. For these, the shadow restores the state of the driver by replaying logged configuration operations and then resubmits to the driver any requests that were outstanding when the driver crashed.

As an example, to restart a sound-card driver, the shadow driver resets the driver and all its open connections back to their pre-failure state. Specifically, the shadow scans its list of open connections and calls the `open` function in the driver to reopen each connection. The shadow then walks its log of configuration commands for each connection and replays commands that set driver properties.

When replaying commands, the shadow must ensure that parameters that usually have user-mode addresses are handled correctly. Drivers normally return an error if parameters from user-mode have kernel-mode addresses. For these calls, shadow drivers reset the user-mode address limit for the current task to include the kernel address space. This ensures that the `copy_from_user` and `copy_to_user` accessor functions accept kernel addresses. Thus, the driver is unaware that the parameter is in the kernel rather than user space.

For some driver classes, the shadow cannot completely transfer its state into the driver. However, it may be possible to compensate in other, perhaps less elegant, ways. For example, a sound-card driver that is recording sound stores the number of bytes it has recorded since the last reset. After recovery, the sound-card driver initializes this counter to zero. Because the interface has no call that sets the counter value, the shadow driver must insert its “true” value into the return argument list whenever the application reads the counter to maintain the illusion that the driver has not crashed. The shadow can do this because it receives control (on its replicated call) before the kernel returns to user space.

After resetting driver and connection state, the shadow must handle requests that were either outstanding when the driver crashed or arrived while the driver was recovering. If a driver crashes after submitting a request to a device but before notifying the kernel that the request has completed, the shadow cannot know whether the device completed the request. As a result, shadow drivers cannot guarantee exactly once behavior and must rely on devices and higher levels of software to absorb duplicate requests. So, the shadow driver has two choices during recovery: restart in-progress requests and risk duplication, or cancel the request and risk lost data. For some device classes, such as disks or networks, duplication is acceptable. However, other classes, such as printers, may not tolerate duplicates. In these cases, the shadow driver cancels outstanding requests and returns an error to the kernel or application in a manner consistent with the driver interface. As I discuss when evaluating shadow drivers in Chapter 6, this is not ideal but may be better than other alternatives, such as crashing the system or the application.

After this final step, the driver has been reinitialized, linked into the kernel, reloaded with its pre-failure state, and is ready to process commands. At this point, the shadow driver notifies the shadow recovery manager, which sets the taps to restore kernel-driver communication and reestablish passive-mode monitoring.

5.3.4 Active-Mode Proxying of Kernel Requests

While a shadow driver is restoring a failed driver, it is also acting as a proxy for the driver to conceal the failure and recovery from applications and the kernel. Thus, the shadow must respond to any request for the driver's service in a way that satisfies and does not corrupt the driver's caller. The shadow's response depends on the driver's interface and the request semantics. In general, the shadow will take one of five actions:

1. Respond with information that it has recorded in its log.
2. Report that the driver is busy and that the kernel or application should try again later.
3. Suspend the request until the driver recovers.
4. Queue the request for processing after recovery.
5. Silently drop the request.

The choice of strategy depends on the caller's expectations of the driver.

Writing the proxying code requires knowledge of the kernel-driver interface, its interactions, and its requirements. For example, the kernel may require that some driver functions never block, while others always block. Some kernel requests are idempotent (e.g., many `ioctl` commands), permitting duplicate requests to be dropped, while others return different results on every call (e.g., many `read` requests). The writer of a shadow for a driver class uses these requirements to select the response strategy.

Device drivers often support the concept of being "busy." This concept allows a driver to manage the speed difference between software running on the computer and the device. For example, network drivers in Linux may reject requests and turn themselves off if packets are arriving from the kernel to quickly and their queues are full. The kernel then refrains from sending packets until the driver turns itself back on. The notion of being "busy" in a driver interface simplifies active proxying. By reporting that the device is currently busy, shadow drivers instruct the kernel or application to block calls to a driver. The shadow network driver exploits this behavior during recovery by returning a "busy" error on calls to send packets. IDE storage drivers support a similar notion when request queues are full. Sound drivers can report that their buffers are temporarily full.

Table 5.1: The proxying actions of the shadow sound-card driver.

Request	**Action**
read / write	suspend caller
interrupt	drop request
query capability ioctl	answer from log
query buffer ioctl	act busy
reset ioctl	queue for later / drop duplicate

The shadow sound-card driver uses a mix of all five strategies for proxying functions in its service interface. Table 5.1 shows the shadow's actions for common requests. The shadow suspends kernel `read` and `write` requests, which play and record sound samples, until the failed driver recovers. It processes `ioctl` calls itself, either by responding with information it captured or by logging the request to be processed later. For `ioctl` commands that are idempotent, the shadow driver silently drops duplicate requests. Finally, when applications query for buffer space, the shadow responds that buffers are full. As a result, many applications block themselves rather than blocking in the shadow driver.

5.3.5 Code Size

The preceding sections described the implementation of shadow drivers. Their class-based approach allows a single shadow driver to recover from the failure of any driver in its class. Thus, the shadow driver architecture leverages a small amount of code, in a shadow driver, to recover from the failure of a much larger body of code, the drivers in the shadow's class. However, if the shadow itself introduces significant complexity, this advantage is lost. This section examines the complexity of shadow drivers in terms of code size, which can serve as a proxy for complexity.

As I previously mentioned, I implemented shadow drivers for three classes of device drivers: sound-card drivers, network interface drivers, and IDE storage drivers. Table 5.2 shows, for each class, the size in lines of code of the shadow driver for the class. For comparison, I also show the size of the drivers that I tested for Chapter 6, and the total number and cumulative size of all Linux device drivers in the 2.4.18 kernel from each class. The total code size is an indication of the leverage gained through the shadow's class-driver structure. The table demonstrates that a shadow

Table 5.2: The size and quantity of shadows and the drivers they shadow.

Driver Class	**Shadow Driver Lines of Code**	**Device Driver Shadowed Lines of Code**	**Class Size # of Drivers**	**Class Size Lines of Code**
Sound	666	7,381 (*audigy*)	48	118,981
Network	198	13,577 (*e1000*)	190	264,500
Storage	321	5,358 (*ide-disk*)	8	29,000

driver is significantly smaller than the device driver it shadows. For example, the sound-card shadow driver is only 9% of the size of the *audigy* device driver it shadows. The IDE storage shadow is only 6% percent of the size of the Linux *ide-disk* device driver.

On top of the 26,000 lines of code in Nooks, I added about 3300 lines of new code to implement shadow drivers. I made no additional changes to the Linux kernel. Beyond the 1200 lines of code in the shadow drivers themselves, additional code consisted of approximately 600 lines of code for the shadow recovery manager, 800 lines of common code shared by all shadow drivers, and another 750 lines of code for general utilities, such as tools to automatically generate the tap code. Of the 177 taps I inserted, only 31 required actual code in a shadow; the remainder were no-ops because the calls did not significantly impact kernel or driver state.

5.3.6 Limitations

As previously described, shadow drivers have limitations. Most significant, shadow drivers rely on a standard interface to all drivers in a class. If a driver provides “enhanced” functionality with new methods, the shadow for its class will not understand the semantics of these methods. As a result, it cannot replicate the driver’s state machine and recover from its failure.

Shadow drivers rely on explicit communication between the device driver and kernel. If kernel-driver communication takes place through an ad-hoc interface, such as shared memory, the shadow driver cannot monitor it. Furthermore, shadow drivers require that all “important” driver state is visible across the kernel-driver interface. The shadow driver cannot restore internal driver state derived from device inputs. If this state impacts request processing, the shadow may be unable to restore the driver to its pre-failure state. For example, a TCP-offload engine that stores connection state cannot be recovered, because the connection state is not visible on the kernel-driver interface.

While shadow drivers are built on top of Nooks, they can be used in other contexts where Nooks' services are available. Microkernels provide much of this infrastructure and could be anther platform for shadow drivers [97]. However, the capabilities of the underlying isolation system limit the effectiveness of shadow drivers. If the isolation system cannot prevent kernel corruption, shadow drivers cannot facilitate system recovery. Shadow drivers also assume that driver failures do not cause irreversible side effects. If a corrupted driver stores persistent state (e.g., printing a bad check or writing bad data on a disk), the shadow driver will not be able to correct that action.

In addition, shadow drivers rely on the underlying isolation system to detect failures. If it does not, then shadow drivers will not recover and applications may fail. Detecting failures is difficult because drivers are complex and may respond to application requests in many ways. It may be impossible to detect a valid but incorrect return value; for example, a sound-card driver may return incorrect sound data when recording. As a result, no failure detector can detect every device driver failure. However, shadows could provide class-based failure detectors that detect violations of a driver's programming interface and reduce the number of undetected failures.

Shadow drivers can only recover from transient failures because deterministic failures may recur during recovery. While unable to recover, shadow drivers are still useful for these failures. The sequence of shadow driver recovery events creates a detailed reproduction scenario of the failure that aids diagnosis. This record contains the driver's calls into the kernel, requests to configure the driver, and I/O requests that were pending at the time of failure. This information enables a software engineer to find and fix the offending bug more efficiently.

Finally, shadow drivers may not be suitable for applications with real-time demands. During recovery, a device may be unavailable for several seconds without notifying the application of a failure. These applications, which should be written to tolerate failures, would be better served by a solution that restores the driver state but does not perform active proxying.

5.4 Summary

Improving the reliability of commodity systems demands that device driver failures be tolerated. While Nooks was designed to keep the OS running in the presence of driver failure, it did little for *applications* that depend on a driver. To improve application reliability, I designed and implemented *shadow drivers*. Shadow drivers mask device driver failures from both the OS and applications.

Shadow drivers provide an elegant mechanism that leverages the properties of device drivers for recovery. Based on an abstract state machine modeling an entire class of drivers, shadow drivers monitor the communication between the kernel and driver to obtain the driver state during normal operation. When a driver fails, the shadow relies on this state to conceal the failure by proxying for the driver. At the same time, the shadow recovers by restarting the driver, and then replaying requests to bring the driver back to its pre-failure state.

My experience shows that shadow drivers, while simple to implement and integrate, are highly leveraged: a single shadow driver can enable recovery for an entire class of device drivers. Thus, with the addition of a small amount of shadow code to an operating system, the failure of a much larger body of driver code can be tolerated. In Chapter 6 I show that shadow drivers can keep applications running in the presence of failures, while Nooks' recovery managers cannot.

Chapter 6

EVALUATION OF NOOKS AND SHADOW DRIVERS

6.1 Introduction

The thesis of my work is that Nooks and shadow drivers can significantly improve system reliability by protecting the kernel and applications from driver failures. This chapter addresses four key questions:

1. *Isolation.* Can the operating system survive driver failures and crashes?
2. *Recovery.* Can applications that use a device driver continue to run even after the driver fails?
3. *Assumptions.* How reasonable is the assumption that driver failures are fail-stop when isolated by Nooks?
4. *Performance.* What is the performance overhead of Nooks in the absence of failures, i.e., the steady-state cost?

Using synthetic fault injection experiments, I show that Nooks can isolate and automatically recover from faults in drivers. In these tests, the operating system survived 99% of driver faults that would otherwise crash Linux. Furthermore, shadow drivers were able to restart the driver while keeping applications running in 98% of failures. I also show that the majority of driver failures are fail-stop under Nooks, and that that Nooks is limited in this regard by its ability to detect failures and not by its ability to isolate drivers. Finally, I show that when running common applications, Nooks imposes a low to moderate performance overhead.

6.2 Test Methodology

6.2.1 Types of Extensions Isolated

In the experiments reported below, I used Nooks to isolate three types of extensions: device drivers, a kernel subsystem (*VFAT*), and an application-specific kernel extension (*kHTTPd*). The device drivers I chose were common network, sound-card, and IDE storage drivers, representative of three major driver interfaces (character, network, and block [52]).

A device driver's interaction with the kernel is well matched to the Nooks isolation model for many reasons. First, the kernel invokes drivers through narrow, well-defined interfaces; therefore, it is straightforward to design and implement their wrappers. Second, drivers frequently deal with blocks of opaque data, such as network packets or disk blocks, that do not require validation. Third, drivers often batch their processing to amortize interrupt overheads. When run with Nooks, batching also reduces isolation overhead.

While Nooks is designed primarily for device drivers, it can also be used to tolerate the failure of other kernel extensions. As a demonstration, I tested Nooks with two non-driver kernel extensions. First, I isolated a loadable kernel subsystem. The subsystem I chose was the optional *VFAT* file system, which is compatible with the Windows 95 FAT32 file system [144]. While drivers tend to have a small number of interfaces with relatively few functions, the file system interface is larger and more complex. *VFAT* has six distinct interfaces that together export over 35 calls; by comparison, the device drivers each have one or two interfaces with between six and thirteen functions. In addition, driver interfaces tend to pass relatively simple data structures, such as network packets and device objects, while the file system interfaces pass complex, heavily linked data structures such as inodes.

Second, I isolated an application-specific kernel extension – the *kHTTPd* Web server [200]. *kHTTPd* resides in the kernel so that it can access kernel network and file system data structures directly, avoiding otherwise expensive system calls. My experience with *kHTTPd* demonstrates that Nooks can isolate even ad-hoc and unanticipated kernel extensions.

Overall, I have isolated fourteen drivers and two other kernel extensions under Nooks, as shown in Table 6.1. I present reliability and performance results for seven of the extensions representing the three driver classes and the other extension types: *sb*, *audigy*, *pcnet32*, *e1000*, *ide-disk*, *VFAT* and *kHTTPd*. Results for the remaining drivers are consistent with those presented.

6.2.2 Test Platforms

To evaluate Nooks isolation and shadow driver recovery, I produced three OS configurations based on the Linux 2.4.18 kernel:

Table 6.1: The Linux extensions tested. Results are reported for those in **boldface**.

Class	Driver	Device
Network	***e1000***	Intel Pro/1000 Gigabit Ethernet
	pcnet32	AMD PCnet32 10/100 Ethernet
	3c59x	3COM 3c509b 10/100 Ethernet
	3c90x	3COM 3c90x series 10/100 Ethernet driver
	e100	Intel Pro/100 Ethernet
	epic100	SMC EtherPower 10/100 Ethernet
Sound	***audigy***	SoundBlaster Audigy sound card
	sb	SoundBlaster 16 sound card
	emu10k1	SoundBlaster Live! sound card
	es1371	Ensoniq sound card
	cs4232	Crystal sound card
	i810_audio	Intel 810 sound card
Storage	***ide-disk***	IDE disk
	ide-cd	IDE CD-ROM
None	***VFAT***	Win95 compatible file system
	kHTTPd	In-kernel Web server

1. *Linux-Native* is the unmodified Linux kernel.
2. *Linux-Nooks* is a version of *Linux-Native* that includes the Nooks fault isolation subsystem and the restart recovery manager. When a driver fails, this system restarts the driver but does not attempt to conceal its failure.
3. *Linux-SD* includes Nooks, the shadow driver recovery manager, and the three shadow drivers.

For isolation tests, I use *Linux-Nooks*, as it can restart both device drivers and other kernel extensions. For the recovery tests, I use *Linux-SD*.

6.2.3 Fault Injection

Ideally, I would test Nooks with a variety of real drivers long enough for many failures to occur. This is not practical, however, because of the time needed for testing. For example, with a mean time-to-failure of 1 week, it would take almost 10 machine years to observe 500 failures. Instead, I use *synthetic fault injection* to insert artificial faults into drivers. I adapted a fault injector developed

Table 6.2: The types of faults injected into extensions.

Fault Type	**Code Transformation**
Source fault	Change the source register
Destination fault	Change the destination register
Pointer fault	Change the address calculation for a memory instruction
Interface fault	Use existing value in register instead of passed parameter
Branch fault	Delete a branch instruction
Loop fault	Invert the termination condition of a loop instruction
Text fault	Flip a bit in an instruction
NOP fault	Elide an instruction

for the Rio File Cache [157] and ported it to Linux. The injector automatically changes single instructions in driver code to emulate a variety of common programming errors, such as uninitialized local variables, bad parameters, and inverted test conditions.

I inject two different classes of faults into drivers. First, I inject faults that emulate specific programming errors that are common in kernel code [191, 46]. *Source* and *destination faults* emulate assignment errors by changing the operand or destination of an instruction. *Pointer faults* emulate incorrect pointer calculations and cause memory corruption. *Interface faults* emulate bad parameters. I emulate bugs in control flow through *branch faults*, which remove a branch instruction, and by *loop faults*, which change the termination condition for a loop.

Second, I expand the range of testing by injecting random changes that do not model specific programming errors. In this category are *text faults*, in which I flip a random bit in a random instruction, and *NOP faults*, in which I delete a random instruction. Table 6.2 shows the types of faults I inject, and how the injector simulates programming errors (see [157] for a more complete description of the fault injector).

Unfortunately, there is no available information on the distribution of programming errors for real-world driver failures. Therefore, I inject the same number of faults of each type. The output of the fault injection tests is therefore a metric of *coverage*, not reliability. The tests measure the fraction of faults (failure causes) that can be detected and isolated, not the fraction of existing failures that can be tolerated.

6.3 System Survival

In this section, I evaluate Nooks' ability to isolate the kernel from extension failure. The goal of these tests is to measure the survival rate of the operating system. I consider application failures in Section 6.4.

6.3.1 Test Environment

The application-level workload consists of four programs that stress the sound-card driver, the network driver, the storage driver and *VFAT*, and *kHTTPd*. The first program plays a short MP3 file. The second performs a series of ICMP-ping and TCP streaming tests, while the third untars and compiles a number of files. The fourth program runs a Web load generator against the kernel-mode Web server.

To ensure a common platform for testing with both the device drivers and the other extensions, I use the restart recovery manager to restore extension operation after a failure. There is one exception: the tests for *ide-disk* use shadow drivers for recovery. The restart recovery manager depends on accessing the disk to reload a failed driver. Hence, when the disk driver fails, the restart manager is unable to reload it. In contrast, the shadow driver recovery manager keeps the driver image in memory and does not need to reload it off disk.

I ran most isolation experiments in the context of the VMware Virtual Machine [190]. The virtual machine allows me to perform thousands of tests remotely while quickly and easily returning the system to a clean state following each one. I spot-checked a number of the VMware trials against a base hardware configuration (i.e., no virtual machine) and discovered no anomalies. The *e1000* tests, however, were run directly on raw hardware, because VMware does not emulate the Intel Pro/1000 Gigabit Ethernet card.

To measure isolation, I conducted a series of trials in which I injected faults into extensions running under two different Linux configurations, both running the *Linux-Nooks* kernel. In the first, called "native," the Nooks isolation services were present but unused.[1] In the second, called "Nooks," the isolation services were enabled for the extension under test. For each extension, I ran

[1] To ensure that I injected *exactly* the same fault in both systems requires using the same binary module. Linux does not support binary compatibility for loadable modules across different kernel versions. Hence, I used the same kernel for both tests.

400 trials (50 of each fault type) on the native configuration. In each trial, I injected five random faults into the extension and exercised the system, observing the results. I then ran those same 400 trials, each with the same five faults, against Nooks. It is important to note that the native and Nooks configurations are identical binaries, allowing the automatic fault injector to introduce identical faults.

6.3.2 System Survival Test Results

As described above, I ran 400 fault-injection trials for each of the six measured extensions on the native and Nooks configurations. Not all fault-injection trials cause faulty behavior, e.g., bugs inserted on a rarely (or never) executed path will rarely (or never) produce an error. However, many trials do cause failures. I now examine different types of failures that occurred.

System Crashes

A system crash is the most extreme and easiest problem to detect, as the operating system panics, becomes unresponsive, or simply reboots. In an ideal world, every system crash caused by a fault-injection trial under native Linux would result in a recovery under Nooks. As previously discussed, in practice Nooks may not detect or recover from certain failures caused by very bad programmers or very bad luck.

Figure 6.1 shows the number of system crashes caused by the fault-injection experiments for each of the extensions running on native Linux and Nooks. Of the 517 crashes observed with native Linux, Nooks eliminated 512, or 99%. In the remaining five crashes, the system deadlocked when the driver went into a tight loop with interrupts disabled. Nooks does not detect this type of failure.

Figure 6.1 also illustrates a substantial difference in the number of system crashes that occur for *VFAT* and *sb* extensions under Linux, compared to *e1000*, *pcnet32*, *ide-disk* and *kHTTPd*. This difference reflects the way in which Linux responds to kernel failures. The *e1000*, *pcnet32* and *ide-disk* extensions are *interrupt oriented*, i.e., kernel-mode extension code is run as the result of an interrupt. *VFAT* and *sb* extensions are *process oriented*, i.e., kernel-mode extension code is run as the result of a system call from a user process. *kHTTPd* is process oriented but manipulates (and therefore can corrupt) interrupt-level data structures. Linux treats exceptions in interrupt-oriented

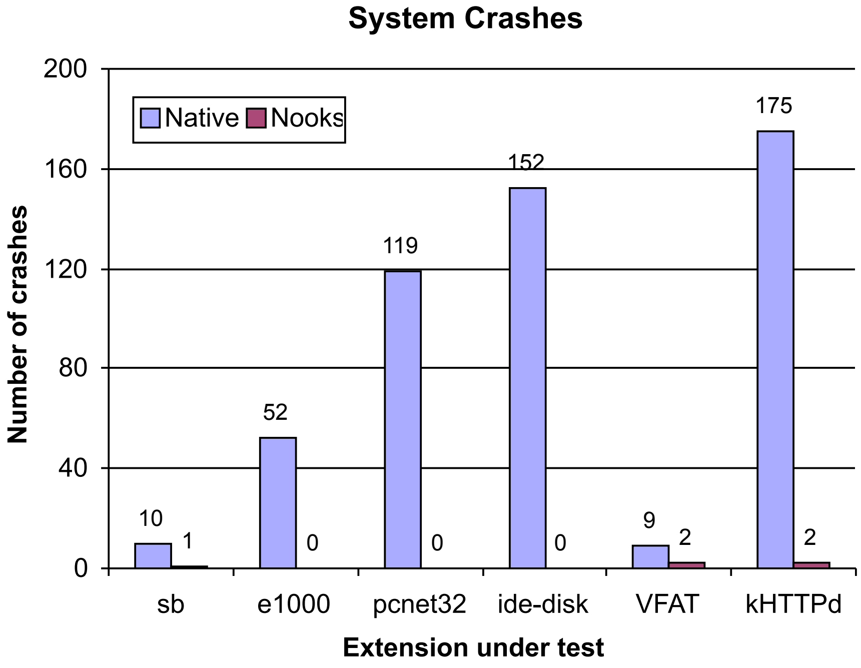

Figure 6.1: The reduction in system crashes observed using Nooks.

code as fatal and crashes the system, hence the large number of crashes in *e1000*, *pcnet32*, *ide-disk* and *kHTTPd*. Linux treats exceptions in process-oriented code as non-fatal, continuing to run the kernel but terminating the offending process even though the exception occurred *in the kernel*. This behavior is unique to Linux. Other operating systems, such as Microsoft Windows XP, deal with kernel processor exceptions more aggressively by always halting the operating system. In such systems, exceptions in *VFAT* and *sb* would cause system crashes.

Non-Fatal Extension Failures

While Nooks is designed to protect the OS from misbehaving extensions, it is not designed to detect all erroneous extension behavior. For example, the network could disappear because the device driver corrupts the device registers, or a mounted file system might simply become non-responsive due to a bug. Neither of these failures is fatal to the system in its own right, and Nooks generally does not detect such problems (nor is it intended to). However, when Nooks' simple failure detectors do detect such problems, its recovery services can safely restart the faulty extensions.

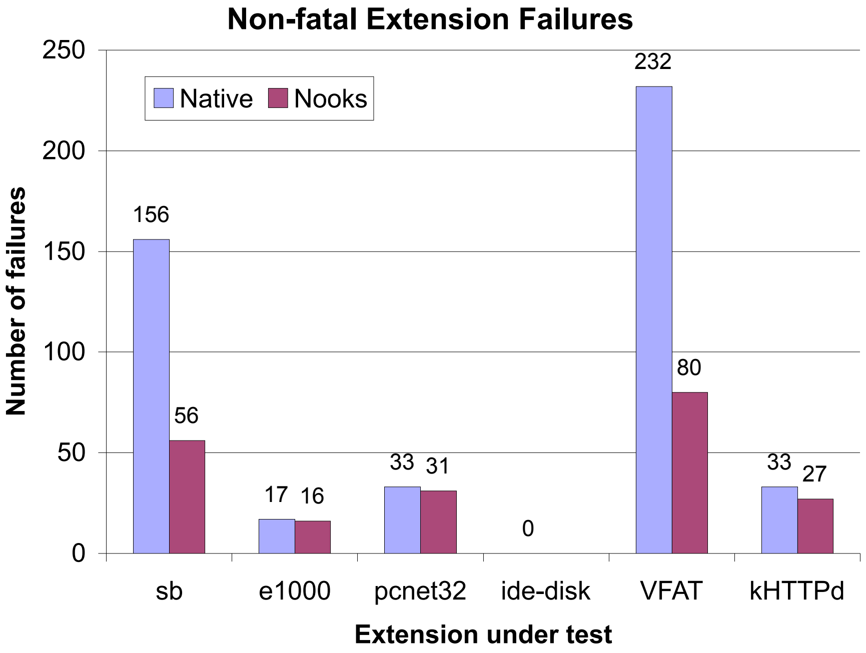

Figure 6.2: The reduction in non-fatal extension failures observed using Nooks.

The fault-injection trials cause a number of non-fatal extension failures, allowing me to examine Nooks' effectiveness in dealing with these cases, as well. Figure 6.2 shows the extent to which Nooks reduces non-fatal extension failures that occurred in native Linux. In reality, these results are simply a reflection of the Linux handling of process- and interrupt-oriented extension code, as described earlier. That is, Nooks can trap exceptions in process-oriented extensions and can recover the extensions to bring them to a clean state in many cases.

For the two interrupt-oriented Ethernet drivers (*e1000* and *pcnet32*), Nooks already eliminated all system crashes resulting from extension exceptions. The remaining non-crash failures are those that leave the device in a non-functional state, e.g., unable to send or receive packets. Nooks cannot remove these failures for *e1000* and *pcnet32*, since it cannot detect them. The few extension failures it eliminated occurred when process-oriented code was manipulating the driver.

For the *ide-disk* storage driver, there were no non-fatal extension failures. The operating system depends on the *ide-disk* to perform paging. The operating system crashes when it malfunctions, because the driver can no longer swap pages to disk. As a result, all failures of *ide-disk* are system crashes.

For *VFAT* and the *sb* sound card driver, Nooks reduced the number of non-fatal extension failures. These failures were caused by kernel exceptions in process-oriented code, which caused Linux to terminate the calling process and leave the extension in an ill-defined state. Nooks detected the processor exceptions and performed an extension recovery, thereby allowing the application to continue, but without access to the failed extension. The remaining non-fatal extension failures, which occurred under native Linux and Nooks, were serious enough to leave the extension in a non-functioning state but not serious enough to generate a processor exception that could be trapped by Nooks.

The *kHTTPd* extension is similar to the interrupt-oriented drivers because it causes corruption that leads to interrupt-level faults. However, a small number of injected faults caused exceptions within the *kHTTPd* process-oriented code. Nooks caught these exceptions and avoided an extension failure.

In general, the remaining non-fatal extension failures under Nooks were the result of deadlock or data structure corruption within the extension itself. Fortunately, such failures were localized to the extension and could usually be recovered from once discovered. In Section 6.5 I further analyze the need for better error detection that is attuned to a specific class of extensions.

Manually Injected Faults

In addition to the automatic fault-injection experiments, I manually inserted 10 bugs across the extensions. To generate realistic synthetic driver bugs, I analyzed patches posted to the Linux Kernel Mailing List [133]. I found 31 patches that contained the strings “patch,” “driver,” and “crash” or “oops” (the Linux term for a kernel fault) in their subject lines. Of the 31 patches, I identified 11 that fix transient bugs (i.e., bugs that occur occasionally or only after a long delay from the triggering test). The most common cause of failure (three instances) was a missing check for a null pointer, often with a secondary cause of missing or broken synchronization. I also found missing pointer initialization code (two instances) and bad calculations (two instances) that led to endless loops and buffer overruns.

I “broke” extensions by removing checks for NULL pointers, failing to initialize stack and heap variables properly, dereferencing a user-level pointer, and freeing a resource multiple times. Nooks automatically detected and recovered from all such failures in all extensions.

Latent Bugs

Nooks revealed several latent bugs in existing kernel extensions. For example, it uncovered a bug in the 3COM 3c90x Ethernet driver that occurs during its initialization.[2] Nooks also revealed a bug in another extension, *kHTTPd* [200], where an already freed object was referenced. In general, I found that Nooks was a useful kernel development tool that provides a "fast restart" whenever an extension under development fails.

6.3.3 Summary of System Survival Experiments

In testing its impact on operating system reliability, Nooks eliminated 99% of the system crashes that occurred with native Linux. The remaining failures directly reflect the best-efforts principle and are the cost, in terms of reliability, of an approach that imposes reliability on legacy extension and operating systems code. In addition to crashes, Nooks can isolate and recover from many non-fatal extension failures. While Nooks cannot detect many kinds of erroneous behavior, it can often trap extension exceptions and initiate recovery. Overall, Nooks eliminated nearly 60% of non-fatal extension failures in the fault-injection trials. Finally, Nooks detected and recovered from all of the commonly occurring faults that I injected manually.

6.4 Application Survival

The previous section evaluated the ability of the operating system to survive extension failures. In this section, I focus on applications. I answer the question of whether applications that use a device driver continue to run even after the driver fails and recovers. I evaluate shadow driver recovery in the presence of simple failures to show the benefits of shadow drivers compared to a system that reveals failures to applications and the OS.

[2] If the driver fails to detect the card in the system, it immediately frees a large buffer. Later, when the driver is unloaded, it zeroes this buffer. Nooks caught this bug because it write-protected the memory when it was freed.

Table 6.3: The applications used for evaluating shadow drivers.

Device Driver	**Application Activity**
Sound *(audigy driver)*	• mp3 player (*zinf*) playing 128kb/s audio • audio recorder (*audacity*) recording from microphone • speech synthesizer (*festival*) reading a text file • strategy game (*Battle of Wesnoth*)
Network	• network file transfer (*scp*) of a 1GB file • remote window manager (*vnc*) • network analyzer (*ethereal*) sniffing packets
Storage *(ide-disk driver)*	• compiler (*make/gcc*) compiling 788 C files • encoder (*LAME*) converting 90 MB file .wav to .mp3 • database (*mySQL*) processing the *Wisconsin Benchmark*

6.4.1 Application Test Methodology

Unlike the system survival experiments, the application survival experiments use a wider variety of drivers than VMware supports and were therefore run on a bare machine, a 3 GHz Pentium 4 PC with 1 GB of RAM and an 80 GB, 7200 RPM IDE disk drive. I built and tested three Linux shadow drivers for three device-driver classes: network interface controller, sound card, and IDE storage device. To ensure that my generic shadow drivers worked consistently across device driver implementations, I tested them on fourteen different Linux drivers, shown in Table 6.1.[3] Although I present detailed results for only one driver in each class (*e1000*, *audigy*, and *ide-disk*), behavior across all drivers was similar.

The crucial question for shadow drivers is whether an application can continue functioning following the failure of a device driver on which it relies. To answer this question, I tested the 10 applications shown in Table 6.3 on each of the three configurations, *Linux-Native*, *Linux-Nooks*, and *Linux-SD*.

I simulated common bugs by injecting a software fault into a device driver while an application using that driver was running. Because both *Linux-Nooks* and *Linux-SD* depend on the same isolation and failure-detection services, I differentiate their recovery capabilities by simulating failures that are easily isolated and detected. Based on the analysis of common drivers bugs in Section 6.3.2,

[3] Shadow drivers target device drivers, so I did not test the *VFAT* and *kHTTPd* kernel extensions.

Table 6.4: The observed behavior of several applications following the failure of the device drivers on which they depend.

Device Driver	Application Activity	Application Behavior		
		Linux-Native	Linux-Nooks	Linux-SD
Sound *(audigy driver)*	mp3 player	CRASH	MALFUNCTION	√
	audio recorder	CRASH	MALFUNCTION	√
	speech synthesizer	CRASH	√	√
	strategy game	CRASH	MALFUNCTION	√
Network *(e1000 driver)*	network file transfer	CRASH	√	√
	remote window manager	CRASH	√	√
	network analyzer	CRASH	MALFUNCTION	√
IDE *(ide-disk driver)*	compiler	CRASH	CRASH	√
	encoder	CRASH	CRASH	√
	database	CRASH	CRASH	√

I injected a null-pointer dereference bug into my three drivers.

6.4.2 Application Test Results

Table 6.4 shows the three application behaviors I observed. When a driver failed, each application continued to run normally (√), failed completely ("CRASH"), or continued to run but behaved abnormally ("MALFUNCTION"). In the latter case, manual intervention was typically required to reset or terminate the program.

This table demonstrates that shadow drivers (*Linux-SD*) enable applications to continue running normally even when device drivers fail. In contrast, all applications on *Linux-Native* failed when drivers failed. Most programs running on *Linux-Nooks* failed or behaved abnormally, illustrating that restart recovery protects the kernel, which is constructed to tolerate driver failures, but does not protect applications. The restart recovery manager lacks two key features of shadow drivers: (1) it does not advance the driver to its pre-fail state, and (2) it has no component to "pinch hit" for the failed driver during recovery. As a result, *Linux-Nooks* handles driver failures by returning an error to the application, leaving it to recover by itself. Unfortunately, few applications can do this.

Some applications on *Linux-Nooks* survived the driver failure but in a degraded form. For example, *mp3 player*, *audio recorder* and *strategy game* continued running, but they lost their ability to input or output sound until the user intervened. Similarly, *network analyzer*, which interfaces directly with the network device driver, lost its ability to receive packets once the driver was reloaded.

A few applications continued to function properly after driver failure on *Linux-Nooks*. One application, *speech synthesizer*, includes the code to reestablish its context within an unreliable sound-card driver. Two of the network applications survived on *Linux-Nooks* because they access the network device driver through kernel services (TCP/IP and sockets) that are themselves resilient to driver failures.

Unlike *Linux-Nooks*, *Linux-SD* can recover from disk driver failures. [4] Recovery is possible because the IDE storage shadow driver instance maintains the failing driver's initial state. During recovery the shadow copies back the driver's initial data and reuses the driver code, which is already stored read-only in the kernel. In contrast, *Linux-Nooks* illustrates the risk of circular dependencies from rebooting drivers. Following these failures, the restart recovery manager, which had unloaded the *ide-disk* driver, was then required to reload the driver off the IDE disk. The circularity could only be resolved by a system reboot. While a second (non-IDE) disk would mitigate this problem, few machines are configured this way.

In general, programs that directly depend on driver state but are unprepared to deal with its loss benefit the most from shadow drivers. In contrast, those that do not directly depend on driver state or are able to reconstruct it when necessary benefit the least. My experience suggests that few applications are as fault-tolerant as *speech synthesizer*. Were future applications to be pushed in this direction, software manufacturers would either need to develop custom recovery solutions on a per-application basis or find a general solution that could protect any application from the failure of a device driver. Cost is a barrier to the first approach. Shadow drivers are a path to the second.

6.4.3 Application Behavior During Driver Recovery

Although shadow drivers can prevent application failure, they are not "real" device drivers and do not provide complete device services. As a result, I often observed a slight timing disruption while the driver recovered. At best, output was queued in the shadow driver or the kernel. At worst, input was lost by the device. The length of the delay depends on the recovering device driver itself, which,

[4] For this reason, the *ide-disk* experiments in Section 6.3.2 use the shadow driver recovery manager.

on initialization, must first discover and then configure the hardware.

Few device drivers implement fast reconfiguration, which can lead to brief recovery delays. For example, the temporary loss of the *e1000* network device driver prevented applications from receiving packets for about five seconds.[5] Programs using files stored on the disk managed by the *ide-disk* driver stalled for about four seconds during recovery. In contrast, the normally smooth sounds produced by the *audigy* sound-card driver were interrupted by a pause of about one-tenth of one second, which sounded like a slight click in the audio stream.

Of course, the significance of these delays depends on the application. Streaming applications may become unacceptably "jittery" during recovery. Those processing input data in real-time might become lossy. Others may simply run a few seconds longer in response to a disk that appears to be operating more sluggishly than usual. In any event, a short delay during recovery is best considered in light of the alternative: application and system failure.

6.4.4 Summary of Application Survival Experiments

This section examined the ability of shadow drivers to conceal failures from applications. The results demonstrate that applications that failed in any form on *Linux-Native* or *Linux-Nooks* ran normally with shadow drivers. Applications that depend on state in a driver benefited the most. In all cases, recovery proceeded quickly, compared to a full system reboot.

6.5 Testing Assumptions

The preceding section assumed that failures were fail-stop. However, driver failures experienced in deployed systems may exhibit a wider variety of behaviors. For example, a driver may corrupt the application, kernel, or device without the failure being detected. In this situation, shadow drivers may not be able to recover or mask failures from applications. This section uses fault injection experiments in an attempt to generate faults that may not be fail-stop.

[5] This driver is particularly slow at recovery. The other network drivers I tested recovered in less than a second.

6.5.1 Testing the Fail-stop Assumption

If Nooks' isolation services cannot ensure that driver failures are fail stop, then shadow drivers may not be useful. To evaluate whether device driver failures can be made fail-stop, I performed large-scale fault-injection tests of drivers and applications running on Linux-SD. For each driver and application combination, I ran 400 fault-injection trials using the methodology from Section 6.2.3. However, because VMware does not support the drivers in these tests, they were run on real hardware instead of a virtual machine. In total, I ran 2400 trials across the three drivers and six applications. I ensured the errors were transient by removing them during recovery. After injection, I visually observed the effect on the application and the system to determine whether a failure or recovery had occurred. For each driver, I tested two applications with significantly different usage scenarios. For example, I chose one sound-playing application (*mp3 player*) and one sound-recording application (*audio recorder*).

If I observed a failure, I then assessed the trial on two criteria: whether Nooks detected the failure, and whether the shadow driver could mask the failure and subsequent recovery from the application. For undetected failures, I triggered recovery manually. Note that a user may observe a failure that an application does not, for example, by testing the application's responsiveness.

Figure 6.3 shows the results of my experiments. For each application, I show the percentage of failures that the system detected and recovered automatically, the percentage of failures that the system recovered successfully after manual failure detection, and the percentage of failures in which recovery was not successful. The total number of failures that each application experienced is shown at the bottom of the chart. Only 16% of the injected faults caused a failure.

In my tests, 390 failures occurred across all applications. The system automatically detected 65% of the failures. In every one of these cases, shadow drivers were able to mask the failure and recover. Similar to the non-fatal extension failures when testing isolation, the system failed to detect 35% of the failures. However, in these cases, I manually triggered recovery. Shadow drivers recovered from nearly all of these failures (127 out of 135). Recovery was unsuccessful in the remaining 8 cases because either the system had crashed (5 cases) or the driver had corrupted the application beyond the possibility of recovery (3 cases). It is possible that recovery would have succeeded had a better failure detector discovered these failures earlier.

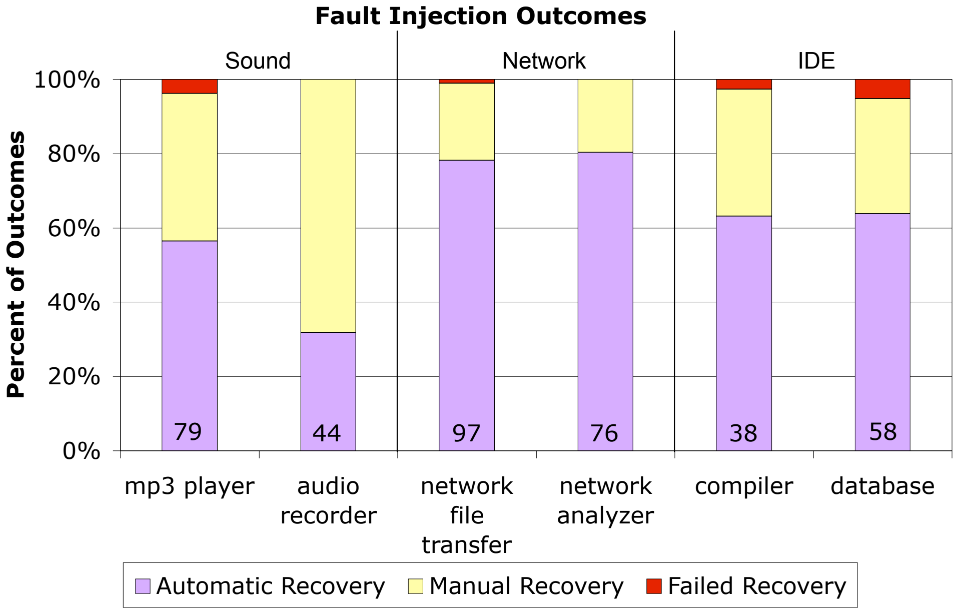

Figure 6.3: Results of fault-injection experiments on *Linux-SD*.

Across all applications and drivers, I found three major causes of undetected failure. First, the system did not detect application hangs caused by I/O requests that never completed (omission failures). Second, the system did not detect errors in the interactions between the device and the driver, e.g., incorrectly copying sound data to a sound card (output failures). Third, the system did not detect certain bad parameters, such as incorrect result codes or data values (again, output failures). For example, the *ide-disk* driver corrupted the kernel's disk request queue with bad entries. Detecting these three error conditions would require that the system better understand the semantics of each driver class. For example, 68% of the sound driver failures with *audio recorder* went undetected. This application receives data from the driver in real time and is highly sensitive to driver output. A small error or delay in the results of a driver request may cause the application to stop recording or record the same sample repeatedly.

My results demonstrate a need for better failure detectors to achieve high levels of reliability. Currently, Nooks relies on generic failure detectors that are not specialized to the behavior of a class of drivers. Shadow drivers provide an opportunity to add specialized detectors as part of the passive-mode monitoring code.

6.5.2 Recovery Errors

The fault injection tests in Section 6.3.2 provide further data on whether Nooks can ensure that extension failures are fail-stop. The *VFAT* file system manages persistent state stored on disk, and hence there is some chance that the extension damaged critical on-disk structures *before* Nooks detected an error condition.

In practice, I found that in 90% of the cases, *VFAT* recovery with the restart recovery manager resulted in on-disk corruption (i.e., lost or corrupt files or directories). Since fault injection occurs after many files and directories have been created, the abrupt shutdown and restart of the file system leaves them in a corrupted state. As an experiment, I caused Nooks to synchronize the disks with the in-memory disk cache before releasing resources on a *VFAT* recovery. This reduced the number of corruption cases from 90% to 10%. These statistics are similar to what occurs when the system crashes, and suggest that a recovery manager for file systems should perform this check during recovery.

6.5.3 Summary

This section examined the assumption that Nooks can make driver failures fail-stop. My test results show that most failures are fail-stop, but a substantial fraction are not. The problem was not one of isolation, as the kernel and applications continued to execute after manually triggered recovery, but of failure detection. Thus, further improving reliability demands driver class-specific failure detectors to raise the detection rate.

6.6 Performance

This section presents benchmark results that evaluate the performance cost of the Nooks and shadow drivers. The experiments use existing benchmarks and tools to compare the performance of a system using Nooks to one that does not. I ran most tests on a Dell 3 GHz Pentium 4 PC running Linux 2.4.18. The machine includes 1 GB of RAM, a SoundBlaster Audigy sound card, an Intel Pro/1000 Gigabit Ethernet adapter, and a single 7200 RPM, 80 GB IDE hard disk drive. My network tests

Table 6.5: The relative performance of Nooks compared to native Linux for six benchmark tests. An '*' in the first column indicates a different hardware platform.

Benchmark	**Extension**	*XPC Rate (per sec)*	*Nooks Relative Perf. (%)*	*Native CPU Util. (%)*	*Nooks CPU Util. (%)*
mp3 player	*audigy*	462	100	0.1	0.1
audio recorder	*audigy*	1059	100	0.3	0.6
network receive	*e1000* (receiver)	13,422	100	24.8	31.2
network send	*e1000* (sender)	58,373	99	27.8	56.7
DB queries	*ide-disk*	294	97	21.4	26.9
Compile-local	*ide-disk*	58	98	99.2	99.6
Compile-local*	*VFAT*	26,979	89	88.7	88.1
Serve-simple-web-page*	*kHTTPd* (server)	61,183	44	96.6	96.8
Serve-complex-web-page*	*e1000* (server)	1,960	97	90.5	92.6

used two identically equipped machines.[6] A few tests, indicated by a '*' in Table 6.5, were run on similar machines with a 1.7 GHz Pentium 4 with 890 MB of RAM. Unlike the reliability tests described previously, I ran all performance tests on a bare machine, i.e., one without the VMware virtualization system.

Table 6.5 summarizes the benchmarks used to evaluate system performance. For each benchmark, I used Nooks to isolate a *single* extension, indicated in the second column of the table. I ran each benchmark on native Linux without Nooks (*Linux-Native*) and then again on a version of Linux with Nooks and shadow drivers enabled (*Linux-SD*). I use the shadow driver recovery manager for device drivers and the restart recovery manager for *VFAT* and *kHTTPd*. [7] The table shows the relative change in performance for Nooks, either in wall clock time or throughput, depending on the benchmark. I also show CPU utilization measured during benchmark execution (which is only accurate to a few percent), as well as the rate of XPCs per second incurred during each test. I compute relative performance either by comparing latency (Play-mp3, Compile-local, DB queries) or throughput (network send, network receive, Serve-simple-web-page, Serve-complex-web-page). The data reflect the average of three trials with a standard deviation of less than 2%. The table shows

[6] I do not report performance information for the slower network and sound cards to avoid unfairly biasing the results in favor of Nooks.

[7] The serve-complex-web-page test uses the restart recovery manager as well.

that Nooks achieves between 44% and 100% of the performance of native Linux for these tests.

To separate the costs of isolation and recovery, I performed performance tests on the device drivers both with the restart recovery manager, which does no logging, and the shadow driver recovery manager, which logs requests to drivers. The performance and CPU utilization with the two recovery managers was within one percent, indicating that shadow drivers and their associated mechanisms do not add to the performance cost of using Nooks. Rather, it is the cost of Nooks' isolation services that determines performance.

As the isolation services are primarily imposed at the point of the XPC, the rate of XPCs offers a telling performance indicator. Thus, the benchmarks fall into three broad categories characterized by the rate of XPCs: low frequency (a few hundred XPCs per second), moderate frequency (a few thousand XPCs per second), and high frequency (tens of thousands of XPCs per second). I now look at each benchmark in turn.

6.6.1 Sound Benchmark

The mp3 player application plays an MP3 file at 128 kilobits per second through the system's sound card, generating only 470 XPCs per second. The audio recorder application records sound at 44,100 16-bit samples per second, and generates a similarly low XPC rate. At this low rate, the additional overhead of Nooks is imperceptible, both in terms of execution time and CPU overhead. For the many low-bandwidth devices in a system, such as keyboards, mice, Bluetooth devices [95], modems, and sound cards, Nooks offers a clear benefit by improving driver reliability with almost no performance cost.

6.6.2 Network Benchmarks

The network receive benchmark is an example of a moderate XPC-frequency test. Network receive performance was measured with the netperf [116] performance tool, where the receiving node used an isolated Ethernet driver to receive a stream of 32KB TCP messages using a 256KB buffer. The Ethernet driver for the Intel Pro/1000 card batches incoming packets to reduce interrupt, and hence XPC, frequency. Nevertheless, the receiver performs XPCs in the interrupt-handling code, which is on the critical path for packet delivery. This results in a no change in throughput but an overall CPU utilization increase of 7 percentage points.

In contrast, the network send benchmark (also measured using netperf) is a high XPC-frequency test that isolates the sending node's Ethernet driver. Unlike the network receive test, which benefits from the batching of received packets, the OS does not batch outgoing packets. Therefore, although the total amount of data transmitted is the same, the network send test executes almost four times more XPCs per second than network receive test. The overall CPU utilization on the sender thus doubles from about 28% on native Linux to 57% with Nooks. As with the network receive benchmark, throughput is almost unchanged. Despite the higher XPC rate, much of the XPC processing on the sender is overlapped with the actual sending of packets, mitigating some of the overhead from Nooks. Nevertheless, on slower processors or faster networks, it may be worthwhile to batch outgoing packets as is done, for example, in network terminal protocols [85].

6.6.3 Database Queries Benchmark

The database benchmark is a disk-intensive application chosen to stress the disk driver. It runs the mySQL database [122] with the Wisconsin database query benchmark [61]. Similar to the sound benchmarks, this benchmark generates a low XPC rate, approximately 300 XPCs per second. However, the CPU utilization during this test rose from 21% on *Linux-native* to 27% on *Linux-Nooks*. I analyze the cause of this slowdown when discussing the compile benchmark in the next section.

6.6.4 Compile Benchmark

As an indication of application-level file system and disk performance, I measured the time to untar and compile the Linux kernel, both on a local ext3 file system over the *ide-disk* driver and on a local *VFAT* file system. Table 6.5 shows that the compilation time was unchanged when *ide-disk* was isolated but ran about 10% slower when *VFAT* was isolated. In both cases, the CPU was nearly 100% utilized in the native case, so the additional overhead resulting from Nooks is directly reflected in the end-to-end execution time.

I take this benchmark as an opportunity to analyze the causes of Nooks' slowdown. For *ide-disk*, the low XPC rate of 58 per second shows why performance was unchanged, so I focus instead on *VFAT*. There are two possible reasons that the compile-local benchmark runs more slowly with Nooks: there is more code to run, and the existing code runs more slowly. To understand both sources of slowdown, I profiled the compile-local benchmark on the native Linux kernel and Linux with Nooks isolation.

I used statistical profiling of the kernel to break the execution time of the benchmark into components. The results are shown in Figure 6.4. User time (not shown) for all configurations was identical, about 477 seconds; however, kernel time is significantly different. For native Linux, the benchmark spends a total of 39 seconds in the kernel. With Nooks, the benchmark spends more than 111 seconds in the kernel. Most of the additional 72 seconds spent in the kernel with Nooks is caused by executing extra code that is not executed without Nooks. The new code in Nooks (shown by the upper bars in the Figure 6.4), accounts for 46 of the additional 72 seconds spent in the kernel under Nooks.

Of the 46 seconds spent in Nooks, more than half (28 seconds) is due just to XPC. Object tracking is another large cost (6 seconds), because Nooks consults a hash table to look up function parameters in the file system interface. Other components of Nooks, such as wrappers, synchronizing page tables with the kernel, and data copying have only minor costs.

These additional execution components that come with Nooks reflect support for isolation and recovery. The isolation costs manifest themselves in terms of XPC overhead, page table synchronization, and copying data in and out of protection domains. Nooks' support for recovery, with either the restart or the shadow driver recovery managers, is reflected in terms of the time spent tracking objects that occurs on every XPC in which a pointer parameter is passed. Tracking allows the recovery managers to recover kernel resources in the event of an extension failure. These measurements demonstrate that recovery has a substantial cost, or, more generally, that very fast inter-process communication (IPC) has it limits in environments where recovery is as important as isolation.

New code in Nooks is not the only source of slowdown. The lower bars in Figure 6.4 show the execution time for native Linux code in the kernel (labeled *other kernel*) and in the *VFAT* file system. For native Linux, 35 seconds were spent in the kernel proper and 4.4 seconds were spent in the *VFAT* code. In contrast, with Nooks the benchmark spent about 54 seconds in the kernel and 12 seconds in *VFAT*. The benchmark executes the same *VFAT* code in both tests, and most of the kernel

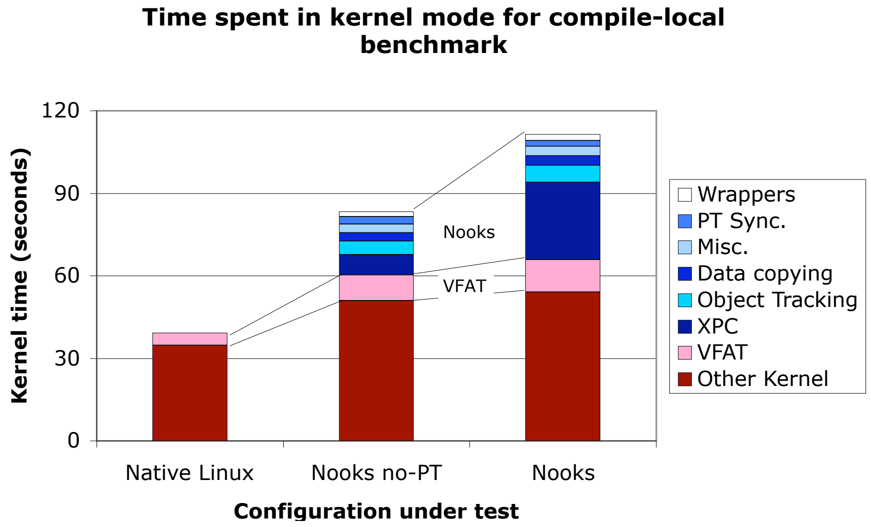

Figure 6.4: Comparative times spent in kernel mode for the Compile-local (*VFAT*) benchmark.

code executing is the same as well (Nooks makes a few additional calls into the kernel to allocate resources for the object tracker). The slowdown of VFAT and the kernel must therefore be caused by the processor executing the same code more slowly, due to micro-architectural events such as cache and TLB misses that Nooks causes.

I speculated that the both the high cost of XPC and the slowdown of existing kernel and *VFAT* code are due to the domain change, which is very expensive on the x86 architecture. Changing the page table register during the domain change takes many cycles to execute and causes the TLB to be flushed, causing many subsequent TLB misses. To further understand this effect, I built a special version of Nooks called *Nooks no-PT* that does *not* use a private page table. That is, all of the mechanism is identical to normal Nooks, but there is no address space switch between the driver and kernel domains. Comparing Nooks no-PT with normal Nooks thus allows us to isolate the cost of the page table switch instruction and the subsequent TLB misses caused by the TLB flush from other Nooks-induced slowdowns.

As shown by the center bar in Figure 6.4, the Nooks kernel without a separate page table (Nooks no-PT) spent 51 seconds in the kernel proper and 9 seconds was in *VFAT*. During the run with Nooks, the system performed 10,934,567 XPCs into the kernel and 4,0086,586 XPCs into the extension. The cost of Nooks was reduced to just 23 seconds, and the time spent doing XPCs dropped from

28 seconds to 7 seconds. These results demonstrate that changing the page table is an expensive operation, causing XPC to take many cycles, but that the TLB and cache misses caused by the change account for only 32% of the slowdown for *VFAT* and 17% of the kernel slowdown. Since the code in both configurations is the same, I speculate that the remaining performance difference between native Linux and Nooks is caused by additional TLB and cache misses that occur because Nooks copies data between the file system and the kernel. The Pentium 4 has just an 8KB L1 data cache, so it is particularly sensitive to additional memory accesses.

I have done little to optimize Nooks, but believe that significant speedup is possible through software techniques that have been demonstrated by other researchers, such as finely tuned data structures [178, 65] and request batching.

6.6.5 Web Server Benchmarks

The final two benchmarks illustrate the impact of Nooks on server performance of transactional workloads. Serve-simple-web-page uses a high XPC-frequency extension (*kHTTPd*) on the server to deliver static content cached in memory. I used httperf [147] to generate a workload that repeatedly requested a single kilobyte-sized Web page. *kHTTPd* on native Linux can serve over 15,000 pages per second. With Nooks, it can serve about 6,000, representing a 60% decrease in throughput.

Two elements of the benchmark's behavior conspire to produce such poor performance. First, the *kHTTPd* server processor is the system bottleneck. For example, when run natively, the server's CPU utilization is nearly 96%. Consequently, the high XPC rate slows the server substantially. Second, since the workload is transactional and non-buffered, the client's request rate drops as a function of the server's slowdown. By comparison, the network send benchmark, which exhibits roughly the same rate of XPCs but without saturating the CPU, degrades by only 10%. In addition, the network send benchmark is not transactional, so network buffering helps to mask the server-side slowdown.

Nevertheless, it is clear that *kHTTPd* represents a poor application of Nooks: it is CPU-limited and performs many XPCs. This service was cast as an extension so that it could access kernel resources directly, rather than indirectly through the standard system call interface. Since Nooks' isolation facilities impose a penalty on those accesses, performance suffers. I believe that other types of extensions, such as virus and intrusion detectors, which are placed in the kernel to access or protect resources otherwise unavailable from user level, would make better candidates as they do not represent system bottlenecks.

In contrast to *kHTTPd*, the second Web server test (Serve-complex-web-page) reflects moderate XPC frequency. Here, we ran the SPECweb99 workload [189] against the user-mode Apache 2.0 Web Server [5], with and without Nooks isolation of the *e1000* Ethernet driver. This workload includes a mix of static and dynamic Web pages. When running without Nooks, the Web server handled a peak of 114 requests per second. [8] With Nooks installed and the Ethernet driver isolated on the server, peak throughput dropped by about 3%, to 110 requests per second. This small slowdown is explained by the moderate XPC rate of almost 2000 per second.

6.6.6 Performance Summary

This section used a small set of benchmarks to quantify the performance cost of Nooks. For the device drivers, Nooks imposed a performance penalty of less than 3%. For the other kernel extensions, the impact was higher, and reached nearly 60% for *kHTTPd*, an ad-hoc application extension. The rate of XPCs is a key factor in the performance impact, as XPCs impose a high burden, due to cost of flushing the x86 TLB in the current implementation. The performance costs of Nooks' isolation services depend as well on the CPU utilization imposed by the workload. If the CPU is saturated, the additional cost can be significant, because there is no spare capacity to absorb Nooks overhead.

6.7 Summary

Overall, Nooks provides a substantial reliability improvement at low cost for common drivers. The results demonstrate that: (1) the performance overhead of Nooks during normal operation is small for many drivers and moderate for other extensions, (2) applications and the OS survived driver failures that otherwise would have caused the OS, application, or both to fail. Overall, Nooks and shadow drivers prevented 99% of system crashes and 98% of application failures, with an average performance cost of 1% for drivers. The additional cost of shadow driver recovery as compared to restarting a driver is negligible.

[8] The test configuration is throughput limited due to its single IDE disk drive.

These results indicate that Nooks and shadow drivers have the potential to significantly improve the reliability of commodity operating systems at low cost. Given the performance of modern systems, I believe that the benefits of isolation and recovery are well worth the costs for many computing environments.

Chapter 7

THE DYNAMIC DRIVER UPDATE MECHANISM

7.1 Introduction

In addition to tolerating driver failures, Nooks and shadow drivers provide a platform for other reliability services. The Nooks interposition service provides a mechanism to hook communication between drivers and the kernel, as evidenced by shadow drivers' use of it for taps. The object tracker provides a mechanism to discover the objects in use by a driver and selectively release those objects to the kernel. Shadow drivers provide the ability to actively proxy for drivers, concealing their unavailability, and to transfer state between two driver instances. As a demonstration of how these capabilities can be leveraged to provide new services, I implemented *dynamic driver update*, a mechanism to replace drivers on-line on top of Nooks and shadow drivers.

As commodity operating systems become more reliable and fault-tolerant, the availability of a system will be determined not by when it crashes, but instead by when it must be shutdown and rebooted due to software maintenance [91, 150]. While many system components can be upgraded on-line, critical low-level components, such as device drivers and other kernel extensions, cannot be updated without rebooting the entire operating system. Thus, scheduled downtime promises to be the ultimate limiter of system availability because operating systems cannot replace all running code on-line.

Device drivers represent a large portion of this problem because they are both critical to OS operation and frequently updated. For example, Microsoft's Windows Update reports 115 updates for Windows XP between 2000 and 2005, but over 480 updates for network drivers and 370 updates for video drivers [143]. The picture is similar in Linux, where IDE storage drivers, needed by the OS for virtual memory swapping, were updated more than 67 times in three years [196]. In addition, studies show that patches and updates frequently cause failures, by introducing new bugs not found in testing [192, 57].

To achieve high availability, operating systems must be able to replace critical device drivers without rebooting the entire system. Furthermore, updating device drivers is inherently risky because a faulty device driver may prevent the system from booting. Thus, driver updates must be isolated to ensure that the new code does not compromise system stability, and it must be possible to "undo" an update that does not function.

Updating drivers on-line presents three major challenges. First, there are thousands of existing drivers, many of which will eventually require an update. To make on-line update practical, the system must be able to update these drivers without any work by the drivers' authors. Existing systems for updating code on-line require that a skilled programmer write code to transfer data between the old and new versions [75, 66, 103, 161].

Second, because drivers assume exclusive access to devices and require access to the device to initialize, the old driver must be disabled while the new driver initializes. Thus, the device is unavailable during an update. Any portion of the update process that depends on a device being present (e.g., loading the new driver off a disk when updating the disk driver) must be performed before the old driver is disabled.

Finally, an updated driver may offer substantially different services or support different interfaces. For example, a new version of a driver may remove functions or capabilities provided by the prior version. The updated device driver may not be compatible with the running version and could cause applications or the OS to crash. Thus, the update system must ensure that an update is safe and does not contain incompatibilities that could compromise system reliability.

In this chapter, I present the design and implementation of dynamic driver update, a service that replaces driver code on-line. It can update common device drivers without restarting applications or the operating system. When combined with Nooks' isolation services, dynamic driver update also addresses the problem that updates are not tested as completely as original code. With isolation, the OS can tolerate any failures of the updated driver either by restarting the new version of the driver or by rolling back to a previous version. As a result, administrators of high-availability systems can apply important but risky updates, such as security patches, whereas today they might forgo such patches.

In the remainder of this chapter, I present background and related work in Section 7.2. In Section 7.3 I present the architecture and implementation of dynamic driver update, and in Section 7.4

I evaluate its usefulness and performance.

7.2 Related work

High-availability systems have long sought to update code on-line [75, 64]. A study of Windows NT 4.0 installations showed that 42% of outages resulted from software maintenance. In addition, a study at IBM showed that 20% of system failures were caused by faults in poorly tested patches [191]. Hence, the reliable upgrade of system components is critical to high availability. Previous approaches to this problem differ in the programmer's responsibility, the update mechanisms and the granularity of updates. I discuss each of these in turn.

Previous on-line update systems require a programmer to provide code to facilitate the update. The programmer's responsibilities range from following design-time rules, such as inheriting from a specific base class, to writing functions that translate data formats when they change. The systems that rely on a compiler to provide the mechanism for updating code require the least programmer effort. OPUS, for example, uses a compiler to generate patches automatically from two versions of the source code [2], but is restricted to patches that do not change data formats. Dymos [129] and Dynamic Software Updating [103] can update arbitrary code but require that programmers write a conversion function if data types change.

Other approaches place more responsibility on the programmer. Systems that rely on object-oriented language features, such as indirect function calls, require that updatable code inherit from an "updatable" virtual base class. Programming-language specific systems, such as dynamic C++ classes [105], and dynamic update for Java [161] take this approach, as does the k42 operating system. Another task placed on programmers is to ensure that data structures remain consistent during an update by identifying locations in the code where updates can occur safely [98]. At the most extreme, the IROS platform includes a method to update driver code in the driver interface, effectively pushing all the work onto the programmer [7].

In some cases, a system can reduce the programmer's role by leveraging the existing properties of the updatable code. For example, the Simplex system updates real-time event handlers with no additional code by running old and new handlers in parallel until their outputs converge [84]. Microsoft's most recent web server, IIS 7.0, relies on the independence of separate web requests to replace extensions at run-time. As soon as an updated extension is available, the web server routes requests to the new version [148].

There have been four major mechanisms used to replace updated code. First, within a cluster of replicated servers, an administrator can restart an entire machine to load the new code while the remaining machines provide service. The Visa payment processing system, for example, applies 20,000 updates a year with this mechanism yet achieves 99.5% availability [166]. Recent work on virtual machines may enable this technique to be applied on a single computer [137]. The benefit of this approach is that it is very general; anything can be updated by rebooting the whole machine. However, it requires the most work from application programmers, as these systems rely on the application to perform failover and state replication [93].

Second, the system can launch a copy of a running process with the new code on the same hardware [94], or on additional hardware [181], and then copy in the state of the existing process. This approach is limited to user-mode processes, as without the support of a virtual machine monitor, two operating system kernels cannot coexist on a single machine.

Third, the system can patch running code by dynamically dispatching callers to functions that have been updated. In this case, wrapper functions [75, 89, 188], dynamically dispatched functions [105, 161, 66], and compilers that produce only indirect function calls [129, 103] can all redirect callers to the new code. This approach, while requiring more infrastructure, has the benefit of being working in many environments and with many program changes.

Finally, the system can directly modify code in memory by replacing the target address of a function call [2] or by replacing any instruction with a branch to new code [120]. This approach is limited to code changes and cannot handle data format changes.

Several different approaches have been used to dynamically update operating system code. The k42 microkernel operating system is built upon object-oriented components that can be hot-swapped and updated dynamically [188, 21]. The k42 update system relies on the unique object model of k42 and hence is difficult to apply to commodity operating systems that are not object-oriented [21]. The IROS real-time operating system includes a method in the driver interface that allows two drivers to coexist simultaneously and to transfer state [7]. The operating system acts as a wrapper and redirects calls to the new driver. Thus, driver writers must provide code to support updating drivers. Kishan and Lam applied instruction editing to the Mach kernel for both extensibility and optimization [120]. As previously mentioned, this approach only can only apply changes that do not modify data formats and hence is not broadly applicable.

Dynamic driver update provides on-line update for device drivers. As with Nooks, my paramount goal is compatibility with existing code. Dynamic driver update relies on shadow drivers to update a device driver in-place without assistance from a programmer. My update system uses the common interfaces of device drivers to capture and transfer their state between versions automatically. Thus, compatibility reduces the deployment cost by removing the programmer support requirement of previous approaches.

7.3 Design and Implementation

In keeping with the Nooks design principles (see Section 4.1), dynamic driver update (DDU) reflects a practical, systems-oriented approach to updating device drivers on-line. Existing approaches to dynamically updating code rely on the compiler, programming language, or programmer to perform the update. In contrast, dynamic driver update is a service that updates drivers on their behalf.

The design of DDU shares the principles of shadow drivers:

1. *Update logic should be generic.*
2. *Device driver updates should be concealed from the driver's clients.*
3. *Update logic should be centralized.*
4. *Update services should have low overhead when not needed.*

and adds one more:

5. *Safety first.* The system may disallow an update if it is not compatible with running code.

The update system must work with existing drivers because it is too hard, expensive, and error-prone to add on-line update support to drivers. With a systems-oriented approach, a single implementation can be deployed and used immediately with a large number of existing drivers. Thus, I centralize the update code in a single system service. As a common facility, the update code can be thoroughly examined and tested for faults. In contrast, practical experience has shown us that many drivers do not receive such scrutiny, and hence are more likely to fail during an update [57].

These principles limit the applicability of dynamic driver update. My solution applies only to drivers that can load and unload dynamically. Furthermore, my design only applies to drivers that share a common calling interface. A driver that uses a proprietary or ad-hoc interface cannot be updated automatically without additional code specialized to that driver. Finally, if the new driver supports dramatically different capabilities than the existing driver, the update will fail because the changes cannot be hidden from the OS and applications.

Because of these limitations, I consider dynamic driver update to be an optimization and preserve the possibility of falling back to a whole-system reboot if necessary.

7.3.1 Design Overview

Dynamic driver update replaces a driver by monitoring the state of a driver as it executes, loading the new code into memory, starting the new driver, and transferring the old driver's state into the new driver. During the update, the system impersonates the driver, ensuring that its unavailability is hidden from applications and the OS. As the new driver starts, the update system connects the new driver to the old driver's resources and verifies that the new driver is compatible with the old driver. If it is not compatible, the system can roll back the update. In essence, dynamic driver update replaces the driver code and then reboots the driver.

Similar to recovering from driver failures, dynamic driver update seeks to conceal the update from the OS and applications. Unlike a whole-system reboot, where the ephemeral state of a driver is lost, restarting just the driver requires that this state be preserved so that application and OS requests can be processed correctly. Hence, dynamic driver update leverages shadow drivers to restart an updated device driver without impacting applications or the OS. Because it is not concerned with fault tolerance, dynamic driver update uses only the object tracking and interposition services necessary for shadow drivers. The object tracker records kernel objects in use by the driver for later reclamation during an update. The interposition service supports taps, which invoke the shadow on every call between a driver and the kernel. Isolation may optionally be used when the updated code is potentially faulty. Dynamic driver update also leverages shadow drivers to obtain the state of a driver and to transfer state between driver versions.

Table 7.1 lists the steps of updating a driver. The process begins when a driver is initially

Table 7.1: The dynamic driver update process.

Update Steps
1. Capture driver state
2. Load new code
3. Impersonate driver
4. Disable/garbage collect old driver
5. Initialize new driver
6. Transfer in driver state
7. Enable new driver

loaded and the system starts capturing its state and ends after when the new driver begins processing requests. I now describe each of the update steps in more detail.

7.3.2 Capturing Driver State

From the moment a driver is first loaded, a shadow driver monitors its state. The state of a device driver consists of: (1) the static global variables and constants in the driver code, (2) the state it receives from hardware, and (3) the state it receives from the kernel. The state from code is static, and hence does not need to be preserved across an update. From hardware, a driver receives persistent data and environmental parameters, such as the speed of a network link. For most drivers either this information is stable (i.e., does not change over time) or the driver maintains no history of this information and reacquires it as needed. As a result, this state need not be explicitly preserved when updating a driver.

From the kernel, the driver receives configuration and I/O requests. While configuration changes must persist across an update, completed I/O requests generally have no impact on future requests. Hence, they need not be retained after a driver update. Thus, the only driver state that must be preserved across an update is the history of configuration requests and the requests it is currently processing.

Dynamic driver update depends on shadow drivers and the object tracker in Nooks to capture the state of a device driver. The object tracker records kernel resources in use by a driver, so that the update system can disable and garbage collect the old driver. The shadow logs configuration requests and in-progress I/O requests. With this information, the shadow can restore the driver's

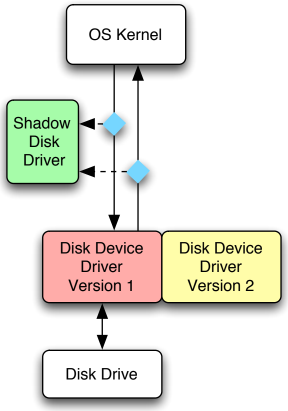

Figure 7.1: A shadow driver capturing the old driver's state.

internal state after an update. The class-based nature of device drivers allows a single shadow driver to capture the state of any driver in its class. As a result, no code specific to a single driver is required for capturing driver state. Figure 7.1 shows a shadow driver capturing driver state by monitoring communication between the kernel and the driver. In this figure, a new driver version has been pre-loaded into memory for an upcoming update.

7.3.3 Loading Updated Drivers

A system operator begins an update by loading the new driver code. As discussed in Section 7.1, the services of a driver are not available while it is restarting. This is an issue, for example, if the update mechanism relies on disk access after the disk driver has been disabled. To avoid such circular dependencies, an operator must pre-load the new driver version into memory so that the system can update drivers that store their own updates.

Dynamic driver update logically replaces the entire driver at once. While the actual change to a driver may be localized to a single function, replacing the entire driver allows the system to leverage existing OS support for loading and initializing drivers. When updating only a subset of the modules comprising a multi-module driver, the update system optimizes the in-memory footprint by replacing only those modules whose memory images change. In addition to new modules, this includes any module that links to the new module, because function addresses may have changed.

Figure 7.1 shows the new version loaded in memory alongside the existing version. Once the new code is in memory, an operator notifies the update system to replace the driver. The driver update system leaves the old driver code in memory until the update completes, which allows the system to revert back to the old driver if the update fails. For example if the new driver does not initialize or is not compatible with the old driver, the update system restarts the old driver.

7.3.4 Impersonating the Driver

Dynamic driver update relies on shadow drivers to impersonate the old driver during the update. I previously described the process in Section 5.3.4. The shadow driver actively proxies for the driver, ensuring that applications and the OS do not observe that it is unavailable. For systems that update user-mode code, the application can be suspended during the update process. However, drivers are frequently called at interrupt level, where calls cannot be suspended, and hence active proxying is necessary.

7.3.5 Garbage Collecting the Old Driver

The goal of this stage is to release any resources in use by the old driver that are not needed as part of the update process. As with impersonating the driver, shadow drivers perform this function. Those objects that the kernel relies on to invoke the driver, such as driver registration objects, and the hardware resources in use by a driver, such as interrupt-request lines and I/O memory regions, are retained for restarting the driver. The shadow releases everything else, including dynamically allocated memory and other support objects, such as timers. Section 5.3.3 contains the details. After this stage, both old and new driver are present in memory in a clean, uninitialized state and either one can be started. Starting the old driver is equivalent to recovering after a failure, while starting the new one updates the driver code.

7.3.6 Initializing the New Driver

Once the old driver is garbage collected, the shadow driver initializes the new driver with the same sequence of calls that the kernel makes when initializing a driver. Thus, the new driver starts exactly as if the kernel initialized it. For example, when starting a network device driver, the shadow calls the driver's `init_module` function, followed by the `open` function to initialize the network, followed by `set_multicast_list`. In addition, the shadow also calls internal kernel routines to ensure the network driver is properly bound into the kernel, such as `dev_activate` and `netif_schedule` to notify the kernel that the driver is prepared to start sending packets. The sequence of calls is hardwired into the shadow driver for a particular class of drivers. Hence, if the kernel evolves and makes additional calls during driver initialization, the shadow must similarly be updated. During this initialization stage, taps send all calls made by the updated device driver to the shadow instead of the kernel, allowing the shadow to connect it to the resources of the old driver.

The benefit of replicating the kernel's initialization sequence is that no new code is needed in the driver for an update. Instead, dynamic driver update repurposes the driver's existing initialization code by controlling the driver's environment during restart, spoofing it into believing that the kernel is starting it normally.

The update system faces two major challenges in starting the new driver: identifying existing kernel objects that the new driver requests, and ensuring that the new driver is compatible with the old driver. I now discuss each of these challenges in detail.

Updating References

When the new driver requests a resource from the kernel or provides a resource to the kernel, the shadow must locate the corresponding resource belonging to the old driver. The shadow calls into the update system to perform this mapping. Once located, the update system updates the resource to reference the new driver. The primary challenge in this process is identifying *which* resource of the old driver corresponds to the resource used by the new driver.

Drivers usually request kernel resources with a unique identifier, which allows the update system to locate the old driver's corresponding resource. For example, drivers provide a unique name, device number, or MAC address when registering with the kernel, and unique memory addresses when requesting I/O regions. The update system uses the unique identifier and resource type to locate the old driver's resource in the object tracker.

In some cases, the driver passes a resource to the kernel without unique identification. The update system then uses the context of the call (i.e., the stage of the restart) to make the association.

For example, when the kernel opens a connection to a sound-card driver, the driver returns a table of function pointers with no identifying device name or number. To transfer these function tables to the new driver, the shadow calls into the new driver to re-open every connection. The new driver returns its function table. Because the shadow driver knows that it is re-opening a connection, it notifies the update system to replace the function table in the existing connection object with the new functions provided by the driver. This contextual approach to matching resources can be applied elsewhere, for example, when updating a network driver that lacks a unique MAC address.

After locating the appropriate kernel resource, the update system modifies it to reflect differences between the old and new driver. For example, the new driver may provide additional functions or capabilities. In some cases, the update system need not update the kernel object directly. Instead, it relies on the layer of indirection provided by wrappers and the object tracker to update the mapping between kernel and driver versions. For example, when updating interface function pointers, the update system replaces the mapping maintained by the object tracker rather than changing function pointers in the kernel. If this is not possible, the update system locks and updates the kernel object directly.

Detecting Incompatibilities

A second issue that arises when updating code without programmer support is ensuring that the new code is compatible with the running code. If an update is not compatible, it may cause the OS and applications to fail. Incompatibilities arise when a new driver implements different functions (e.g., higher-performance methods), different interfaces (e.g., a new management interface), or different options for an existing function. Complicating matters, applications may expose incompatibilities long after a new driver starts. For example, an application may use capabilities it queried from the old driver that are not present in the new driver. Reflecting the “safety first” principle, the update system must detect whether the update could cause callers to function incorrectly in order to prevent applications from seeing unexpected behavior.

Drivers express their capabilities in two ways. First, drivers may provide only a subset of the functions in an interface. For example, a driver writer may choose not to implement certain high-performance methods. Second, drivers can express their capabilities through feature fields. For example, network drivers pass a bit-mask of features, such as the ability to offload TCP checksumming and do scatter/gather I/O, to the kernel. The update system considers an update compatible only if a new driver provides at least the same capabilities as the old driver.

As part of transferring references to the new driver, the update system performs three compatibility checks. First, it verifies that the new driver provides at least the same interfaces to the kernel as the old driver. If a driver has not registered all of the old driver's interfaces during initialization, it is incompatible. Second, the update system verifies that, within an interface, the new driver implements at least the functions that the old driver provided. If so, the system updates the object tracker to reference the new driver. Third, for drivers with features that can be queried by applications, the update system explicitly queries the new driver's feature set during an update and compares it against the old driver's feature set. Thus, the update system detects incompatibilities during the update process, ensuring that there will not be latent problems discovered long after the process completes.

When it detects an incompatible update, the update system has three options. It can take a conservative approach and revert to the existing driver. Or, it can let the update go forward but fail any future request that would reveal the incompatibility. The final option is to forgo compatibility checks and allow applications to observe the incompatibilities. These options allow an operator to prioritize an important update (for example, one that prevents unauthorized use of the system) over compatibility. Lower-priority incompatible updates can still be applied with a whole-system reboot.

Rolling back an update presents compatibility problems as well. If the new driver provides a superset of the old driver's capabilities, it may not be possible to roll back an update without notifying applications that observed the new driver's capabilities. To preserve the option to roll back, the update system can optionally conceal the new capabilities of a driver by masking out new capabilities or functions to equal those in the old driver. Once the update has been tested, an administrator can apply it again without concealing its capabilities. This ensures that a potentially faulty update can be tested without risking system stability.

For some driver updates, these rules are overly conservative. For example, if a driver provides a function or capability that is never used, a new version need not provide that function. However, the update system cannot know whether an application will call the function later, and hence uses conservative checks.

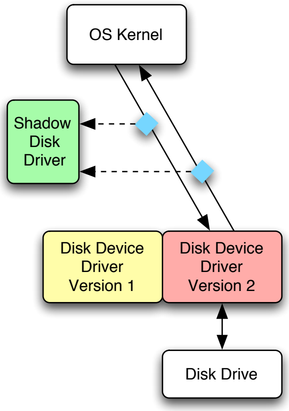

Figure 7.2: Device driver and shadow after an update.

7.3.7 Transferring Driver State

While previous on-line update systems require applications to transfer their own state between versions, dynamic driver update transfers driver state automatically, without the involvement of driver writers. As discussed in Section 5.3.3, the shadow driver captures the state of a driver by logging its communication with the kernel. It then transfers the old driver's state into the new driver by replaying the log, using the same calls into the driver that it previously monitored. Shadow drivers' ability to automatically capture and restore the state of unmodified drivers is critical to providing dynamic update without programmer involvement.

7.3.8 Enabling the New Driver

The final step of an update is to enable the new driver. First, the shadow submits any requests that it queued during the update, and then it notifies the taps that requests should be sent to the real driver instead of the shadow driver. After the new driver is running, operators may either free the memory occupied by the old driver code or leave it in memory if they desire the opportunity to roll back the update later. The system state after an update is shown in Figure 7.2. In this figure, the kernel calls into the new driver while the old driver remains in memory in case the update must be rolled back.

Table 7.2: The number of lines of code dynamic driver update.

Source Components	**# Lines**
Linux kernel	181
Nooks	17
Shadow drivers	51
Dynamic driver update	402
Total number of lines of code	651

7.3.9 Implementation Size

I constructed dynamic driver update as a service on top of Nooks and shadow drivers. As result, little additional code was necessary to implement this service. Table 7.2 shows the breakdown of code between changes to the kernel, changes to Nooks isolation code, changes to shadow driver recovery code, and new code components. The kernel had to be changed to support loading multiple versions of a driver simultaneously. The changes to Nooks allow waiting for the driver to become quiescent instead of preempting threads currently executing in the driver. The most substantial changes were to the shadow driver code, in which I added new call-outs for mapping between old and new versions of driver resources and for compatibility checks. The small size of the entire implementation demonstrates the usefulness of Nooks and shadow drivers as a platform, as without this code the entire service would have been many times larger.

7.3.10 Summary

Dynamic driver update takes a systems-oriented approach to on-line update by providing a centralized service to update unmodified drivers. Before an update, it loads driver code into memory to avoid circular dependencies. During an update, a shadow driver hides the driver's unavailability from applications and the OS by handling requests on the driver's behalf. After an update, the system presents the option of rolling back to the old version automatically if the new version fails.

Dynamic driver update relies on existing driver interfaces to update the driver and transfer the state of the current driver. As the new driver initializes, the update system connects it to the resources of the old driver. The update system reassigns kernel structures referencing the old driver to the new driver. Once the new driver has initialized, the shadow driver transfers in the state of the old driver by replaying its log of requests and by re-opening connections to the driver. Finally, the update system dynamically detects whether the old and new drivers are compatible by comparing their capabilities. If the new driver is not compatible, either the update or the action that would reveal the incompatibility can be canceled. When used with Nooks isolation services, the system allows potentially faulty updates to be tested without compromising availability.

Dynamic driver update is built on top of Nooks and shadow drivers. Little new code was needed to implement this service, as Nooks and shadow drivers provide object tracking, interposition, active proxying, and restarting services. Thus, dynamic driver update demonstrates that Nooks and shadow drivers are a useful platform for implementing additional reliability and availability services.

7.4 Evaluation of Dynamic Driver Update

In this section I answer three questions about dynamic driver update:

1. *Transparency.* Can dynamic driver update replace *existing* drivers?
2. *Compatibility.* Can it detect and roll back incompatible updates?
3. *Performance.* What is the performance overhead of DDU during normal operation (i.e., in the absence of updates), and are on-line updates significantly faster than rebooting the whole system?

Based on a set of experiments using existing, unmodified device drivers and applications, my results show that dynamic driver update (1) can apply most released driver updates, (2) can detect when an update is incompatible with the running code and prevent application failures by rolling back the update, and (3) imposes only a minimal performance overhead.

7.4.1 Methodology and Platforms

I implemented dynamic driver update in the Linux 2.4.18 operating system kernel. I produced two OS configurations based on the Linux 2.4.18 kernel:

1. *Linux-Native* is the unmodified Linux kernel.
2. *Linux-DDU* is a version of Linux that includes the Nooks wrappers, object tracker, shadow drivers plus the dynamic driver update code. It does not include the Nooks isolation services.

Table 7.3: The three classes of shadow drivers and the Linux drivers tested.

Class	**Driver**	**Device**
Sound	***emu10k1***	**SoundBlaster Live! sound card**
	audigy	SoundBlaster Audigy sound card
	es1371	Ensoniq sound card
Network	***e1000***	**Intel Pro/1000 Gigabit Ethernet**
	3c59x	3COM 3c509b 10/100 Ethernet
	pcnet32	AMD PCnet32 10/100 Ethernet
Storage	***ide-disk***	**IDE disk**

I ran the experiments on a 3 GHz Pentium 4 PC with 1 GB of RAM and a single 80 GB, 7200 RPM IDE disk drive. Networking tests used two identically configured machines.

I tested dynamic driver update with the same three shadow drivers used previously in Chapter 6: sound card, network interface controller, and IDE storage device. To ensure that dynamic driver update worked consistently across device driver implementations, I tested it on seven different Linux drivers, shown in Table 7.3. Although I present detailed results for only one driver in each class (shown in boldface, *emu10k1*, *e1000*, and *ide-disk*), behavior across all drivers was similar.

Of these three classes of drivers, only IDE storage drivers require a full system reboot in Linux to be updated. However, updating members of the other two classes, sound and network, disrupts applications. Dynamic driver update improves the availability of applications using these drivers, because they need not be restarted when a driver is updated.

7.4.2 Effectiveness

To be effective, the dynamic driver update system must hide updates from the applications and the OS, apply to most driver updates, and detect and handle incompatible updates. Dynamic driver update depends on shadow drivers to conceal updates. As a result, it shares their concealment abilities. Section 6.4 demonstrated that shadow drivers could conceal recovery from all tested applications in a variety of scenarios. As a result, I only examine the latter two requirements: whether dynamic driver update can apply existing driver updates and whether it can detect and roll back incompatible updates.

Table 7.4: Tested drivers, the number of updates, and the results.

Driver	*Module*	*# of updates*	*# Failed*
emu10k1	emu10k1	10	0
	ac97_codec	14	0
	soundcore	6	0
e1000	e1000	10	1
ide-disk	ide-disk	17	0
	ide-mod	17	0
	ide-probe-mod	17	0

.

Applicability to Existing Drivers

I tested whether dynamic driver update can update existing drivers by creating a set of different versions of the drivers. I compiled past versions of each driver, which I obtained either from the Linux kernel source code or from vendor web sites. For drivers that consist of multiple modules, I created separate updates for each module and replaced the modules independently. Because the driver interface in the Linux kernel changes frequently, I selected only those drivers that compile and run correctly on the 2.4.18 Linux kernel. Table 7.4 show the modules comprising each of my tested drivers, the number of updates for each module, and the number of updates that failed due to compatibility checks.

Using the *Linux-DDU* kernel, I applied each driver update from oldest to newest while an application used the driver. For the sound-card driver, I played an mp3 file. For the network drivers, I performed a network file copy, and for the IDE storage driver I compiled a large program. For each update, I recorded whether the DDU successfully updated the driver and whether the application and OS continued to run correctly. If the update process, the application or the OS failed, I consider the update a failure.

The results, shown in Table 7.4, demonstrate that dynamic driver update successfully applied 90 out of 91 updates. In one case, when updating the *e1000* driver from version 4.1.7 to 4.2.17, the shadow could not locate the driver's I/O memory region when the new driver initialized. The new driver used a different kernel API to request the region, which prevented the update system from finding the region in the object tracker. As a result, the new driver failed to initialize and caused the update system to roll back. The file-copy application, however, continued to run correctly using the old driver.

This single failure of dynamic driver update demonstrates the complexity of transferring kernel resources between driver versions. Shadow drivers require a unique identifier to locate the resource of an old driver that corresponds to a new driver's request. Some of the unique identifiers I chose depend on the API used to request the resource. When a driver changes APIs, the shadow cannot locate the old driver's resources during an update. Ideally, the identifier should be independent of the API used to request the resource. In practice, this is difficult to achieve within the existing Linux kernel API, where there are many resources that can be accessed through several different APIs.

Compatibility Detection

While I found no compatibility problems when applying existing driver updates, compatibility detection is nonetheless critical for ensuring system stability. To test whether dynamic driver update can detect an incompatible update, I created a new version of each of the tested drivers that lacks a capability of the original driver. I created *emu10k1-test*, a version of the *emu10k1* driver that supports fewer audio formats. The update system detects the sound-card driver's supported formats by querying the driver's capabilities with an `ioctl` call after it initializes. From the *e1000* driver, I created *e1000-test* that removes the ability to do TCP checksumming in hardware. The driver passes a bit-mask of features to the kernel when it registers, so this change should be detected during the update. Finally, I created *ide-disk-test*, which removes the `ioctl` function from *ide-disk* . This change should be detected when the driver registers and provides the kernel with a table of function pointers.

For each test driver, I first evaluate whether compatibility detection is necessary by updating to the test driver with the compatibility checks disabled. Similarly to the previous tests, I ran an application that used the driver at the time of update. With compatibility checking enabled, I then applied the same update. For each test, I recorded whether the system applies the update and whether the application continued to run correctly.

Table 7.5 shows the outcome of this experiment. With compatibility checking disabled, all three drivers updated successfully. The sound application failed, though, because it used a capability of the old driver that the new driver did not support. The disk and network applications continued to

Table 7.5: The results of compatibility tests.

Driver	**No Compatibility Check?**	**Compatibility Check?**
emu10k1-test	Application fails	Update rolled back
e1000-test	Works	Update rolled back
ide-disk-test	Works	Update rolled back

execute correctly because the kernel checks these drivers' capabilities on every call, and therefore handles the change in drivers gracefully. With compatibility checking enabled, dynamic driver update detected all three driver updates as incompatible and rolled back the updates. The applications in these tests continued to run correctly.

These tests demonstrate that compatibility checking is important for drivers whose capabilities are exposed to applications. The sound application checks the driver's capabilities once, when starting, and then assumes the capabilities do not change. In contrast, only the kernel is aware of the network and disk drivers' capabilities. Unlike the sound application, the kernel checks the driver capabilities before every invocation of the driver, and hence the compatibility check during update is not necessary.

7.4.3 Performance

Relative to normal operation, driver updates are a rare event. As a result, an on-line update system must not degrade performance during the intervals between updates. In addition, updates must occur quickly to achieve high availability. Dynamic driver update introduces overhead from taps, which invoke the shadow on every function call between the kernel and drivers, and from the shadow driver itself, which must track and log driver information.

I selected a variety of common applications that use the three device driver classes. The application names and workloads are shown in Table 7.6. I measured their performance on the *Linux-Native* and *Linux-DDU* kernels. This test measures the cost of the update system alone, without isolation. If the updated driver is potentially faulty, then Nooks isolation should be used as well. Because dynamic driver update adds no additional overhead beyond the logging performed by shadow drivers, the performance with isolation is the same as reported in Section 6.6.

Table 7.6: The applications used for evaluating dynamic driver update performance.

Device Driver	**Application Activity**
Sound *(emu10k1 driver)*	• mp3 player (*zinf*) playing 128kb/s audio • audio recorder (*audacity*) recording from microphone
Network *(e1000 driver)*	• network send (*netperf*) over TCP/IP • network receive (*netperf*) over TCP/IP
Storage	• compiler (*make/gcc*) compiling 788 C files • database (*mySQL*) processing the *Wisconsin Benchmark*

Different applications have different performance metrics of interest. For the sound-card driver, I measured the available CPU while the programs ran. For the network driver, throughput is a more useful metric; therefore, I ran the throughput-oriented *network send* and *network receive* benchmarks. For the IDE storage driver, I measured elapsed time to compile a program and to perform a standard database query benchmark. I repeated all measurements several times and they showed a variation of less than one percent.

Figure 7.3 shows the performance of *Linux-DDU* relative to *Linux-Native*, and Figure 7.4 shows the CPU utilization of the tests on both platforms. These figures make clear that dynamic driver update imposes only a small performance penalty compared to running on *Linux-Native*. Across all six applications, performance on *Linux-DDU* averaged 99% of the performance on *Linux-Native*. However, CPU utilization varied more, rising by nine percentage points for the network send benchmark. As a comparison, the average CPU utilization increase with Nooks isolation was 5%, while with dynamic driver update it was just 2%. Overall, these results show that dynamic driver update imposes little run-time overhead for many drivers.

As with the cost of Nooks, this overhead can be explained in terms of frequency of communication between the driver and the kernel. On each kernel-driver call, the shadow driver may have to log a request or track an object. For example, the kernel calls the driver approximately 1000 times per second when running *audio recorder*, each of which causes the shadow to update its log. For the most disk-intensive of the IDE storage applications, the *database* benchmark, the kernel and driver interact only 290 times per second. The *network send* benchmark, in contrast, transmits 45,000

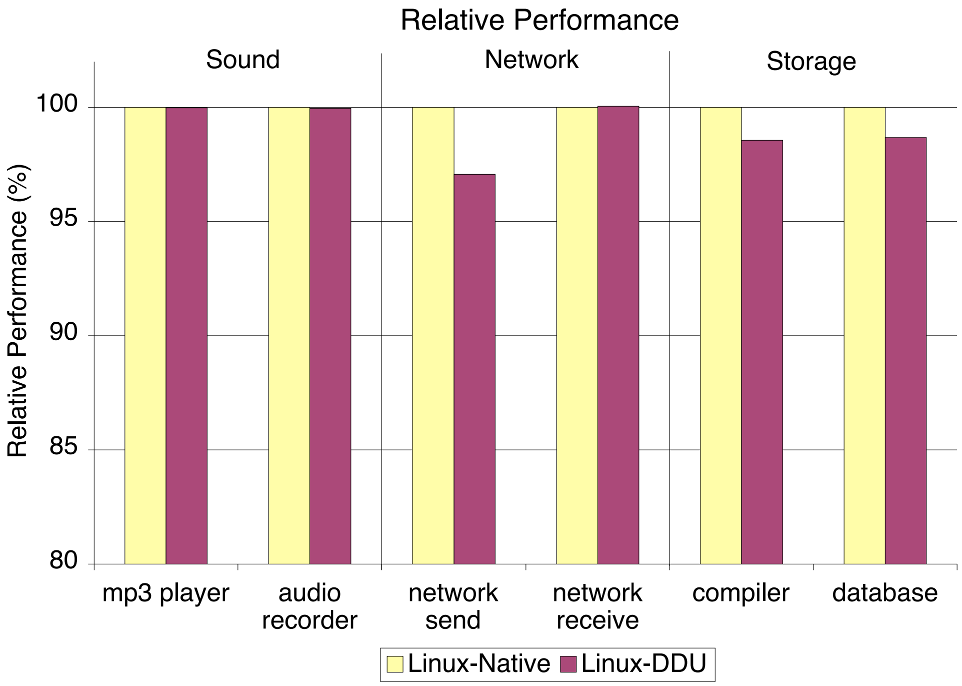

Figure 7.3: Comparative application performance on *Linux-DDU* relative to *Linux-Native*. The X-axis crosses at 80%.

packets per second, causing 45,000 packets to be tracked. While throughput decreases only slightly, the additional work increases the CPU overhead per packet by 34%, from on 3μs on *Linux-Native* to 4μs on *Linux-DDU*.

Another important aspect of performance is the duration of time when the driver is not available during an update. For each of the three drivers, I measured the delay from when the shadow starts impersonating the device driver until it enables the new driver. The delay for the *e1000* and *ide-disk* drivers was approximately 5 seconds, almost all due to the time the driver spends probing hardware[1]. The *emu10k1* updates much faster and is unavailable for only one-tenth of a second. In contrast, rebooting the entire system can take minutes when including the time to restart applications.

[1] The *e1000* driver is particularly slow at recovery. The other network drivers I tested restarted in less than a second.

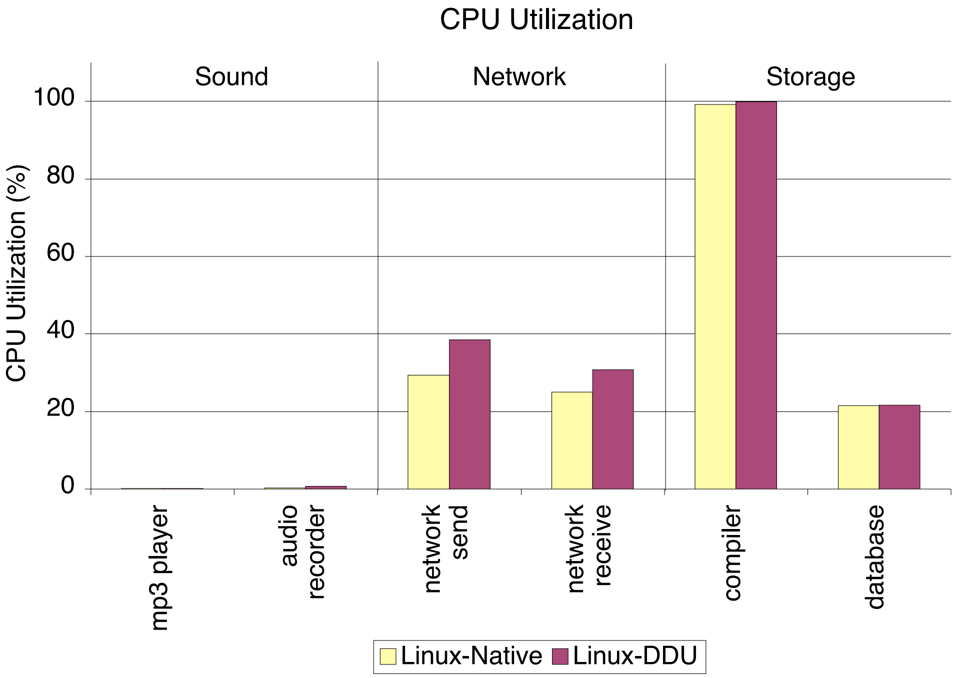

Figure 7.4: Absolute CPU utilization on *Linux-DDU* and *Linux-Native*.

7.5 Summary

While operating systems themselves are becoming more reliable, their availability will ultimately be limited by scheduled maintenance required to update system software. This chapter presented a system for updating device drivers on-line that allows critical system drivers, such as storage and network drivers, to be replaced without impacting the operating system and applications, or more importantly, availability.

Dynamic driver update demonstrates that shadow drivers and Nooks can be a platform for additional reliability services. The update system builds on this platform to update driver code in-place with no changes to the driver itself. The system loads the new driver code, reboots the driver, and transfers kernel references from the old driver to the new driver. To ensure that applying an update will not reduce reliability due to changes in the driver, dynamic driver update checks the new driver for compatibility, and can rollback incompatible updates. In testing, I found that dynamic driver update could apply 99% of the existing driver updates and had minimal impact on performance.

Thus, this system reduces the downtime caused by driver maintenance.

Chapter 8

LESSONS

8.1 Introduction

In this chapter, I present lessons from the process of designing and implementing Nooks. While I built Nooks within Linux, I also advised several undergraduate students on an effort to implement the Nooks architecture within Windows 2000. The Windows implementation was not completed, but the effort highlighted differences between the Linux and Widows kernels and provided valuable insight into how an operating system's structure affects its reliability.

To execute device drivers reliably, Nooks is involved on every event that impacts drivers. These events may be explicit, such as function calls between a driver and the kernel, or implicit, such as changes to the kernel address space. The ease of implementing a reliability layer depends largely on the ease of identifying these events and injecting code when they occur. Thus, OS structure has an impact on the ease of tolerating driver failures. At a high level, when relevant events are more frequent, the performance cost of fault tolerance rises. Similarly, when the difficulty of detecting or hooking events rises, the complexity of providing fault tolerances rises.

The Linux and Windows operating systems differ greatly in their internal organization and structure. Many of these differences have a large impact on the ease of supporting fault-tolerant execution of drivers. The differences between the two OSs thus provide a lens for examining these structural impacts. In this chapter, I discuss the lessons learned about the relationship between OS structure and reliability from the two Nooks implementation efforts.

8.2 Driver Interface

The interface between the kernel and drivers affects the ease of isolating drivers and recovering from their failure. The impact comes from the size, complexity and organization of the kernel-driver calling interface. Smaller interfaces, with fewer functions, less-complex parameters, and explicit communication reduce the coding effort to isolate drivers.

At the most basic level, every function in the driver interface requires a wrapper and object tracking code. Thus, the amount of code is proportional to the size and complexity of the driver interface. For example, the NDIS interface for Windows network drivers contains 147 functions that drivers may implement; it therefore requires more coding effort than the network interface in Linux, which defines just 21 functions. However, other factors beyond interface size determine the total complexity of implementing this code.

Nooks must interpose on all communication between drivers and the kernel to contain driver failures. Interposition is straightforward when every act of communication is explicit, through function calls and their parameters. The code to synchronize objects across protection domains then can be confined to parameters passed through the kernel-driver interface. In contrast, communication through shared memory requires additional code to detect communication channels and to detect when communication occurs. For example, Linux provides a function for accessing the memory of a user-mode process. Nooks can interpose on the function and provide safe access to the user-mode data by XPCing into the kernel, which maps the calling process' address space. In contrast, Windows drivers directly access user-mode pointers. Nooks must detect this access as a communication channel and not as a bad-pointer access. Thus, this access requires that Nooks map the user-mode address space of client processes into driver protection domains, which is a more complex and expensive approach than the XPC-based solution on Linux.

Isolation is also simplified when the kernel-driver interface relies on abstract data types (ADTs), because all access to an ADT goes through accessor functions. These functions provide two benefits. First, they are an explicit signal that the kernel and driver are communicating, providing an opportunity to interpose on that communication. Second, accessor functions allow the choice of performing the operation locally or remotely in a different protection domain. For infrequently modified objects, it is often faster to modify shared objects in the context of the kernel instead of creating a local copy of an object, while for frequently modified objects a copy may be faster.

As previously discussed, opaque pointers in the kernel-driver interface complicate fault-tolerance because they may point to a complex data structure not understood by wrappers and shadow drivers. While isolation may be possible, without knowledge of the semantics of opaque parameters, transparent recovery is not. For example, a shadow driver cannot know whether a request is idempotent.

The frequency of opaque data types varies widely between driver classes and operating systems. In Linux, most drivers that provide a common service, such as block storage or network access, implement a common interface with no opaque or driver-specific parameters. Instead, drivers export their additional capabilities through the `/proc` file system interface, which does not use opaque pointers. In contrast, Windows drivers provide extended capabilities through the `ioctl` and `fsctl` interfaces, which use opaque pointers.

Finally, the complexity of the driver interface itself is a major source of bugs. If driver writers must concern themselves with low-level details of synchronization and memory management, then bugs are more likely. If, instead, the interface provides high-level abstractions that hide this complexity, then driver code is simpler and less error-prone. For example, the Windows Driver Model requires complex and error-prone synchronization logic for canceling a pending I/O request [145]. The driver must remove the I/O request from a queue while ensuring that it is not completed by another thread. Unfortunately, this complex interface, provided to support high-performance I/O, often leads to driver bugs that reduce reliability [13]. In contrast, the Linux driver interface has a much simpler design with many fewer functions. Furthermore, the kernel performs much of locking before invoking device drivers, so that drivers need not synchronize access to shared data structures. Rather than require driver writers to implement complex cancellation logic, Linux relies on the Unix signal mechanism to cancel an I/O request. Thus, simpler interfaces, while potentially reducing performance for high-bandwidth and low-latency devices, may substantially improve the reliability of the system.

8.3 Driver Organization

The organization of drivers in the kernel and their communication patterns also affect the ease of tolerating their failure. When drivers only communicate with the kernel, then the complete semantics of the driver interface must be understandable to the kernel. Hence, these semantics can be used to produce wrapper, object tracking, and recovery code. In contrast, if drivers communicate with other drivers, additional code is needed for every driver-to-driver communication channel, which reduces the leverage provided by the class structure of drivers.

Linux and Windows differ in their organization of drivers, and this impacts the ease of isolating individual drivers. Linux uses a library model for drivers, in which drivers link with a support library containing common routines and then export a single interface to the kernel. There is no support for extending the driver capabilities with additional code, so there is no need to isolate cooperating drivers from each other. In the Linux model, the driver and the library are isolated together in a single protection domain.

In contrast, Windows employs a *stacked* driver model, in which an I/O request may pass through a sequence of drivers before delivery to a device. Each driver in the layer may provide additional service, such as RAID for storage systems, or common abstractions, such as Ethernet service for network devices. In addition, the logical driver for a single device may be split into two separate drivers within the kernel, a high-level driver for interfacing with applications and a low-level driver for interfacing with hardware. The benefit of the stacked model is that it simplifies extending the capabilities of a device by inserting a *filter* driver between two layers in the stack. The difficulty of stacked drivers, though, is that it encourages direct driver-to-driver communication. Drivers may pass opaque pointers, including function pointers, amongst themselves. Without knowledge of the types of these pointers, an isolation system cannot interpose on all communication or provide call-by-value-result semantics. Hence, it cannot isolate the drivers from each other. In this model, either the entire stack of drivers must be isolated in a single domain, or additional code is required for each layer of the stack.

8.4 Memory Management

Much of the Nooks isolation code pertains to memory management. For example, the kernel invokes Nooks whenever the kernel address space changes, in order to synchronize the page tables used by lightweight kernel protection domains with the kernel page table. The object tracking code, to provide call-by-value-result semantics, must be aware of how objects shared by drivers and the kernel are allocated. As a result, how an operating system manages memory has a large impact on the ease of isolating drivers.

Nooks provides isolation with lightweight kernel protection domains that map the kernel address space with reduced access privileges. In a kernel that is not paged, the kernel address space changes only when virtual mappings are needed to create large, contiguous memory blocks. Hence, when the kernel page table changes, the shadow tables can be synchronized without affecting performance. In a pageable kernel, such as Windows, changes to the address space are frequent, which leads to higher costs to manage shadow page tables. In Linux, which is not paged, I chose to keep the shadow and real page tables tightly synchronized. In a pageable kernel it may be more efficient to update driver page tables on demand, because drivers access just a small subset of the entire kernel address space.

The organization of the kernel memory management code itself also impacts the difficulty of implementing lightweight kernel protection domains. To maintain tight synchronization of pages tables with the kernel, Nooks must be notified whenever the kernel changes its page table. In Linux, a small set of pre-processor macros manipulate the processor page tables. As a result, it is straightforward to identify locations in the source code where the kernel updates page tables and to insert synchronization code at this point. In contrast, Windows directly modifies hardware page tables through union variables. As others have experienced [17], this design complicates the task of identifying all code locations that manipulate page table contents. In our Windows implementation, we chose to synchronize page tables when the TLB was flushed instead of tediously identifying these locations.

The allocation patterns for data shared between the kernel and drivers also affect the ease of isolation. The simplest pattern to isolate is when the kernel allocates a data object, passes it to the driver, and the driver notifies the kernel when it is done accessing the object. In this pattern, the life cycle of the object is clear: it has a clear birth (when it is allocated), use, and death (when it is released). In contrast, it is more difficult to accurately track objects that are allocated on the extension's stack. For example, Linux drivers pass stack-allocated objects to the kernel when waiting for an event to occur. The object has no explicit birth, because it is created on the stack, and no explicit death, because it dies when the stack is popped. Hence, it is difficult to track these objects accurately. Nooks must instead track the lifetimes of these objects based on the kernel's use of the object; it creates the object when it is first passed to the kernel and deletes it when the kernel releases it. This may be overly conservative, as the driver may reuse the same object multiple times. An analogous problem occurs in the Windows kernel. While Windows provides routines for allocating and destroying I/O request packets (IRPs), drivers are also free to use their own allocation routines. It is therefore difficult to detect whether an IRP is valid and when an IRP is dead, because a driver may free it with many different routines. The object tracker conservatively frees the IRP whenever it is unshared, which leads to unnecessary adding and removing of the same request packets from the tracker.

The Nooks reliability system isolates drivers through memory protection. As a result, the complexity of Nooks is strongly related to the complexity its hosting operating system's memory management code. When the events of interest to Nooks, such as address space changes and object creation and deletion, are explicit, then the isolation code is straightforward and efficient. When these events are difficult to detect, then the isolation code is more complex and more costly. Furthermore, the performance of Nooks depends on the frequency of such events. If address space changes are frequent, as in paged kernels, then maintaining synchronized pages tables may become prohibitively expensive and alternative isolation mechanisms, such as software fault isolation [205], become more attractive.

8.5 Source Code Availability

The availability of driver source code proved important for two reasons. First, driver source code provided details about the kernel-driver interface that are not documented, such as how drivers manipulate kernel data structures. Second, the open-source development model encouraged Linux to evolve towards a cleaner kernel-driver interface, which facilitates interposition, isolation and recovery.

In implementing Nooks, I analyzed the source code from many device drivers. While device driver code can potentially call *any* kernel function and manipulate *any* kernel data structure, this analysis showed that, in practice, driver code is much more restrained. For example, of the 76 fields in the `net_device` structure for network device drivers, the drivers only write to a single field after initialization. As a result of this analysis, the wrappers and object tracker avoid synchronizing the other 75 fields on every entry and exit from a network driver. While this analysis may be possible without extension source code, it would be far more difficult and error prone.

The source-code analysis was a snapshot of the behavior of a small number of drivers, and may not remain valid as the kernel and drivers evolve. Once Nooks is incorporated into the kernel, though, its existence pressures driver writers to follow Nooks assumptions: drivers that violate the rules cannot be isolated or recovered automatically. Thus, in order to ensure that developers conform to Nooks' assumptions, it is crucial that Nooks achieve a critical mass early as a mandatory component of the kernel and not an optional patch.

The open-source development model, in which driver code is available to be read and modified by kernel developers, also had an impact on Nooks. From reading driver code across multiple kernel versions, I observed a common pattern concerning the evolution of the kernel-driver interface. In early versions of the kernel, many drivers directly modified kernel data structures, for example changing flag fields in the current task structure. In later kernels, these accesses were codified into pre-processor macros and inline functions. A simple change to the Linux header files therefore allowed me to interpose on these functions and isolate the drivers with no changes to their source code.

I believe that this process of abstracting common programming patterns is a positive feature of open-source development. First, the driver code that accessed the kernel directly was visible to kernel programmers and led them to extend the kernel interface. Thus, drivers use fewer so-called "hacks" to access kernel resources, because kernel developers can formalize these hacks and add them to the kernel-driver interface. Second, when modifying the kernel, developers could change both the interface and the affected driver code. Thus, the open-source model allows a short feedback loop between kernel and driver developers and leads to a cleaner interface, subject to fewer work-arounds. In a closed-source environment, in contrast, the kernel developers may have little knowledge of what driver writers are doing and have few means to encourage driver writers to take advantage of new interfaces [155].

The availability of driver source code to kernel developers improves reliability by allowing kernel developers to understand, at many levels, how drivers work and what drivers do. This leads to cleaner kernel-driver interfaces that are subject to fewer work-arounds, which in turn lead to drivers that are simpler to isolate and recover.

8.6 Summary

In summary, adding a reliability layer to an existing operating system requires injecting code on all code paths that affect drivers. When these code paths are clearly identified, for example by explicit function invocations, and easy to modify, then implementing a reliability subsystem can be straightforward. If these events are widely distributed, difficulty to find, and difficult to detect, then it is more difficult to tolerate driver failures.

Reliability also requires understanding the properties of the code that can fail, to tolerate the failure and recover. When driver code is available, a reliability layer can make accurate assumptions about the behavior of driver code. In addition, this code enables kernel developers to evolve the kernel-driver interface in directions that simplify fault tolerance.

Chapter 9

FUTURE WORK

9.1 Introduction

While Nooks, shadow drivers, and dynamic driver update proved effective at improving system reliability and availability, there are many opportunities for future work. These opportunities fall into two broad categories. First, the existing limitations of Nooks can be lifted or lessened and its performance improved. For example, better analysis of device driver and extension code may remove many of the compatibility problems mentioned in previous chapters. Second, new device driver interfaces and architectures can lessen the substantial problems caused by device drivers. Drivers must be rewritten for each operating system kernel and modified for each revision or update of device hardware. As a result, research operating system are often limited by the availability of drivers. New driver architectures can lessen these problems by substantially reducing the dependency of driver code on the hardware and the hosting OS.

9.2 Static Analysis of Drivers

The Nooks architecture makes strong assumptions about driver behavior. A driver that violates these assumptions, while potentially bug-free, cannot run with Nooks. For example, wrappers allow drivers to modify only a subset of the parameters they receive. A driver may fail at run-time if it attempts to write to a read-only kernel object or if wrappers do not propagate changes it made to a kernel object. In addition, shadow drivers require that device drivers implement the class interface strictly and not store extra internal state based on device or kernel inputs. If a driver violates these assumptions, then it may fail at run-time or may not recovery correctly.

These compatibility problems raise two opportunities to reduce the reliability impact of correct code that Nooks cannot currently execute safely. First, tools and techniques that verify that a driver is compatible with Nooks' assumptions can ensure that all failures detected at run-time are real and not caused by driver incompatibilities. Static analysis tools that operate on source code [68] or binaries [12] could model check drivers against Nooks' assumptions, such as which parameters are read-only and which are read-write. By applying this check at install time, the system could avoid isolating incompatible drivers.

The second opportunity for future work is to increase the range of drivers that are compatible with Nooks. The existing Nooks implementation is based on a hand-analysis of a small number of drivers. While this approach was sufficient for the drivers tested, automated analysis of a much larger body of drivers would provide a more accurate model of driver behavior, and therefore allow Nooks to support a wider variety of drivers. In addition, with automated analysis, it may be possible to create multiple models of drivers and find the best fit, i.e., one that minimizes the dynamic overhead of copying data between protection domains. Again, static analysis can provide much of the necessary information about how drivers use kernel objects. This analysis could also lead to automated generation of wrapper, object-tracking.

A final use of static analysis techniques is to derive the interface and abstract state machine of a driver automatically from its source code. Recent work on deriving interfaces from source code [3] and on deriving interface properties from their implementations [69] could be applied to drivers. Once found, this information can be used to model check new drivers and to generate shadow driver code automatically.

9.3 Hybrid Static/Run-Time Checking

Nooks takes a purely fault-tolerance approach to reliability. Static analysis and safe languages provide many, but not all, of the same benefits as Nooks. For example, the Vault language [58] can verify statically that locking protocols are implemented correctly. Type-safe languages, such as Java [88] and C♯ [99], ensure that code does not access invalid memory addresses. To date, though, these techniques do not ensure that the driver provides the correct service, for example that it correctly interfaces with hardware. Thus, even with static analysis of driver code, Nooks' run-time error detection and recovery services are still required for high availability.

Nonetheless, Nooks' complexity and run-time overhead can be lessened by integrating these fault avoidance techniques to reduce the amount of run-time checking. For example, drivers verified statically do not require the memory isolation provided by lightweight kernel protection domains. In addition, lock misuse need not be checked for drivers that have been verified to correctly implement locking protocols. Thus, the run-time cost and complexity of Nooks can be reduced by checking properties statically instead of at run-time.

9.4 Driver Design

Existing driver and extension architectures optimize for high performance, flexibility, and ease of programming. For example, the Windows Driver Model supports three different memory management models, allowing driver writers to balance performance and complexity [158]. The MacOS I/O Kit provides an object-oriented framework for developing drivers, based on driver-class-specific base classes, that reduces the amount of code needed for a specific driver [6].

However, these driver and extension architectures are not designed for reliability. First, they ignore the cost of invoking an extension, a fundamental cost in any isolation or recovery system. Second, they share kernel data at very fine granularity, complicating isolation. Third, they allow opaque data to be passed between extensions, which prohibits marshaling or verifying of the data contents. Finally, existing driver architectures require that all driver code run in the kernel, even if it performs no privileged operations. For example, much of the code in network device drivers concerns initialization and cleanup, error handling, and statistics gathering. None of these functions are performance critical, and many of them require no privileged access to kernel data structures. Hence, they need not execute with kernel privilege.

There several opportunities to rectify these limitations of the kernel-extension interface. First, all invocations of extensions should be treated *as if* they were remote invocations, so that the parameters are fully typed and could be marshaled if necessary. Second, the interface to the driver should explicitly recognize the cost of invoking drivers and allow batching and asynchronous execution. Third, driver code should be factored into code that must run in the kernel, for either performance or privilege reasons, and code that need not. For example, driver initialization, cleanup, error handling, and management code could run in a user-mode process instead of in the kernel. Thus, code in the kernel that can potentially destabilize the system is kept to a minimum

Finally, there are opportunities for reducing the complexity of drivers that have not yet been fully exploited by existing operating systems. While most operating systems implement abstractions for devices that provide a common service, such as Ethernet drivers, or with a common command protocol, such as SCSI disks, they do not take advantage of common hardware command interfaces. For example, most network cards present a ring-buffer abstraction to their drivers. When sending a packet, the driver places a pointer to the packet in the send ring and then signals the device that it has updated the ring. Currently, every network driver re-implements the code to receive a packet from the kernel, place it in the ring, and notify the device. The major difference between drivers is the exact data format of the ring, such as the structure of the entries, and the commands for notifying the device of new data. As a result, it may be possible to simplify drivers by providing abstractions of hardware data structures, such as ring buffers, in the driver interface. Rather than writing the entire send and receive path, the driver writer would instead provide a description of the ring data structure and the command sequence. The OS can provide the rest of the code. The I/O system for Mach 3.0 attempted a similar decomposition to abstract device-independent code from drivers [82]. The benefit of this approach is that it greatly simplifies the code for performing I/O, reducing the cost of writing drivers and the risk of driver bugs.

An important aspect of evolving the driver interface is automatic migration from existing drivers to the new framework. Recent work, such as the Ivy project for replacing C as a systems program language [34], and Privtrans for automatically partitioning code for privilege separation [35], points towards possible solutions to this problem.

There have been two recent projects that reinvent the driver interface. Project UDI (Uniform Driver Interface) seeks to create a common driver interface across all Unix platforms [169]. The design calls for fully asynchronous code with no blocking and no direct access to hardware. Instead, blocking calls take callback functions and drivers access hardware though mini-programs written in a domain-specific language. The Windows Driver Foundation seeks to simplify Windows device drivers by providing a stripped-down calling interface and an interface definition language for driver parameters [146]. Both efforts are a step in the right direction, but so far their adoption has been hampered by lack of compatibility with existing driver code.

9.5 Driverless Operating Systems

Beyond redesigning drivers, it is possible to imagine operating systems that do away with device drivers altogether. Even with an improved interface, drivers would continue to pose problems: they must be updated for every device revision, they must be ported to every operating system, and they are critical to performance and reliability. As a result, drivers will continue to hamper system development unless a new solution is found.

Drivers today serve as a combination of a translation layer, connecting the operating system interface to the device interface, an adaptation layer, masking the impedance mismatch between processor and device speeds, and a service layer, providing functionality beyond the device's abilities. However, these three roles may be filled by different software that need not suffer from driver problems. Existing safe-extension technology can provide the service layer functionality, but new communication abstractions are needed for the other two functions.

The job of interfacing to device hardware is necessary because higher-level communication abstractions do not exist for drivers. Elsewhere in operating system design, these abstractions have simplified application design. For example, remote procedure call systems remove much of the complexity in implementing client-server communication. In contrast, many drivers still access devices by reading and writing memory-mapped device registers. While new device connection technologies, such as FireWire [62] and USB [49], allow drivers and devices to communicate with messages, the richness of distributed programming techniques is not yet available to drivers.

To address this issue, new communications mechanisms that take into account the needs of devices, such as low latency, high bandwidth, and direct access to memory, are needed. These mechanisms must incorporate the costs of communication and the relative speeds of the device and the host processor to serve the impedance-matching role of today's drivers. While most existing approaches to programming distributed systems do not apply to device drivers directly, the abstractions, lessons, and approaches can greatly simplify driver design. For example, Devil's use of interface definition languages replaces much of the handcrafted code in drivers that communicates with devices [142]. High-performance messaging abstractions, such as Active Messages [203] may also apply.

An important benefit of high-level communication protocols is that it pushes driver functionality out of the operating system, where it impacts system reliability and must be ported or reimplemented for each additional OS, onto the hardware itself. Thus, much of the code in drivers need be implemented just once, on the driver hardware, rather than separately for each operating system.

With these techniques, operating systems will not need drivers as we conceive of them today. Instead, the function of drivers can be shifted to user-level services, application libraries, and hardware devices. Systems will invoke driver services over a common communication substrate, simplifying the OS and improving its reliability. The code that currently constitutes drivers will be easier to update as new devices are released, because it relies on high-level mechanisms instead of low-level device access. Finally, this code can be ported between operating systems more easily because it can be executed outside of the kernel, as an application library or user-level service.

9.6 Summary

There are two major directions to pursue future research. First, the existing architecture of Nooks can be extended to cover a greater fraction of kernel extensions and to incorporate static checking as a substitute for run-time checking. With these technologies, Nooks isolation and recovery services could be applied to other extensible systems that depend on extension correctness for reliability.

Second, the lessons from implementing Nooks point toward new considerations for device driver design. A new interface to drivers, tailored to reliability concerns, could greatly reduce driver-induced failures. Longer term, new hardware designs and new communication mechanisms could lead to operating systems without device drivers as we know them.

Chapter 10

CONCLUSIONS

Reliability has become a critical challenge for commodity operating systems as we depend on computers for more of our daily lives. The competitive pressure on these systems and their huge installed base, though, prevents the adoption of traditional fault-tolerance techniques.

This thesis presents a new approach to improving the reliability of operating systems that is at once efficient and backwards compatible. Rather than tolerate all possible failures, my approach is to target the most common failures and thereby improve reliability at very low cost. In today's commodity operating systems, device driver failures are the dominant cause of system failure.

In this thesis, I described Nooks, a new reliability layer intended to prevent drivers from forcing either the OS or applications to restart. Nooks uses hardware and software techniques to isolate device drivers, trapping many common faults and permitting extension recovery. Shadow drivers ensure that the OS and applications continue to function during and after recovery. Dynamic driver update ensures that applications and the OS continue to run when applying driver updates.

The Nooks system accomplishes these tasks by focusing on *backward compatibility*. That is, Nooks sacrifices complete isolation and fault tolerance for compatibility and transparency with existing kernels and drivers. Nevertheless, Nooks demonstrates that it is possible to realize an extremely high level of operating system reliability with low performance lost for common device drivers. My fault-injection experiments show that Nooks recovered from 99% of the faults that caused native Linux to crash. Shadow drivers kept applications running for 98% of the faults that would otherwise have caused a failure. In testing, dynamic driver update could apply 99% of driver updates without rebooting the system or restarting applications. Furthermore, the performance cost of Nooks averaged only 1% across common applications and drivers.

My experience shows that: (1) implementation of a Nooks layer is achievable with only modest engineering effort, even on a monolithic operating system like Linux, (2) device drivers can be isolated without change to driver code, and (3) isolation and recovery can dramatically improve the system's ability to survive extension faults.

Looking forwards, Nooks demonstrates that reliability should be a first-class citizen when designing extension interfaces. In addition to performance and programmability, interfaces should be designed to enable efficient isolation and recovery by exposing key events and establishing clear boundaries between extensions and the core system. Only when this lesson has been learned will extensible systems, such as operating systems, browsers, and web servers, achieve high reliability.

BIBLIOGRAPHY

[1] Edward N. Adamas. Optimizing preventive maintenance of software products. *IBM Journal of Research and Development*, 28(1):2–14, January 1984.

[2] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *Proceedings of the 14th USENIX Security Symposium*, pages 287–302, Baltimore, Maryland, August 2005.

[3] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109, Long Beach, California, January 2005.

[4] Scott Maxwell an Frank Hartman. Linux and JPL's mars exploratin rover project: Earth-based planning, simulation, and really remote scheduling. USENIX Annual Technical Conference Invited Talk, April 2005.

[5] Apache Project. Apache HTTP server version 2.0, 2000. Available at `http://httpd.apache.org`.

[6] Apple Computer. I/O kit fundamentals, 2005. Available at `http://developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/IOKitFundamentals.pdf`.

[7] Hitoshi Araki, Shin Futagami, and Kaoru Nitoh. A non-stop updating technique for device driver programs on the IROS platform. In *IEEE International Conference on Communications*, pages 18–22, Seattle, WA, June 1995.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems*, pages 33–40, Schloss Elmau, Germany, May 2001.

[9] Sandy Arthur. Fault resilient drivers for Longhorn server. Microsoft WinHec 2004 Presentation DW04012, May 2004.

[10] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[11] Özalp Babaoğlu. Fault-tolerant computing based on Mach. In *Proceedings of the USENIX Mach Symposium*, pages 185–199, Burlington, VT, October 1990.

[12] Gogul Balakrishnan, Radu Gruian, Thomas R. Reps, and Tim Teitelbaum. CodeSurfer/x86 – a platform for analyzing x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 250–254, Edinburgh, Scotland, April 2005.

[13] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.

[14] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, OR, January 2002.

[15] Roger Barga, David Lomet, German Shegalov, and Gerhard Weikum. Recovery guarantees for internet applications. *ACM Transactions on Internet Technology*, 2004.

[16] Roger Barga, David Lomet, and Gerhard Weikum. Recovery guarantees for general multi-tier applications. In *International Conference on Data Engineering*, pages 543–554, San Jose, California, 2002. IEEE.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, New York, October 2003.

[18] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec♯ programming system: An overview. In *CASSIS International Workshop*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Marseille, France, March 2004. Springer-Verlag.

[19] Joel F. Bartlett. A NonStop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, Pacific Grove, California, December 1981.

[20] Elisa Batista. Windows to power ATMs in 2005. *Wired News*, September 2003. `http://www.wired.com/news/technology/0,1282,60497,00.html`.

[21] Andrew Baumann and Gernot Heiser. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 297–291, Anaheim, California, April 2005.

[22] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop advanced architecture. In *Proceedings of the 2005 IEEE International Conference on Dependable Systems and Networks*, pages 12–21, Yokohama, Japan, June 2005.

[23] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[24] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[25] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[26] Common Criteria Implementation Board. Common criteria for information technology security evaluation, part 3: Security assurance requirements, version 2.2. Technical Report CCIMB-2004-01-003, National Institute of Standards and Technology, January 2004.

[27] Aniruddha Bohra, Iulian Neamtiu, Pascal Gallard, Florin Sultan, and Liviu Iftode. Remote repair of operating system state using Backdoors. In *Proceedings of the International Conference on Autonomic Computing*, pages 256–263, New York, New York, May 2004.

[28] Anita Borg, Wolfgang Balu, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[29] David A. Borman. Implementing TCP/IP on a Cray computer. *ACM SIGCOMM Computer Communication Review*, 19(2):11–15, April 1989.

[30] Luciano Porto Borreto and Gilles Muler. Bossa: A language-based approach for the design of real time schedulers. In *Proceedings of the 14th Euromicro Conference on Real Time Systems*, Paris, France, March 2002.

[31] Daniel P. Bovet and Marcho Cesati. *Understanding the Linux Kernel, 2nd Edition*. O'Reilly Associates, December 2002.

[32] Thomas C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Proceedings of the 28th Symposium on Fault-Tolerant Computing*, pages 128–137, Munich, Germany, June 1998. IEEE.

[33] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[34] Eric Brewer, Jeremy Condit, Bill McCloskey, and Feng Zou. Thirty years is long enough: Getting beyond C. In *Proceedings of the Tenth IEEE Workshop on Hot Topics in Operating Systems*, Santa Fe, New Mexico, June 2005.

[35] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, pages 57–72, San Diego, California, August 2004.

[36] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems*, pages 125–132, Schloss Elmau, Germany, May 2001.

[37] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, Louisiana, February 1999.

[38] Subhachandra Chandra and Peter M. Chen. How fail-stop are faulty programs? In *Proceedings of the 28th Symposium on Fault-Tolerant Computing*, pages 240–249, Munich, Germany, June 1998. IEEE.

[39] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 12–25, Copper Mountain Resort, Colorado, December 1995.

[40] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[41] L. Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the 8th Symposium on Fault-Tolerant Computing*, pages 3–9, Toulouse, France, June 1978.

[42] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson Jim Lloyd, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proceedings of the 2004 USENIX Symposium on Network Systems Design and Implementation*, pages 309–322, San Francisco, California, March 2004.

[43] Peter M. Chen and Brian Noble. When virtual is better than real. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems*, pages 133–138, Schloss Elmau, Germany, May 2001.

[44] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 140–153, Kiawah Island Resort, South Carolina, December 1999.

[45] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alberta, October 2001.

[46] Jörgen Christmansson and Ram Chillarege. Generation of an error set that emulates software faults - based on field data. In *Proceedings of the 26th Symposium on Fault-Tolerant Computing*, pages 304 – 313, Sendai, Japan, June 1996. IEEE.

[47] Yvonne Coady and Grigor Kiczales. Back to the future: A retroactive study of aspect evolution in operating systems code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 50–59, Boston, Massachusetts, March 2003.

[48] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. Performance effects of architectural complexity in the Intel 432. *ACM Transactions on Computer Systems*, 6(3):296–339, August 1988.

[49] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Universal serial bus (USB) 2.0 specification, April 2000. Available at `http://www.usb.org/developer/docs`.

[50] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, California, USA, June 2003.

[51] Fernando J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 27, pages 185–196, Las Vegas, Nevada, 1965. Spartan Books.

[52] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Associates, February 2005.

[53] Coverity. Anaylsis of the Linux kernel, 2004. Available at `http://www.coverity.com`.

[54] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[55] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[56] D. H. Brown Associates. Stratus Introduces Four-Way, Fault-Tolerant SMP Server with Hyper-Threading for Windows 2000 Advanced Server, September 2002. Available at `http://www.stratus.com/resources/pdf/brown092002.pdf`.

[57] Joseph Dadzie. Understanding software patching. *ACM Queue*, 3(2), March 2005.

[58] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001.

[59] Peter J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, December 1976.

[60] Jack B. Dennis and Earl Van Horn. Programming semantics for multiprogramming systems. *Communications of the ACM*, 9(3), March 1966.

[61] David J. DeWitt. The Wisconsin benchmark: Past, present, and future. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems, Second Edition*, pages 269–315. Morgan Kaufmann, 1993.

[62] Mindshare Inc. Don Anderson. *FireWire System Architecture: IEEE 1394A*. Addison-Wesley Professional, 1999.

[63] R. H. Doyle, R. A. Meyer, and R. P. Pedowitz. Automatic failure recovery in a digital data processing system. *IBM Journal of Research and Development*, 3(1):2–12, January 1959.

[64] Richard P. Draves. The case for run-time replaceable kernel modules. In *Proceedings of the Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 160–164, Napa, California, October 1993. IEEE Computer Society Press.

[65] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communications in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, California, October 1991.

[66] Dominic Duggan. Type-based hot swapping of running modules. Technical Report SIT CS Report 2001-7, Stevens Institute of Technology, Castle Point on the Hudson, Hoboken, New Jersey, October 2001.

[67] Gregory Duval and Jacques Julliand. Modeling and verification of the RUBIS μ-kernel with SPIN. In *Proceedings of the International SPIN Workshop*, Montreal, Canada, October 1995.

[68] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, California, October 2000.

[69] Dawson Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Lake Louise, Alberta, October 2001.

[70] Dawson R. Engler. personal communication, March 2005.

[71] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[72] Úlfar Erlingsson, Tom Roeder, and Ted Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-05-82, Microsoft Research, June 2005.

[73] Jean-Charles Fabre, Manuel Rodrí, Jean Arlat, Frédéric Salles, and Jean-Michel Sizun. Building dependable COTS microkernel-based systems using MAFALDA. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, pages 85–94, Los Angeles, California, December 2000.

[74] Robert S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[75] Robert S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 470–476, October 1976.

[76] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[77] Wu-Chun Feng. Making a case for efficient supercomputing. *ACM Queue*, 1(7), October 2003.

[78] Christof Fetzer and Zhen Xiao. HEALERS: A toolkit for enhancing the robustness and security of existing applications. In *Proceedings of the 2003 IEEE International Conference on Dependable Systems and Networks*, pages 317–322, San Francisco, California, June 2003. IEEE.

[79] Charles Fishman. They write the right stuff. *Fast Company*, (6):95–103, December 1996.

[80] Marc E. Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. patch(1) considered harmful. In *Proceedings of the Tenth IEEE Workshop on Hot Topics in Operating Systems*, Santa Fe, New Mexico, June 2005.

[81] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: a substrate for OS language and research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, October 1997.

[82] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach. In *Proceedings of the Usenix Mach Symposium*, pages 163–176, November 1991.

[83] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On- Demand IT Infrastructure*, October 2004.

[84] Michael Gagliardi, Ragunathan Rajkumar, and Lui Sha. Designing for evolvability: Building blocks for evolvable real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 100–109, Boston, Massachusetts, June 1996.

[85] James Gettys, Philip L. Carlton, and Scott McGregor. The X window system version 11. Technical Report CRL-90-08, Digital Equipment Corporation, December 1900.

[86] Al Gillen, Dan Kusnetzky, and Scott McLaron. The role of Linux in reducing the cost of enterprise computing, January 2002. IDC white paper.

[87] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 45–58, Monterey, California, June 2002.

[88] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[89] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 295–304, May 1978.

[90] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, Los Angeles, California, September 1981.

[91] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, Los Angeles, California, January 1996. IEEE.

[92] Jim Gray. Heisenbugs: A probabilistic approach to availability. Talk given at ISAT, availble at `http://research.microsoft.com/~Gray/talks/ISAT_Gray_FT_Avialiability_talk.ppt`, April 1999.

[93] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, September 1991.

[94] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software–Practice and Experience*, 23(9):949–94, September 1993.

[95] Jaap C. Haarsten. The Bluetooth radio system. *IEEE Personal Communications Magazine*, 7(1):28–36, February 2000.

[96] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, New Orleans, Lousiana, February 1999.

[97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schöberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, Saint-Malo, France, October 1997.

[98] Steffen Hauptmann and Josef Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 70–80, May 1996.

[99] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C♯ Programming Language*. Addison-Wesley, Boston, Massachusetts, October 2003.

[100] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 526–538. Springer-Verlag, July 2002.

[101] Scott Herboldt. How to improve driver quality with Winqual/WHQL. Microsoft WinHec 2005 Presentation TWDE05006, April 2005.

[102] Hewlett Packard. Hewlett Packard Digital Entertainment Center, October 2001. `http://www.hp.com/hpinfo/newsroom/press/31oct01a.htm`.

[103] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 13–23, Snowbird, Utah, June 2001.

[104] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Seattle, Washington, 1992.

[105] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 65–76, New Orleans, Louisiana, June 1998.

[106] Gerald J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[107] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th ACM International Symposium on Computer Architecture*, pages 341–348, Minneapolis, Minnesota, May 1981.

[108] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[109] Galen Hunt. Creating user-mode device drivers with a proxy. In *Proceedings of the 1997 USENIX Windows NT Workshop*, Seattle, WA, August 1997.

[110] Galen C. Hunt and James R. Larus. Singularity design motivation (Singularity Technical Report 1). Technical Report MSR-TR-2004-105, Microsoft Research, December 2004.

[111] RTCA Inc. Software considerations in airborne systems and equipment certification, December 1992. RTCA/DO-178B.

[112] Intel Corporation. *The IA-32 Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, January 2002. Available at `http://www.intel.com/design/pentium4/manuals/24547010.pdf`.

[113] Doug Jewett. Integrity S2: A fault-tolerant Unix platform. In *Proceedings of the 21st Symposium on Fault-Tolerant Computing*, pages 512–519, Quebec, Canada, June 1991. IEEE.

[114] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 275–288, Monterey, California, June 2002.

[115] Andreéas Johansson and Neeraj Suri. Error propagation profiling of operating systems. In *Proceedings of the 2005 IEEE International Conference on Dependable Systems and Networks*, pages 86–95, Yokohama, Japan, June 2005.

[116] Rick Jones. Netperf: A network performance benchmark, version 2.1, 1995. Available at `http://www.netperf.org`.

[117] Jungo. Windriver cross platform device driver development environment. Technical report, Jungo Corporation, February 2002. `http://www.jungo.com/windriver.html`.

[118] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videria Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväsklyä, Finland, June 1997. Springer-Verlag.

[119] Mark J. Kilgard, David Blythe, and Deanna Hohn. System support for OpenGL direct rendering. In *Proceedings of Graphics Interface*, pages 116–127, Toronto, Ontario, May 1995. Canadian Human-Computer Communications Society.

[120] Arun Kishan and Monica Lam. Dynamic kernel modification and extensibility, May 2002. `http://suif.stanford.edu/papers/kishan.pdf`.

[121] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[122] Michael Kofler. *The Definitive Guide to mySQL, Second Edition*. Apress, Berkeley, California, October 2003.

[123] Eric J. Koldinger, Jefrey S. Chase, and Susan J. Eggers. Architectural support for single address space operating systems. In *Proceedings of the Fifth ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, Boston, Massachusetts, October 1992.

[124] Sanjeev Kumar and Kai Li. Using model checking to debug device firmware. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 61–74, Boston, Massachusetts, December 2002.

[125] Jean-Claude Laprie, Jean Arlat, Christian Béounes, Kerama Kanoun, and Catherine Hourtolle. Hardware and software fault tolerance: Definition and analysis of architectural solutions. In *Proceedings of the 17th Symposium on Fault-Tolerant Computing*, pages 116–121, Pittsburgh, Pennsylvania, July 1987. IEEE.

[126] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus adn Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Transactions on Software Engineering*, 30(5):92–100, May 2004.

[127] Julia L. Lawall, Gilles Muller, and Richard Urunuela. Tarantula: Killing driver bugs before they hatch. In *Proceedings of the Fourth AOSD Workhop on Aspects, Components, and Patterns for Infrastructure Software*, pages 13–18, Chicago, Illinois, March 2005.

[128] Inhwan Lee and Ravishankar K. Iyer. Software dependability in the Tandem GUARDIAN system. *IEEE Transactions on Software Engineering*, 21(5), May 1995.

[129] Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, Wisconsin, April 1983.

[130] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gö. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, California, December 2004.

[131] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. Available at `http://www.cs.washington.edu/homes/levy/capabook`.

[132] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain Resort, Colorado, December 1995.

[133] Linux Kernel Mailing List. Available at `http://www.uwsg.indiana.edu/hypermail/linux/kernel`.

[134] Bev Littlewood and Lorenzo Strigini. Software reliablity and dependability: a roadmap. In Anthony Finkelstein, editor, *Proceedings of the International Conference on Software Engineering - Future of Software Engineering Track*, pages 175–188, Limerick, Ireland, June 2000.

[135] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 289–304, San Diego, California, October 2000.

[136] David E. Lowell and Peter M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, November 1998.

[137] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the Eleventh ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, October 2004.

[138] LWN.net. Kernel patch page. Available at `http://lwn.net/Kernel/Patches`.

[139] R. E. Lyons and W. Vanderkulk. The use of triple modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 7(2):200–209, April 1962.

[140] Sean McGrane. Windows Server platform: Overview and roadmap. Microsoft WinHec 2005 Presentation TWSE05002, April 2005.

[141] Paul R. McJones and Garret F. Swart. Evolving the unix system interface to support multi-threaded programs. Technical Report SRC Research Report 21, Digital Systems Research Center, September 1987. Available at `http://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-021-html/evolve.html`.

[142] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, California, October 2000.

[143] Microsoft Corporation. Windows Update. Available at `http://www.windowsupdate.com`.

[144] Microsoft Corporation. FAT: General overview of on-disk format, version 1.03, December 2000.

[145] Microsoft Corporation. Cancel logic in windows drivers, May 2003. Available at `http://www.microsoft.com/whdc/driver/kernel/cancel_logic.mspx`.

[146] Microsoft Corporation. Architecture of the windows driver foundation, July 2005. Available at `http://www.microsoft.com/whdc/driver/wdf/wdf-arch.mspx`.

[147] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998. ACM.

[148] Bob Muglia. Microsoft windows server platform: The next three years. Microsoft Professsional Developers Conference Keynote Address, September 2005. Available at `http://www.microsoft.com/presspass/exec/bobmuglia/09-15PDC2005.mspx`.

[149] Gilles Muller, Michel Banâtre, Nadine Peyrouze, and Bruno Rochat. Lessons from FTM: An experiment in design and implementation of a low-cost fault-tolerant system. *IEEE Transactions on Software Engineering*, 45(2):332–339, June 1996.

[150] Brendan Murphy. Fault tolerance in this high availability world. Talk given at Stanford University and University of California at Berkeley. Available at `http://research.microsoft.com/users/bmurphy/FaultTolerance.htm`, October 2000.

[151] Brendan Murphy and Mario R. Garzia. Software reliability engineering for mass market products. *The DoD Software Tech News*, 8(1), December 2004. Available at `http://www.softwaretechnews.com/stn8-1/sremm.html`.

[152] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 2004 USENIX Symposium on Network Systems Design and Implementation*, pages 155–168, San Francisco, California, March 2004.

[153] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 75–88, Boston, Massachusetts, December 2002.

[154] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 580–586, St. Louis, Missouri, May 2005.

[155] Nar Ganapathy, Architect, Windows Device Experience. personal communication, April 2005.

[156] Chandra Narayanaswami, Noboru Kamijoh, Mandayam Raghunath, Tadanobu Inoue, Thomas Cipolla, Jim Sanford, Eugene Schlig, Sreekrishnan Venkiteswaran, and Dinakar Guniguntala. Ibm's linux watch: The challenge of miniaturization. *IEEE Computer*, 35(1), January 2002.

[157] Wee Teck Ng and Peter M. Chen. The systematic improvement of fault tolerance in the Rio file cache. In *Proceedings of the 29th Symposium on Fault-Tolerant Computing*, pages 76–83, Madison, Wisconsin, June 1999. IEEE.

[158] Walter Oney. *Programming the Microsoft Windows Driver Model, Second Edition*. Microsoft Press, Redmond, Washington, December 2002.

[159] Elliott I. Organick. *A Programmer's View of the Intel 432 System*. McGraw Hill, 1983.

[160] Vince Orgovan and Mike Tricker. An introduction to driver quality. Microsoft WinHec 2004 Presentation DDT301, August 2003.

[161] Alessandro Orso, Anup Rao, and Marj Jean Harrold. A technique for dynamic updating for Java software. In *Proceedings of the IEEE International Conference on Software Maintenance 2002*, pages 649–658, Montréal, Canada, October 2002.

[162] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[163] David L. Parnas. The influence of software structure on system reliability. In *Proceedings of the International Converence on Software Engineering*, pages 358–362, April 1975.

[164] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kýcýman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report CSD-02-1175, UC Berkeley Computer Science, March 2002.

[165] Marc C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. The capability maturity model, version 1.1. *IEEE Software*, 10(4):18–27, July 1993.

[166] David Pescovitz. Monsters in a box. *Wired*, 8(12):341–347, December 2000.

[167] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Conference*, pages 213–224, New Orleans, Louisiana, January 1995.

[168] Jon Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[169] Project-UDI. Introduction to UDI version 1.0. Technical report, Project UDI, August 1999.

[170] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1:220–232, June 1975.

[171] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 15–28, Lake Louise, Alberta, October 2001.

[172] Mark Russinovich, Zary Segall, and Dan Siewiorek. Application transparent fault management in Fault Tolerant Mach. In *Proceedings of the 23rd Symposium on Fault-Tolerant Computing*, pages 10–19, Toulouse, France, June 1993. IEEE.

[173] Mark E. Russionvich and David A. Solomon. *Inside Windows 2000*. Microsoft Press, Redmond, Washington, 2000.

[174] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[175] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[176] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 239–253, Pacific Grove, California, October 1991.

[177] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[178] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 83–90, Litchfield Park, AZ, December 1989.

[179] Ephriam Schwartz. Check your oil and your e-mail. *PC World*, September 1999.

[180] Ephriam Schwartz. Linux emerges as cell phone dark horse. *Inforworld*, October 2003. Available at `http://www.infoworld.com/article/03/10/03/39NNphoneos_1.html`.

[181] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 2(10):53–65, March 1993.

[182] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996.

[183] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 827–838, Paris, France, June 2004.

[184] Sharp. Zaurus handheld SL-6000, 2005. `http://www.sharpusa.com/products/TypeLanding/0,1056,112,00.html`.

[185] Takahiro Shinagawa, Kenji Kono, and Takashi Masuda. Exploiting segmentation mechanism for protecting against malicious mobile code. Technical Report 00-02, Dept. of Information Science, University of Tokyo, May 2000.

[186] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, Seventh Edition*. John Wiley & Sons, Inc., March 2004.

[187] Slashdot. Windows upgrade, FAA error cause LAX shutdown. `http://it.slashdot.org/it/04/09/21/2120203.shtml?tid=128&tid=103&tid=201`.

[188] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 141–154, Austin, Texas, June 2003.

[189] Standard Performance Evaluation Corporation. The SPECweb99 benchmark, 1999.

[190] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 1–14, Boston, Massachusetts, June 2001.

[191] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. In *Proceedings of the 21st Symposium on Fault-Tolerant Computing*, pages 2–9, Quebec, Canada, June 1991. IEEE.

[192] Mark Sullivan and Michael Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 171–180, Barcelona, Spain, September 1991. Morgan Kaufman Publishing.

[193] Paul Thurrott. Windows 2000 server: The road to gold, part two: Developing Windows. *Paul Thurrott's SuperSite for Windows*, January 2003.

[194] TiVo Corporation. TiVo digital video recorder, 2001. `www.tivo.com`.

[195] Wilfredo Torres-Pomales. Software fault tolerance: A tutorial. Technical Report Tm-2000-210616, NASA Langley Research Center, October 2000.

[196] Linux Torvalds. Linux kernel source tree. Available at `http://www.kernel.org`.

[197] Jeff Tranter. Open sound system programmer's guide, January 2000. Available at `http://www.opensound.com/pguide/oss.pdf`.

[198] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification – now! In *Proceedings of the Tenth IEEE Workshop on Hot Topics in Operating Systems*, Santa Fe, New Mexico, June 2005.

[199] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *London Math Society, Series 2*, 42(2):230–265, November 1936.

[200] Arjan van de Ven. kHTTPd: Linux HTTP accelerator. Available at `http://www.fenrus.demon.nl/`.

[201] Kevin Thomas Van Maren. The Fluke device driver framework. Master's thesis, Department of Computer Science, University of Utah, December 1999.

[202] Werner Vogels, Dan Dumitriu, Ken Birman Rod Gamache, Mike Massa, Rob Short, John Vert, Joe Barrera, and Jim Gray. The design and architecture of the microsoft cluster service. In *Proceedings of the 28th Symposium on Fault-Tolerant Computing*, pages 422–431, Munich, Germany, June 1998. IEEE.

[203] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th ACM International Symposium on Computer Architecture*, pages 256–266, Queensland, Australia, May 1992.

[204] John von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, Princeton, New Jersey, 1956.

[205] Robert S. Wahbe, Steven E. Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, North Carolina, December 1993.

[206] David A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size, July 2002. Available at `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`.

[207] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 195–209, Boston, Massachusetts, December 2002.

[208] Toh Ne Win and Michael Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report MIT-LCS-TR-841, MIT Lab for Computer Science, May 2002.

[209] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the Tenth ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, October 2002.

[210] William A. Wulf. Reliable hardware-software architecture. *IEEE Transactions on Software Engineering*, 1(2):233–240, June 1975.

[211] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file systems errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 273–288, San Francisco, California, December 2004.

[212] Michael Young, Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, and Avadis Tevanian. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–113, Atlanta, Georgia, June 1986.

VITA

Michael Swift was born and raised in Amherst, Massachusetts. After graduating from Cornell University in 1992, he moved to the Northwest and started working for Microsoft. There he implemented security features in Windows NT 4.0 and Windows 2000, primarily access control and authentication code. However, his lifelong dream was to become a computer science professor. The expiration of his G.R.E. scores in 1996 provided the final kick in the pants that got him out of Microsoft. Michael applied to graduate school that year and started at the University of Washington two years later. He earned a Doctor of Philosophy in Computer Science from the University of Washington in 2005.