Recovering Device Drivers

MICHAEL M. SWIFT, MUTHUKARUPPAN ANNAMALAI, BRIAN N. BERSHAD, and HENRY M. LEVY
University of Washington

This article presents a new mechanism that enables applications to run correctly when device drivers fail. Because device drivers are the principal failing component in most systems, reducing driver-induced failures greatly improves overall reliability. Earlier work has shown that an operating system can survive driver failures [Swift et al. 2005], but the applications that depend on them cannot. Thus, while operating system reliability was greatly improved, application reliability generally was not.

To remedy this situation, we introduce a new operating system mechanism called a *shadow driver*. A shadow driver monitors device drivers and transparently recovers from driver failures. Moreover, it assumes the role of the failed driver during recovery. In this way, applications using the failed driver, as well as the kernel itself, continue to function as expected.

We implemented shadow drivers for the Linux operating system and tested them on over a dozen device drivers. Our results show that applications and the OS can indeed survive the failure of a variety of device drivers. Moreover, shadow drivers impose minimal performance overhead. Lastly, they can be introduced with only modest changes to the OS kernel and with no changes at all to existing device drivers.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms: Reliability, Management

Additional Key Words and Phrases: Recovery, device drivers, I/O

1. INTRODUCTION

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 result in losses in the millions [Feng 2003].

Most of these system failures are caused by the operating system's device drivers. Failed drivers cause 85% of Windows XP crashes [Orgovan and Tricker

This work was supported in part by the National Science Foundation under grants ITR-0085670 and CCR-0121341.

Authors' address: Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195; email: swift@cs.wisc.edu; {muthu,bershad,levy}@cs.washington.edu.

2003], while Linux drivers have up to seven times the bug rate of other kernel code [Chou et al. 2001]. A failed driver typically causes the application, the OS kernel, or both, to crash or stop functioning as expected. Hence, preventing driver-induced failures improves overall system reliability.

Earlier failure-isolation systems within the kernel were designed to prevent driver failures from corrupting the kernel itself [Swift et al. 2005]. In these systems, the kernel unloads a failed driver and then restarts it from a safe initial state. While isolation techniques can reduce the frequency of system crashes, *applications* using the failed driver can still crash. 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.

This article presents a new mechanism, called a *shadow driver*, that improves overall system reliability by concealing 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 restores the failed driver to a state where it can resume processing requests.

Our design for shadow drivers 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) *Recovery logic should be centralized in a single subsystem.* We want to consolidate recovery knowledge in a small number of components to simplify the implementation.
(3) *Driver recovery logic should be generic.* The increased reliability offered by driver recovery should not be offset by potentially destabilizing changes to the tens of thousands of existing drivers. Therefore, the architecture must enable a single shadow driver to handle recovery for a large number of device drivers.
(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 are intended to minimize the cost required to make and use shadow drivers while maximizing their value in existing commodity operating systems.

We implemented the shadow driver architecture for sound, network, and IDE storage drivers on a version of the Linux operating system. Our results show that shadow drivers: (1) mask device driver failures from applications, allowing applications to run normally during and after a driver failure, (2) impose minimal performance overhead for many drivers, (3) require no changes to existing applications and device drivers, and (4) integrate easily into an existing operating system.

This article describes the design, implementation and performance of shadow drivers. The following section reviews general approaches to protecting applications from system faults. Section 3 describes device drivers and the shadow driver design and components. Section 4 presents the structure of shadow drivers and the mechanisms required to implement them in Linux. Section 5 presents experiments that evaluate the performance, effectiveness, and complexity of shadow drivers. The final section summarizes our work.

2. RELATED WORK

This section describes previous research on recovery strategies and mechanisms. The importance of recovery has long been known in the database community, where transactions [Gray and Reuter 1993] prevent data corruption and allow applications to manage failure. More recently, the need for failure recovery has moved from specialized applications and systems to the more general arena of commodity systems [Patterson et al. 2002].

A general approach to recovery is to run application 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 [Jewett 1991], at the hardware-software interface [Bressoud and Schneider 1996], at the system call interface [Babaoğlu 1990; Borg et al. 1989; Bressoud 1998], or at a message passing or application interface [Bartlett 1981]. Shadow drivers similarly 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 our 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 is typically much simpler than the driver it shadows.

Another common recovery approach is to restart applications after a failure. Many systems periodically checkpoint application state [Lowell and Chen 1998; Muller et al. 1996; Plank et al. 1995], while others combine checkpoints with logs [Babaoğlu 1990; Borg et al. 1989; Russinovich et al. 1993]. These systems transparently restart failed applications from their last checkpoint (possibly on another machine) and replay the log if one is present. Shadow drivers take a similar approach by replaying a log of requests made to drivers. Recent work has shown that this approach is limited when recovering from *application* faults: applications often become corrupted before they fail; hence, their logs or checkpoints may also be corrupted [Chandra and Chen 1998; Lowell et al.

2000]. Shadow drivers reduce this potential by logging only a small subset of requests. Furthermore, application bugs tend to be deterministic and recur after the application is restarted [Chandra and Chen 2000]. Driver faults, in contrast, often cause transient failures because of the complexities of the kernel execution environment [V. Orgovan, Systems Crash Analyst, Windows Core OS Group, Microsoft Corp. 2004, personal communication].

Another approach is simply to reboot the failed component, for example, unloading and reloading failed kernel extensions, such as device drivers [Swift et al. 2005]. Rebooting has been proposed as a general strategy for building high-availability software [Candea and Fox 2001]. However, rebooting forces *applications* to handle the failure, for example, reinitializing state that has been lost by the rebooted component. Few existing applications do this [Candea and Fox 2001], and those that do not share the fate of the failed driver. Shadow drivers transparently restore driver state lost in the reboot, invisibly to applications.

Shadow drivers rely on device driver isolation to prevent failed drivers from corrupting the OS or applications. Isolation can be provided in various ways. Vino [Seltzer et al. 1996] encapsulates extensions using software fault isolation [Wahbe et al. 1993] and uses transactions to repair kernel state after a failure. Nooks [Swift et al. 2005] and Palladium [Chiueh et al. 1999] isolate extensions in protection domains enforced by virtual memory hardware. Microkernels [Liedtke 1995; Wulf 1975; Young et al. 1986] and their derivatives [Engler et al. 1995; Ford et al. 1997; Hand 1999] force isolation by executing extensions in user mode.

Rather than concealing driver failures, these systems all reflect a *revealing* strategy, one in which the application or user is made aware of the failure. The OS typically returns an error code, telling the application that a system call failed, but little else (e.g., it does not indicate which component failed or how the failure occurred). The burden of recovery then rests on the application, which must decide what steps to take to continue executing. As previously mentioned, most applications cannot handle the failure of device drivers [Whittaker 2001], since driver faults typically crash the system. When a driver failure occurs, these systems expose the failure to the application, which may then fail. By impersonating device drivers during recovery, shadow drivers conceal errors caused by driver failures, and thereby protect applications.

Several systems have narrowed the scope of recovery to focus on a specific subsystem or component. For example, the Rio file cache [Chen et al. 1996] provides high performance by isolating a single system component, the file cache, from kernel failures. Phoenix [Barga et al. 2002] provides transparent recovery after the failure of a single problematic component type: database connections in multi-tier applications. Similarly, our shadow driver research focuses on recovery for a single OS component type, the device driver, which is the leading cause of OS failure. By abandoning general-purpose recovery, we transparently resolve a major cause of application and OS failure, while maintaining a low runtime overhead.

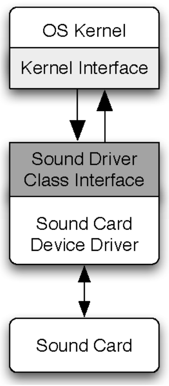

Fig. 1. A sample device driver. The device driver *exports* the services defined by the device's class interface and *imports* services from the kernel's interface.

3. DEVICE DRIVERS AND SHADOW DRIVER DESIGN

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 that provides access to the device, and the kernel interface that drivers *import* from the operating system. For example, Figure 1 shows the kernel calling into a sound 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 sound.

In practice, most device drivers are members of a *class*, which is defined by its interface. For example, all network drivers obey the same kernel-driver interface, and all sound-card drivers obey the same kernel-driver interface. This class orientation simplifies the introduction of new drivers into the operating system, since no OS changes are required to accommodate them.

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

3.1 Driver Faults

Most drivers fail due to bugs that result from unexpected inputs or events [V. Orgovan, Systems Crash Analyst, Windows Core OS Group, Microsoft Corp. 2004, personal communication]. For example, a driver may corrupt a data structure if an interrupt arrives during a sensitive portion of request processing. Device drivers may crash in response to (1) the stream of requests from the kernel, both configuration and I/O, (2) messages to and from the device, and (3) the kernel environment, which may raise or lower power states, swap pages

[1]This paper uses the terms "device driver" and "driver" interchangeably; similarly, we use the terms "shadow driver" and "shadow" interchangeably.

of memory, and interrupt the driver at arbitrary times. A driver bug triggered solely by a sequence of configuration or I/O requests is called a *deterministic* failure. No generic recovery technique can transparently recover from this type of bug, because any attempt to complete an offending request may trigger the bug [Chandra and Chen 2000]. In contrast, *transient* failures are triggered by additional inputs from the device or the operating system and occur infrequently.

A driver failure that is detected and stopped by the system before any OS, device, or application state is affected is termed *fail-stop*. More insidious failures may corrupt the system or application and never be detected. The system's response to failure determines whether a failure is fail-stop. For example, a system that detects and prevents accidental writes to kernel data structures exhibits fail-stop behavior for such a bug, whereas one that allows corruption does not.

Appropriate OS techniques can ensure that drivers execute in a fail-stop fashion [Seltzer et al. 1996; Swift et al. 2005; Wahbe and Lucco 1998]. For example, in earlier work we described Nooks [Swift et al. 2005], a kernel reliability subsystem that executes each driver within its own in-kernel protection domain. Nooks detects faults caused by memory protection violations, excessive CPU usage, and certain bad parameters passed to the kernel. When Nooks detects a failure, it stops execution within the driver's protection domain and triggers a recovery process. We reported that Nooks was able to detect approximately 75% of failures in synthetic fault-injection tests [Swift et al. 2005]. For this work, we assume that drivers fail by crashing, hanging, producing invalid outputs, and exceeding resource limits. We assume that drivers are not malicious, however.

Shadow drivers can recover only from failures that are both transient and fail-stop. Deterministic failures may recur when the driver recovers, again causing a failure. In contrast, transient failures are triggered by environmental factors that are unlikely to persist during recovery. In practice, many drivers experience transient failures, caused by the complexities of the kernel execution environment (e.g. asynchrony, interrupts, locking protocols, and virtual memory) [Arthur 2004], which are difficult to find and fix. Deterministic driver failures, in contrast, are more easily found and fixed in the testing phase of development because the failures are repeatable [Gray 1985]. Recoverable failures must also be fail-stop, because shadow drivers *conceal* failures from the system and applications. Hence, shadow drivers require a reliability subsystem to detect and stop failures before they are visible to applications or the operating system. Although shadow drivers may use any mechanism that provides these services, our implementation uses Nooks.

3.2 Shadow Drivers

A *shadow driver* is a kernel agent that improves reliability for a single device driver. It compensates for and recovers from a driver that has failed. When a driver fails, its shadow restores the driver to a functioning state in which it can process I/O requests made before the failure. While the driver recovers, the shadow driver services its requests.

Shadow drivers execute in one of two modes: passive or active. In *passive* mode, used during normal (non-faulting) operation, 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 a 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. They are not intended to provide any service to either party and exist only to track the state of the driver as necessary for recovery.

In *active* mode, which occurs during recovery from a failure, the shadow driver performs two functions. First, it "impersonates" the failed driver, intercepting and responding to calls from the kernel. Therefore, the kernel and higher-level applications continue operating in as normal a fashion as possible. Second, the shadow driver impersonates the kernel to restart the failed driver, intercepting and responding to calls from the restarted driver to the kernel. In other words, in active mode the shadow driver looks like the kernel to the driver and like the driver to the kernel. Only the shadow driver is aware of the deception. This approach hides recovery details from the driver, which is unaware that a shadow driver is restarting it after a failure.

Once the driver has restarted, the active-mode shadow reintegrates the driver into the system. It re-establishes any application configuration state downloaded into the driver and then resumes pending requests.

A shadow driver is a "class driver," aware of the interface to the drivers it shadows but *not* of their implementations. A single shadow driver implementation can recover from a failure of any driver in the class. The class orientation has three key implications. First, an operating system can leverage a few implementations of shadow drivers to recover from failures in a large number of device drivers. 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. Finally, if a new driver is loaded into the kernel, no new shadow driver is required *as long as* a shadow for that class already exists. 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. Shadow updating is required only to respond to a change in the kernel-driver programming interface.

3.3 Taps

As we have seen, a shadow driver monitors communication between a functioning driver and the kernel, and impersonates one component to the other during failure and recovery. These activities are made possible by a new mechanism, called a *tap*. Conceptually, a tap is a T-junction placed between the kernel and its drivers. It can be set to replicate calls during passive mode and redirect them during recovery.

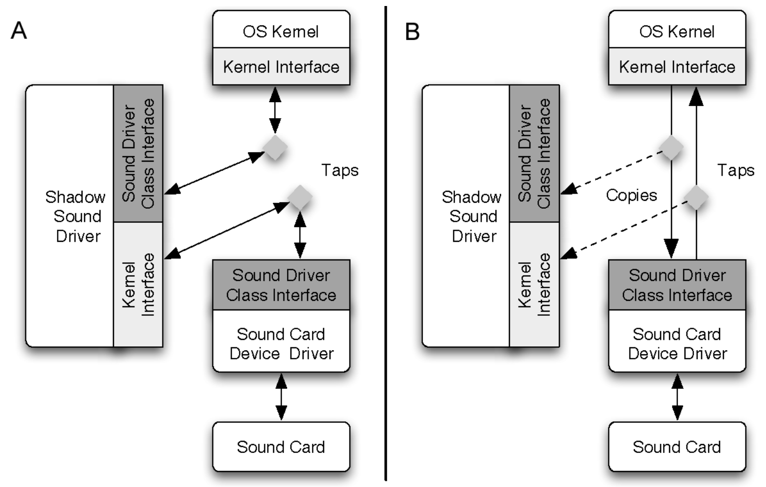

Fig. 2. (A) A sample shadow driver operating in passive mode. Taps inserted between the kernel and sound driver ensure that all communication between the two is passively monitored by the shadow driver. (B) A sample shadow driver operating in active mode. The taps redirect communication between the kernel and the failed driver directly to the shadow driver.

A tap operates in passive or active mode, corresponding to the state of the shadow driver attached to it. During passive-mode operation, the tap: (1) invokes the original driver, then (2) invokes the shadow driver with the parameters and results of the call. This operation is shown in Figure 2A.

On failure, the tap switches to active mode, shown in Figure 2B. In this mode, it: (1) terminates all communication between the driver and kernel, and (2) redirects all invocations to their corresponding interface in the shadow. In active mode, both the kernel and the recovering device driver interact only with the shadow driver. Following recovery, the tap returns to its passive-mode state.

Taps depend on the ability to dynamically dispatch all communication between the driver and the OS. Consequently, all communication into and out of a driver being shadowed must be explicit, such as through a procedure call or a message. Most drivers operate this way, but some do not, and so cannot be shadowed. For example, kernel video drivers often communicate with user-mode applications through shared memory regions [Kilgard et al. 1995].

3.4 The Shadow Manager

Recovery is supervised by the *shadow manager*, which is a kernel agent that interfaces with and controls all shadow drivers. The shadow manager instantiates new shadow drivers and injects taps into the call interfaces between the device driver and kernel. It also receives notification from the fault-isolation subsystem that a driver has stopped due to a failure.

When a driver fails, the shadow manager transitions its taps and shadow driver to active mode. In this mode, requests for the driver's services are redirected to an appropriately prepared shadow driver. The shadow manager then initiates the shadow driver's recovery sequence to restore the driver. When recovery ends, the shadow manager returns the shadow driver and taps to passive-mode operation so the driver can resume service.

3.5 Summary

Our 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 fault. A shadow driver, normally passive, monitors communication between the kernel and the driver. It becomes an active proxy when a driver fails, and then manages its recovery.

4. SHADOW DRIVER IMPLEMENTATION

This section describes the implementation of shadow drivers in the Linux operating system [Bovet and Cesati 2002]. We have implemented shadow drivers for three classes of device drivers: sound card drivers, network interface drivers, and IDE storage drivers.

4.1 General Infrastructure

All shadow drivers rely on a generic service infrastructure that provides three functions. An *isolation service* prevents driver errors from corrupting the kernel by stopping a driver on detecting a failure. A transparent *redirection mechanism* implements the taps required for transparent shadowing and recovery. Last, an *object tracking service* tracks kernel resources created or held by the driver so as to facilitate recovery.

Our shadow driver implementation uses Nooks to provide these functions. Through its fault isolation subsystem, Nooks [Swift et al. 2005] isolates drivers within separate kernel protection domains. The domains use memory protection to trap driver faults and ensure the integrity of kernel memory. Nooks interposes proxy procedures on all communication between the device driver and kernel. We insert our tap code into these Nooks proxies to replicate and redirect communication. Finally, Nooks tracks kernel objects used by drivers to perform garbage collection of kernel resources during recovery.

Our implementation adds a shadow manager to the Linux operating system. In addition to receiving failure notifications from Nooks, the shadow manager handles the initial installation of shadow drivers. In coordination with the kernel's module loader, which provides the driver's class, the shadow manager creates a new shadow driver instance for a driver. Because a single shadow driver services a 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. The new instance shares the same code with all other instances of that shadow driver class.

Figure 3 shows the driver recovery subsystem, which contains the Nooks fault isolation subsystem, the shadow manager, and a set of shadow drivers, each of which can monitor one or more device drivers.

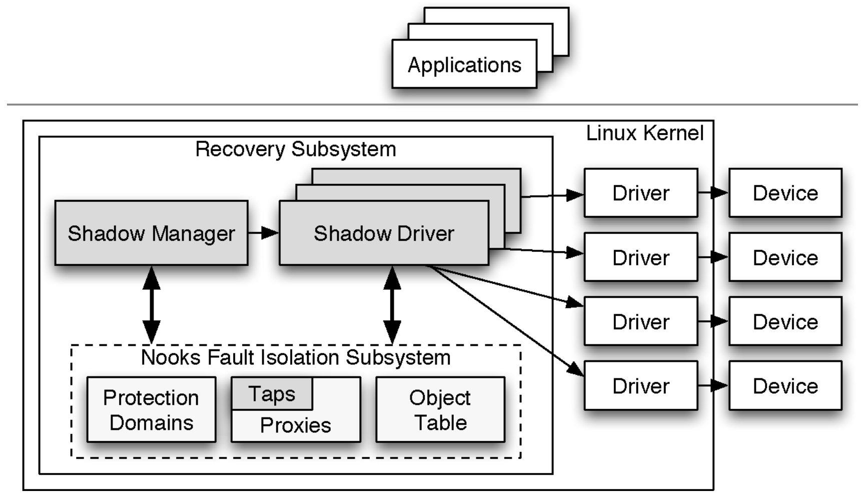

Fig. 3. The Linux operating system with several device drivers and the driver recovery subsystem. New code components include the taps, the shadow manager, and a set of shadow drivers, all built on top of the Nooks driver fault isolation subsystem.

4.2 Passive-Mode Monitoring

In passive mode, a shadow driver records several types of information. First, it tracks requests made to the driver, enabling pending requests to execute correctly after recovery. For connection-oriented drivers, the shadow driver records the state of each active connection, such as offset or positioning information. For request-oriented drivers, the shadow driver maintains a *log* of pending commands and arguments. An entry remains in the log until the corresponding request has been handled.

The shadow driver also records configuration and driver parameters that the kernel passes into the driver. During recovery, the shadow uses this information to act in the driver's place, returning the same information that was passed in previously. This information also assists in reconfiguring the driver to its prefailure state when it is restarted. For example, the shadow sound driver keeps a log of `ioctl` calls (command numbers and arguments) that configure the driver. This log makes it possible to: (1) act as the device driver by remembering the sound formats it supports, and (2) recover the driver by resetting properties, such as the volume and sound format in use.

The shadow driver maintains only the *configuration* of the driver in its log. For stateful devices, such as frame buffers or storage devices, it 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. It can restore transient state (state that is lost when the device resets) if it can force the device's clients to recreate that state, for example, by redrawing the contents of a frame buffer.

Last, the shadow tracks all kernel objects that the driver allocated or received from the kernel. These objects would otherwise be lost when the driver fails, causing a memory leak. For example, the shadow must record all timer callbacks registered and all hardware resources owned, such as interrupt lines and I/O memory regions.

In many cases, passive-mode calls do no work and the shadow returns immediately to the caller. For example, the dominant calls to a sound-card driver are `read` and `write`, which record or play sound. In passive mode, the shadow driver implements these calls as *no-ops*, since there is no need to copy the real-time sound data flowing through the device driver. For an `ioctl` call, however, the sound-card shadow driver logs the command and data for the connection. Similarly, the shadow driver for an IDE disk does little or no work in passive mode, since the kernel and disk driver handle all I/O and request queuing. Finally, for the network shadow driver, much of the work is already performed by the Nooks object-tracking system, which keeps references to outstanding packets.

Opaque parameters, passed as `void *` pointers, pose problems for shadow drivers because their type is not declared in the driver interface. As a result, a shadow may be unable to capture and interpret the parameter. However, the class-based approach allows shadow drivers to interpret most opaque pointers. Because the interface to the driver is standardized, callers of the interface have no special knowledge of how the driver interprets these pointers. As a result, the interface specification must define the type of opaque pointers. For example, the Open Sound System interface [Tranter 2000] defines opaque pointer parameters to the `ioctl` call for sound drivers. The shadow sound driver relies on this standard to interpret and log `ioctl` requests.

4.3 Active-Mode Recovery

A driver typically fails by generating an illegal memory reference or passing an invalid parameter across a kernel interface. The kernel-level failure detector notices the failure and invokes the shadow manager, which locates the appropriate shadow driver and directs it to recover the failed driver. 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.

4.3.1 *Stopping the Failed Driver.* The shadow manager begins recovery by informing the responsible shadow driver that a failure has occurred. It also switches 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. In addition, shadow drivers call the underlying isolation subsystem to preempt any threads that had called into the driver. These threads then invoke the shadow driver proxying code for the function they had been executing.

Informed of the failure, the shadow driver first disables execution of 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 from enter into kernel memory.

To prepare for restarting the device driver, the shadow garbage collects resources held by the driver. It retains objects that the kernel uses to request driver services, to ensure that the kernel does not see the driver "disappear" as it is restarted. The shadow releases the remaining resources.

When releasing objects, the shadow must avoid confusing higher levels of software that may still be using the objects. Drivers themselves face this problem when shutting down. Hence, the object tracker uses the same routines that extensions use to release objects. For example, the shadow calls the `kfree_skb` function to release network packets in flight, which decrements a reference count and potentially frees the packet. This is the same function used by network device drivers to release packets to the kernel. As a result, higher levels of software, such as network protocol code, continue to function correctly during and after recovery.

The recovery code must also take care to prevent livelock when multiple drivers share a single interrupt request line. Disabling an interrupt blocks all drivers that share the line. Under normal conditions, all the drivers using the line 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 spurious interrupt. Shadow drivers rely on this behavior by registering a periodic timer callback that invokes all drivers sharing an interrupt. If a device *did* generate an interrupt during the timer period, its driver will handle it. Otherwise, its driver 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.

Shadow drivers depend on the underlying isolation subsystem to record the status of locks that are shared between the driver and the kernel, such as locks integral to kernel data structures. When a driver fails, the shadow releases all shared locks acquired by the driver to prevent the system from hanging. Locks that are private to a driver need not be recorded, because any threads blocking on a private lock are preempted when recovery begins.

4.3.2 *Reinitializing the Driver.*

The shadow driver next "reboots" the driver from a clean state. Normally, restarting a driver requires reloading the driver from disk. However, we cannot assume that the disk is functional during recovery. For this reason, when creating a new shadow driver instance, the shadow manager caches in the shadow instance a copy of the device driver's initial, clean data section. These sections tend to be small. The driver's code is kernel-read-only, so it can be reused from memory.

The shadow restarts the driver by initializing the driver's state and then repeating the kernel's driver initialization sequence. 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 its prefailure kernel resources. During driver reboot, the driver makes a number of calls into the kernel to discover information about itself and to link itself into the kernel. For example, the driver calls the kernel to register itself as a driver and to request hardware and kernel resources. The taps redirect these calls to the shadow driver, which reconnects the driver to existing kernel data structures. Thus, when the driver attempts to register with the kernel, the shadow intercepts the call and reuses the existing driver registration, avoiding the allocation of a new one. For requests that generate callbacks, such as a request to register the driver with the PCI subsystem, the shadow emulates the kernel, making the same callbacks to the driver with the same parameters. The driver also acquires hardware resources. If these resources were previously disabled at the first step of recovery, the shadow re-enables them, for example, 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.

4.3.3 *Transferring State to the New Driver.* The final recovery step restores the driver state that existed at the time of the fault, 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 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 reopens 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 and then resubmits to the driver any requests that were outstanding when the driver crashed.

As an example, for a failed sound card driver, the shadow driver resets the sound driver and all its open connections back to their prefailures 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 and replays any commands that set driver properties.

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 driver that is recording sound stores the number of bytes it has recorded since the last reset. After recovery, the sound driver initializes this counter to zero. Because no interface call is provided to change 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. Unfortunately, shadow drivers cannot guarantee exactly-once behavior for driver requests and must rely on devices and higher levels of software to absorb duplicate requests. For example, 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 request was actually processed. During recovery, the shadow driver has two choices: 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, which may limit its ability to mask failures.

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

4.4 Active-Mode Proxying of Kernel Requests

While a shadow driver is restoring a failed driver, it is also acting in place of the driver to conceal the failure and recovery from applications and the kernel. The shadow driver's response to a driver request depends on the driver class and request semantics. In general, the shadow will take one of five actions: (1) respond with information that it has recorded, (2) silently drop the request, (3) queue the request for later processing, (4) block the request until the driver recovers, or (5) report that the driver is busy and the kernel or application should try again later. The choice of strategy depends on the caller's expectations of the driver.

Writing a shadow driver that proxies for a failed driver requires knowledge of the kernel-driver interface 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 shadow for a driver class uses these requirements to select the response strategy.

Active proxying is simplified for driver interfaces that support a notion of "busy." By reporting that the device is currently busy, shadow drivers instruct the kernel or application to block calls to a driver. For example, network drivers in Linux may reject requests and turn themselves off if their queues are full. The kernel then refrains from sending packets until the driver turns itself back on. Our 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 fill up. Sound drivers can report that their buffers are temporarily full.

Our shadow sound card driver uses a mix of all five strategies for emulating functions in its service interface. The shadow blocks kernel `read` and `write` requests, which play or 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.

4.5 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 recover by restarting the failed driver. As a result, they can only recover for drivers that can be killed and restarted safely. This is true for device drivers that are dynamically unloaded when hardware devices are removed from a system, but is not true for all drivers

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.

Shadow drivers also assume that driver failure does 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.

To be effective, 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, then the shadow may be unable to restore the driver to its prefailure 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.

The abilities of the isolation and failure detection subsystem also limit the effectiveness of shadow drivers. If this subsystem cannot prevent kernel corruption, then shadow drivers cannot facilitate system recovery. In addition, if the fault isolation subsystem does not detect a failure, 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 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.

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 restarts the driver but does not perform active proxying.

4.6 Summary

This section presented the details of our Linux shadow driver implementation. The shadow driver concept is straightforward: passively monitor normal operations, proxy during failure, and reintegrate during recovery. Ultimately, the value of shadow drivers depends on the degree to which they can be implemented correctly, efficiently, and easily, in an operating system. The following section evaluates some of these questions both qualitatively and quantitatively.

5. EVALUATION

This section evaluates four key aspects of shadow drivers.

(1) *Performance.* What is the performance overhead of shadow drivers during normal, passive-mode operation (in the absence of failure)? This is the dynamic cost of our mechanism.
(2) *Fault-Tolerance.* Can applications that use a device driver continue to run even after the driver fails? We evaluate shadow driver recovery in the presence of simple failures to show the benefits of shadow drivers compared to a system that provides failure isolation alone.
(3) *Limitations.* How reasonable is our assumption that driver failures are fail-stop? Using synthetic fault injection, we evaluate how likely it is that driver failures are fail-stop.
(4) *Code size.* How much code is required for shadow drivers and their supporting infrastructure? We evaluate the size and complexity of the shadow driver implementation to highlight the engineering cost of integrating shadow drivers into an existing system.

Based on a set of controlled application and driver experiments, our results show that shadow drivers: (1) impose relatively little performance overhead, (2) keep applications running when a driver fails, (3) are limited by a system's ability to detect that a driver has failed, and (4) can be implemented with a modest amount of code.

The experiments were run on a 3 GHz Pentium 4 PC with 1 GB of RAM and an 80 GB, 7200 RPM IDE disk drive. We built and tested three Linux shadow drivers for three device-driver classes: network interface controller, sound card, and IDE storage device. To ensure that our generic shadow drivers worked consistently across device driver implementations, we tested them on thirteen different Linux drivers, shown in Table I. Although we present detailed results for only one driver in each class (*e1000*, *audigy*, and *ide-disk*), behavior across all drivers was similar.

5.1 Performance

To evaluate performance, we produced three OS configurations based on the Linux 2.4.18 kernel:

(1) *Linux-Native* is the unmodified Linux kernel.
(2) *Linux-Nooks* is a version of *Linux-Native*, which includes the Nooks fault isolation subsystem but no shadow drivers. When a driver fails, this system restarts the driver but does not attempt to conceal its failure.
(3) *Linux-SD* is a version of *Linux-Nooks*, which includes our entire recovery subsystem, including the Nooks fault isolation subsystem, the shadow manager, and our three shadow drivers.

We selected a variety of common applications that depend on our three device driver classes and measured their performance. The application names and behaviors are shown in Table II.

Table I. The Three Classes of Shadow Drivers and the Linux Drivers Tested

Class	Driver	Device
Network	e1000	Intel Pro/1000 Gigabit Ethernet
	pcnet32	AMD PCnet32 10/100 Ethernet
	3c59x	3COM 3c509b 10/100 Ethernet
	e100	Intel Pro/100 Ethernet
	epic100	SMC EtherPower 10/100 Ethernet
Sound	audigy	SoundBlaster Audigy sound card
	emu10k1	SoundBlaster Live! sound card
	sb	SoundBlaster 16 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

Table II. 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 *(e1000 driver)*	network send (*netperf*) over TCP/IP network receive (*netperf*) over TCP/IP 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*

Different applications have different performance metrics of interest. For the disk and sound drivers, we ran the applications shown in Table II and measured elapsed time. For the network driver, throughput is a more useful metric; therefore, we ran the throughput-oriented *network send* and *network receive* benchmarks. For all drivers, we also measured CPU utilization while the programs ran. All measurements were repeated several times and showed a variation of less than one percent.

Most of the test applications are I/O bound and the CPU is not fully utilized. For these tests, we expected that isolation and recovery would have a negligible performance impact because there is spare capacity to absorb the overhead. The CPU utilization, though, should increase. For the CPU-bound workloads, *compiler* and *encoder*, the performance cost of fault tolerance is reflected directly in the end-to-end execution time.

Figure 4 shows the performance of *Linux-Nooks* and *Linux-SD* relative to *Linux-Native*. Figure 5 compares CPU utilization for execution of the same applications on the three OS versions. Both figures make clear that compared

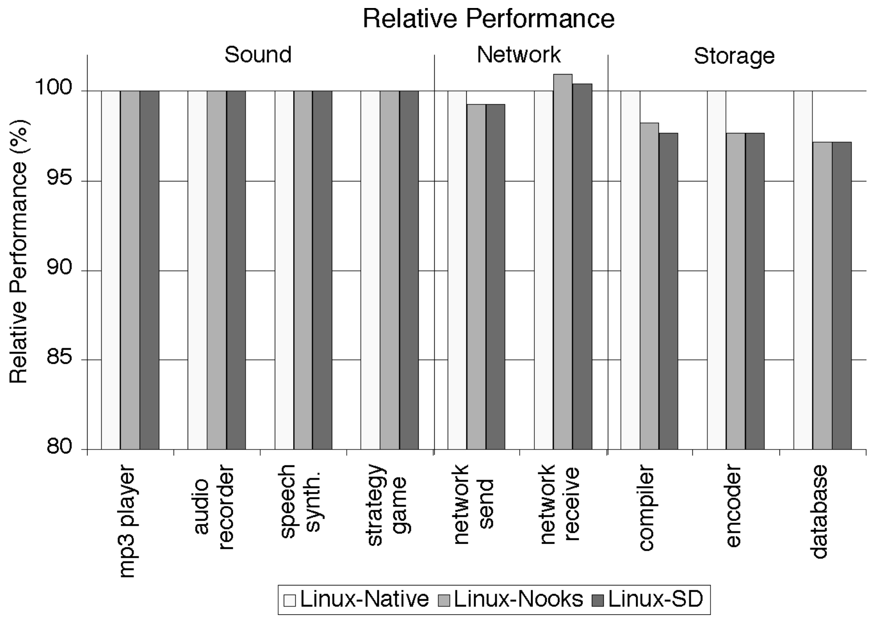

Fig. 4. Comparative application performance, relative to *Linux-Native*, for three configurations. The X-axis crosses at 80%.

to running with no isolation at all, shadow drivers impose only a small performance penalty for low bandwidth drivers, such as the sound card driver and disk driver, and a moderate penalty for the high-bandwidth network driver. Furthermore, shadow drivers impose *no* additional penalty beyond that imposed by isolation alone. Across all nine applications, performance of the system with shadow drivers averaged 99% of the system without them, and was never worse than 97%.

The low overhead of shadow drivers can be explained in terms of its two constituents: fault isolation and the shadowing itself. As mentioned previously, fault isolation runs each driver in its own domain, leading to overhead caused by domain crossings. Each domain crossing takes approximately 3000 cycles, mostly to change page tables and execution stacks. As a side effect of changing page tables, the Pentium 4 processor flushes the TLB, resulting in TLB misses that can noticeably slow down drivers [Swift et al. 2005].

For example, the kernel calls the driver approximately 1000 times per second when running *audio recorder*. Each invocation executes only a small amount of code. As a result, isolating the sound driver adds only negligibly to CPU utilization, because there are not many crossings and not much code to slow down. For the most disk-intensive of the IDE storage applications, the *database* benchmark, the kernel and driver interact only 290 times per second. However, each call into the *ide-disk* driver results in substantial work to process a queue of disk requests. The TLB-induced slowdown doubles the time *database* spent in the driver relative to *Linux-Native* and increases the application's CPU utilization from 21% to 27%. On the other hand, the *network send* benchmark transmits

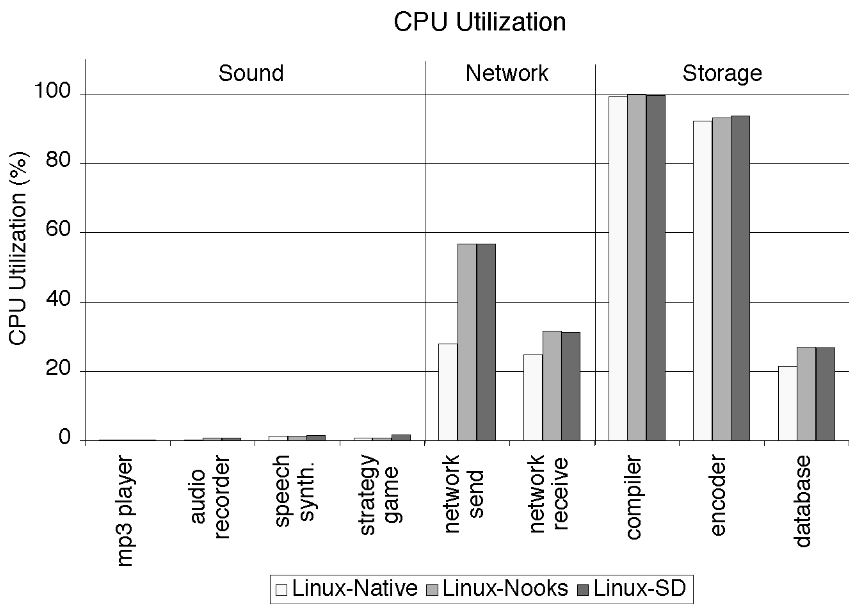

Fig. 5. Absolute CPU utilization by application for three configurations.

45,000 packets per second, causing 45,000 domain crossings. The driver does little work for each packet, but the overall impact is visible in Figure 5, where CPU utilization for this benchmark increases from 28% to 57% with driver fault isolation.

In the case the actual shadowing, we see from a comparison of the *Linux-Nooks* and *Linux-SD* bars in Figures 4 and 5, that the cost is small or negligible. As noted in Section 4.2, many passive-mode shadow-driver functions are no-ops. As a result, the incremental passive-mode performance cost over basic fault isolation is low or unmeasurable in many cases.

In summary, then, the overall performance penalty of shadow drivers during failure-free operation is low, suggesting that shadow drivers could be used across a wide range of applications and environments.

5.2 Fault-Tolerance

Regardless of performance, 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, we tested 10 applications on the three configurations, *Linux-Native*, *Linux-Nooks*, and *Linux-SD*. For the disk and sound drivers, we again ran the applications shown in Table II. Because we were interested in the applications' response to driver failures and not performance, we substituted *network file copy*, *remote window manager*, and *network analyzer* for the networking benchmarks.

We simulated common bugs by injecting a software fault into a device driver while an application using that driver was running. Because

Table III. The Observed Behavior of Several Applications Following the Failure of the Device Drivers Upon Which They Rely

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 mgr.	CRASH	√	√
	network analyzer	CRASH	MALFUNCTION	√
IDE (ide-disk driver)	compiler	CRASH	CRASH	√
	encoder	CRASH	CRASH	√
	database	CRASH	CRASH	√

both Linux-Nooks and Linux-SD depend on the same isolation and failure-detection services, we differentiate their recovery capabilities by simulating failures that are easily isolated and detected. To generate realistic synthetic driver bugs, we analyzed patches posted to the Linux Kernel Mailing List [http://www.uwsg.indiana.edu/hypermail/linux/kernel]. We 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, we 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. We also found missing pointer initialization code (two instances) and bad calculations (two instances) that led to endless loops and buffer overruns. Because Nooks detects these faults, they cause fail-stop failures on *Linux-Nooks* and *Linux-SD*.

We injected a null-pointer dereference bug derived from these patches, into our three drivers. We ensured that the synthetic bug was transient by inserting the bug into uncommon execution paths, such as code that handles unusual hardware conditions. These paths are rarely executed, so we accelerated the occurrence of faults by also executing the bug at random intervals. The fault code remains active in the driver during and after recovery.

Table III shows the three application behaviors we 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 Nooks' kernel-focused recovery system does not extend to applications. For example, Nooks isolates the kernel from driver faults and reboots (unloads, reloads, and restarts) the driver. However, it 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.

Linux-SD recovers transparently from disk driver failures. Recovery is possible because the IDE storage shadow driver instance maintains the failing driver's initial state. During recovery the shadow copies back the 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, Nooks, 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. Our 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 kernel device driver. Cost is a barrier to the first approach. Shadow drivers are a path to the second.

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, we often observed a slight timing disruption while the driver recovered. At best, output was queued in the shadow driver. At worst, input was lost by the device. The length of the delay was primarily determined by the recovering device driver itself, which, 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.[2] 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.

5.3 Limits to Recovery

The previous 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 state in 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.

Non-Fail-Stop Failures. If driver failures are not fail stop, then shadow drivers may not be useful. To evaluate whether device driver failures are indeed fail-stop, we performed large-scale fault-injection tests of our drivers and applications running on Linux-SD.

Ideally, we would test shadow drivers with a variety of real drivers long enough for many failures to occur. This is not practical, however, because of the sheer time needed for testing. Instead, our experiments use *synthetic fault injection* to insert faults into Linux kernel extensions. We adapted a fault injector developed for the Rio File Cache [Ng and Chen 1999] and ported it to Linux. The injector automatically changes single instructions in the extension code to emulate a variety of common programming errors, such as uninitialized local variables, bad parameters, and inverted test conditions.

We inject two different classes of faults into the system. First, we inject faults that emulate specific programming errors that are common in kernel code [Sullivan and Chillarege 1991; Christmansson and Chillarege 1996]. *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. We 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, we expand the range of testing by injecting random changes that do not model specific programming errors. In this category are *text faults*, in which we flip a random bit in a random instruction, and *NOP faults*, in which

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

Table IV. 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

we delete a random instruction. Table IV shows the types of faults we inject, and how the injector simulates programming errors (see Ng and Chen [1999] for a more complete description of the fault injector).

Unfortunately, there is no available information on the distribution of fault sources for real-world driver failures. Instead, we inject the same number of faults of each type for each test. 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.

For each driver and application combination, we ran 400 fault-injection trials. In total, we ran 2400 trials across the three drivers and six applications. Between trials, we reset the system and reloaded the driver. For each trial, we injected five random errors into the driver while the application was using it. We ensured the errors were transient by removing them during recovery. After injection, we visually observed the impact on the application and the system to determine whether a failure or recovery had occurred. For each driver, we tested two applications with significantly different usage scenarios. For example, we chose one sound-playing application (*mp3 player*) and one sound-recording application (*audio recorder*).

If we observed a failure, we then assessed the trial on two criteria: whether the fault was detected, and whether the shadow driver could mask the failure and subsequent recovery from the application. For undetected failures, we 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 shows the results of our experiments. For each application, we 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. Only 18% of the injected faults caused a visible failure.

In our 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 facilitate driver recovery. The system failed to detect 35% of the failures. In these cases, we 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

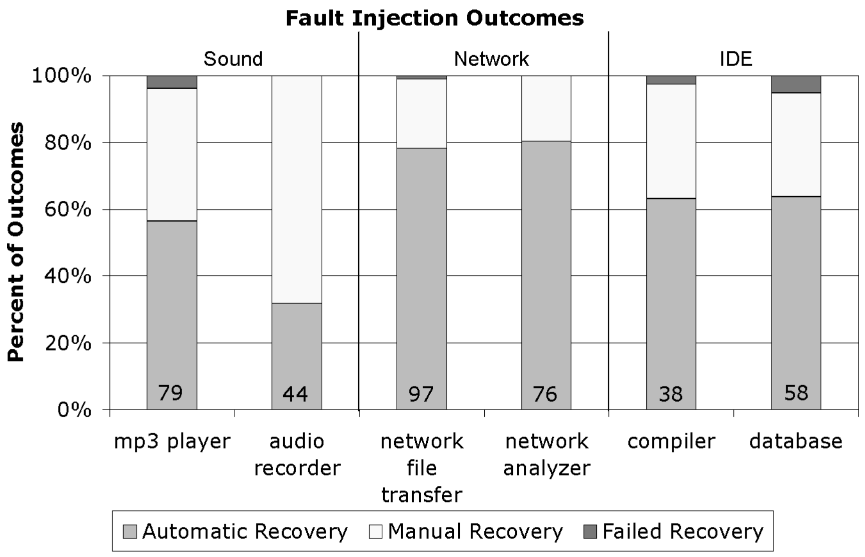

Fig. 6. Results of fault-injection experiments on Linux-SD. We show (1) the percentage of failures that are automatically detected by the fault isolation subsystem, (2) the percentage of failures that shadow drivers successfully recovered, and (3) the percentage of failures that could not be recovered. The total number of failures each application experienced is shown at the bottom of the chart.

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 these failures been detected earlier with a better failure detector.

Across all applications and drivers, we found three major causes of undetected failure. First, the system did not detect application hangs caused by I/O requests that never completed. Second, the system did not detect errors in the interactions between the device and the driver, for example, incorrectly copying sound data to a sound card. Third, the system did not detect certain bad parameters, such as incorrect result codes or data values. 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.

Our results demonstrate a need for class-based failure detectors that can detect violations of the driver interface to achieve high levels of reliability. However, driver failures need not be detected quickly to be fail-stop. There was a significant delay between the failure and the subsequent manual recovery in our tests, and yet the applications survived the vast majority of undetected failures. Thus, even a slow failure detector can be effective at improving application reliability.

Table V. Size and Quantity of Shadows and the Drivers they Shadow

Driver Class	Shadow Driver Lines of Code	Device Driver 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

Nontransient Failures. Shadow drivers can recover from transient failures only. In contrast, deterministic failures may recur during recovery when the shadow configures the driver. While unable to recover, shadow drivers are still useful for these failures. When a failure recurs during recovery, the sequence of shadow driver recovery events creates a detailed reproduction scenario that aids diagnosis. This record of recovery 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.

5.4 Code Size

The preceding sections evaluated the *efficiency* and *effectiveness* of shadow drivers. This section examines the *complexity* of shadow drivers in terms of code size, which can serve as a proxy for complexity.

Table V shows, for each class, the size in lines of code of the shadow driver for the class. For comparison, we show the size of the driver from the class that we tested and the total number and cumulative size of existing Linux device drivers in that class in the 2.4.18 kernel. The total code size is an indication of the leverage gained through the shadow's class-driver structure. Furthermore, the table shows that a shadow driver is significantly smaller than the device driver it shadows. For example, our 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.

The Nooks driver fault isolation subsystem we built upon contains about 23,000 lines of code. In total, we added about 3300 lines of new code to Nooks to support our three class drivers. Otherwise, we made no changes to the remainder of the Linux kernel. Shadow drivers required the addition of approximately 600 lines of code for the shadow manager, 800 lines of common code shared by all shadow drivers, and another 750 lines of code for general utilities. Of the 177 taps we inserted, only 31 required actual code; the remainder were no-ops.

5.5 Summary

This section examined the performance, fault-tolerance, limits, and code size of shadow drivers. Our results demonstrate that: (1) the performance overhead of shadow drivers during normal operation is small, particularly when compared to a purely isolating system, (2) applications that failed in any form on *Linux-Native* or *Linux-Nooks* ran normally with shadow drivers, (3) the reliability provided by shadow drivers is limited by the system's ability to detect failures, and (4) shadow drivers are small, even relative to single device driver. Overall, these results indicate that shadow drivers have the potential to significantly improve the reliability of applications on modern operating systems, with only modest cost.

6. CONCLUSIONS

Improving the reliability of modern systems demands that we increase their resilience. To this end, we designed and implemented *shadow drivers*, which mask device driver failures from both the operating system and applications.

Our experience shows that shadow drivers improve application reliability by concealing a driver's failure while facilitating recovery. A single shadow driver can enable recovery for an entire class of device drivers. Shadow drivers are also efficient, imposing little performance degradation. Finally, they are transparent, requiring no code changes to existing drivers.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Peter Chen, who provided many valuable insights.

REFERENCES

Arthur, S. 2004. Fault resilient drivers for Longhorn server. Tech. Rep. WinHec 2004 Presentation DW04012, Microsoft Corporation, Redmond, WA. May.

Babaoğlu, Ö. 1990. Fault-tolerant computing based on Mach. In *Proceedings of the USENIX Mach Symposium*. Burlington, VT, 185–199.

Barga, R., Lomet, D., and Weikum, G. 2002. Recovery guarantees for general multi-tier applications. In *International Conference on Data Engineering*. IEEE, San Jose, California, 543–554.

Bartlett, J. F. 1981. A NonStop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, 22–29.

Borg, A., Balu, W., Graetsch, W., Herrmann, F., and Oberle, W. 1989. Fault tolerance under UNIX. *ACM Trans. Comput. Syst. 7*, 1 (Feb.), 1–24.

Bovet, D. P. and Cesati, M. 2002. *Inside the Linux Kernel*. O'Reilly & Associates, Sebastopol, California.

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

Bressoud, T. C. and Schneider, F. B. 1996. Hypervisor-based fault tolerance. *ACM Tran. Compu. Syst. 14*, 1 (Feb.), 80–107.

Candea, G. and Fox, A. 2001. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems*. 125–132.

Chandra, S. and Chen, P. M. 1998. How fail-stop are faulty programs? In *Proceedings of the 28th Symposium on Fault-Tolerant Computing*. IEEE, Munich, Germany, 240–249.

Chandra, S. and Chen, P. M. 2000. Whither generic recovery from application faults? A fault study using open-source software. In *Proceedings of the 2000 IEEE International Conference on Dependable Systems and Networks*. New York, New York, 97–106.

Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. 1996. The Rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, Massachusetts, 74–83.

Chiueh, T., Venkitachalam, G., and Pradhan, P. 1999. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. Kiawah Island Resort, South Carolina, 140–153.

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. 2001. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. Lake Louise, Alberta, 73–88.

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

Engler, D. R., Kaashoek, M. F., and Jr., J. O. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, Colorado, 251–266.

Feng, W. 2003. Making a case for efficient supercomputing. *ACM Queue 1*, 7 (Oct.).

Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. 1997. The Flux OSKit: a substrate for OS language and research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. 38–51.

Gray, J. 1985. Why do computers stop and what can be done about it? Tech. Rep. 85-7, Tandem Computers, Cupertino, CA. June.

Gray, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Hand, S. M. 1999. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. New Orleans, LA, 73–86.

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

Kilgard, M. J., Blythe, D., and Hohn, D. 1995. System support for OpenGL direct rendering. In *Proceedings of Graphics Interface*. Canadian Human-Computer Communications Society, Toronto, Ontario, 116–127.

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

Linux Kernel Mailing List. `http://www.uwsg.indiana.edu/hypermail/linux/kernel`.

Lowell, D. E., Chandra, S., and Chen, P. M. 2000. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*. San Diego, California, 289–304.

Lowell, D. E. and Chen, P. M. 1998. Discount checking: Transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan (Nov.).

Muller, G., Banâtre, M., Peyrouze, N., and Rochat, B. 1996. Lessons from FTM: An experiment in design and implementation of a low-cost fault-tolerant system. *IEEE Trans. Softw. Eng. 45*, 2 (June), 332–339.

Ng, W. T. and Chen, P. M. 1999. The systematic improvement of fault tolerance in the Rio file cache. In *Proceedings of the 29th Symposium on Fault-Tolerant Computing*. IEEE, 76–83.

Orgovan, V. and Tricker, M. 2003. An introduction to driver quality. *Microsoft WinHec Presentation DDT301* (Aug). Redmond, WA.

Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kýcýman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. 2002. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Tech. Rep. CSD-02-1175, UC Berkeley Computer Science. Mar.

Plank, J. S., Beck, M., Kingsley, G., and Li, K. 1995. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Conference*. New Orleans, Louisiana, 213–224.

Russinovich, M., Segall, Z., and Siewiorek, D. 1993. Application transparent fault management in Fault Tolerant Mach. In *Proceedings of the 23rd Symposium on Fault-Tolerant Computing*. IEEE, Toulouse, France, 10–19.

Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*. Seattle, Washington, 213–227.

Sullivan, M. and Chillarege, R. 1991. 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*. IEEE, Quebec, Canada, 2–9.

Swift, M. M., Bershad, B. N., and Levy, H. M. 2005. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. 23*, 1 (Feb.), 1–34.

Tranter, J. 2000. Open sound system programmer's guide. Available at `http://www.opensound.com/pguide/oss.pdf`.

Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. Asheville, North Carolina, 203–216.

Wahbe, R. S. and Lucco, S. E. 1998. Methods for safe and efficient implementation of virtual machines. US Patent 5,761,477.

Whittaker, J. A. 2001. Software's invisible users. *IEEE Softw. 18*, 3 (May), 84–88.

Wulf, W. A. 1975. Reliable hardware-software architecture. In *Proceedings of the International Conference on Reliable Software*. Los Angeles, California, 122–130.

Young, M., Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., and Tevanian, A. 1986. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference*. Atlanta, Georgia, 93–113.

Received March 2005; revised October 2005; accepted December 2005