Improving the Reliability of Commodity Operating Systems

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

Despite decades of research in extensible operating system technology, extensions such as device drivers remain a significant cause of system failures. In Windows XP, for example, drivers account for 85% of recently reported failures.

This paper describes Nooks, a *reliability subsystem* that seeks to greatly enhance OS reliability by isolating the OS from driver failures. The Nooks approach is practical: rather than guaranteeing complete fault tolerance through a new (and incompatible) OS or driver architecture, our goal is to prevent the *vast majority* of driver-caused crashes with *little or no change* to existing driver and system code. Nooks isolates drivers within lightweight protection domains inside the kernel address space, where hardware and software prevent them from corrupting the kernel. Nooks also tracks a driver's use of kernel resources to facilitate automatic clean-up during recovery.

To prove the viability of our approach, we implemented Nooks in the Linux operating system and used it to fault-isolate several device drivers. Our 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, we show that Nooks recovers automatically from 99% of the faults that otherwise cause Linux to crash.

While Nooks was designed for drivers, our techniques generalize to other kernel extensions. We demonstrate this by isolating a kernel-mode file system and an in-kernel Internet service. Overall, because Nooks supports existing C-language extensions, runs on a commodity operating system and hardware, and enables automated recovery, it represents a substantial step beyond the specialized architectures and type-safe languages required by previous efforts directed at safe extensibility.

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

General Terms: Reliability, Management

Additional Key Words and Phrases: Recovery, Device Drivers, Virtual Memory, Protection, I/O

1. INTRODUCTION

This paper describes the architecture, implementation, and performance of Nooks, a new operating system subsystem that allows existing OS extensions (such as device drivers and loadable file systems) to execute safely in commodity kernels.

Authors address: Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA.

In contemporary systems, any fault in a kernel extension can corrupt vital kernel data, causing the system to crash. To reduce the threat of extension failures, Nooks executes each extension in a *lightweight kernel protection domain* – a privileged kernel-mode environment with restricted write access to kernel memory. Nooks' interposition services track and validate all modifications to kernel data structures performed by the kernel-mode extension, thereby trapping bugs as they occur and facilitating subsequent automatic recovery.

Three factors motivated our research. First, computer system reliability remains a crucial but unsolved problem [Gillen et al. 2002; Patterson et al. 2002]. While the cost of high-performance computing continues to drop, 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. In addition, the growing sector of "unmanaged" systems, such as digital appliances and consumer devices based on commodity hardware and software [Hewlett Packard 2001; TiVo Corporation 2001], amplifies the need for reliability.

Second, OS extensions have become increasingly prevalent in commodity systems such as Linux (where they are called *modules* [Bovet and Cesati 2001]) and Windows (where they are called *drivers* [Custer 1993]). Extensions are optional components that reside in the kernel address space and typically communicate with the kernel through published interfaces. In addition to device drivers, extensions include file systems, virus detectors, and network protocols. Extensions now account for over 70% of Linux kernel code [Chou et al. 2001], while over 35,000 different drivers with over 120,000 versions exist on Windows XP desktops [Short 2003]. Many, if not most, of these extensions are written by programmers significantly less experienced in kernel organization and programming than those who built the operating system itself.

Third, extensions are a leading cause of operating system failure. In Windows XP, for example, drivers cause 85% of recently reported failures [Short 2003]. In Linux, the frequency of coding errors is seven times higher for device drivers than for the rest of the kernel [Chou et al. 2001]. While the core operating system kernel reaches high levels of reliability due to longevity and repeated testing, the *extended* operating system cannot be tested completely. With tens of thousands of extensions, operating system vendors cannot even identify them all, let alone test all possible combinations used in the marketplace.

Improving OS reliability will therefore require systems to become highly tolerant of failures in drivers and other extensions. Furthermore, the hundreds of millions of existing systems executing tens of thousands of extensions demand a reliability solution that is at once *backward compatible* and *efficient* for common extensions. Backward compatibility improves the reliability of already deployed systems. Efficiency avoids the classic tradeoff between robustness and performance.

Our focus on extensibility and reliability is not new. The last twenty years have produced a substantial amount of research on improving extensibility and reliability through the use of new kernel architectures [Engler et al. 1995], new driver architectures [Project-UDI 1999], user-level extensions [Forin et al. 1991; Liedtke 1995; Young et al. 1986], new hardware [Fabry 1974; Witchel et al. 2002], or type-safe languages [Bershad et al. 1995].

While many of the underlying techniques used in Nooks have been used in pre-

vious systems, Nooks differs from earlier efforts in two key ways. First, we target existing extensions for commodity operating systems rather than propose a new extension architecture. We want today's extensions to execute on today's platforms without change if possible. Second, we use C, a conventional programming language. We do not ask developers to change languages, development environments, or, most importantly, perspective. Overall, we focus on a single and very serious problem – reducing the huge number of crashes due to drivers and other extensions.

We implemented a prototype of Nooks in the Linux operating system and experimented with a variety of kernel extension types, including several device drivers, a file system, and a kernel Web server. Using automatic fault injection [Hsueh et al. 1997], we show that when injecting synthetic bugs into extensions, Nooks can gracefully recover and restart the extension in 99% of the cases that cause Linux to crash. In addition, Nooks recovered from all of the common causes of kernel crashes that we manually inserted. Extension recovery occurs quickly, as compared to a full system reboot, leaving most applications running. For drivers – the most common extension type – the impact on performance is low to moderate. Finally, of the eight kernel extensions we isolated with Nooks, seven required no code changes, while only 13 lines changed in the eighth. Although our prototype is Linux based, we expect that the architecture and many implementation features would port readily to other commodity operating systems.

The rest of this paper describes the design, implementation and performance of Nooks. The next section describes the system's guiding principles and high-level architecture. Section 3 discusses the system's implementation on Linux. We present experiments that evaluate the reliability of Nooks in Section 4 and its performance in Section 5. We then summarize related work in OS extensibility and reliability. Section 7 summarizes our work and draws conclusions.

2. ARCHITECTURE

The Nooks architecture is based on two core principles:

(1) *Design for fault resistance, not fault tolerance.* The system must prevent and recover from most, but not necessarily all, extension failures.
(2) *Design for mistakes, not abuse.* Extensions are generally well-behaved but may fail due to errors in design or implementation.

From the first principle, we are not seeking a complete solution for all possible extension errors. However, since extensions cause the vast majority of system failures, eliminating most extension errors will substantially improve system reliability. From the second principle, we have chosen to occupy the design space somewhere between "unprotected" and "safe." That is, the extension architecture for conventional operating systems (such as Linux or Windows) is unprotected: nearly any bug within the extension can corrupt or crash the rest of the system. In contrast, safe systems (such as SPIN [Bershad et al. 1995] or the Java Virtual Machine [Gosling et al. 1996]) strictly limit extension behavior and thus make no distinction between buggy and malicious code. We trust kernel extensions not to be malicious, but we 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 us to define an architecture that directly

supports existing driver code with only moderate performance costs. On the negative side, our solution does not detect or recover from 100% of all possible failures and can be easily circumvented by malicious code acting within the kernel. As examples, consider a malfunctioning driver that continues to run and 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. We do not attempt to detect or correct such failures.

Among failures that can crash the system, a spectrum of possible defensive approaches exist. 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). Our approach lies in the middle. Like all possible approaches, it reflects tradeoffs among performance, compatibility, complexity, and completeness. Section 3.6 describes our current limitations. Some limitations are architectural, while others are induced by the current hardware or software implementation. Despite these limitations, given tens of thousands of existing drivers, and the millions of failures they cause, a fault-resistant solution like the one we propose has practical implications and value.

2.1 Goals

Given the preceding principles, the Nooks architecture seeks to achieve three major goals:

(1) *Isolation.* The architecture must isolate the kernel from extension failures. Consequently, it must detect failures in the extension before they infect other parts of the kernel.

(2) *Recovery.* The architecture must support automatic recovery to permit applications that depend on a failing extension to continue.

(3) *Backward Compatibility.* The architecture must apply to existing systems and existing extensions, 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 safety 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.

2.2 Functions

We achieve the preceding goals by creating a new operating system *reliability layer* that is inserted between the extensions and the OS kernel. The reliability layer intercepts all interactions between the extensions and the kernel to facilitate isolation and recovery. A crucial property of this layer is *transparency*, i.e., to meet our backward compatibility goals, it must be largely invisible to existing components.

Figure 1 shows this new layer, which we call the *Nooks Isolation Manager* (NIM). Above the NIM is the operating system kernel. The NIM function lines jutting up into the kernel represent kernel-dependent modifications, if any, the OS kernel programmer makes to insert Nooks into a particular OS. These modifications need only be made once. Underneath the NIM is the set of isolated extensions. The

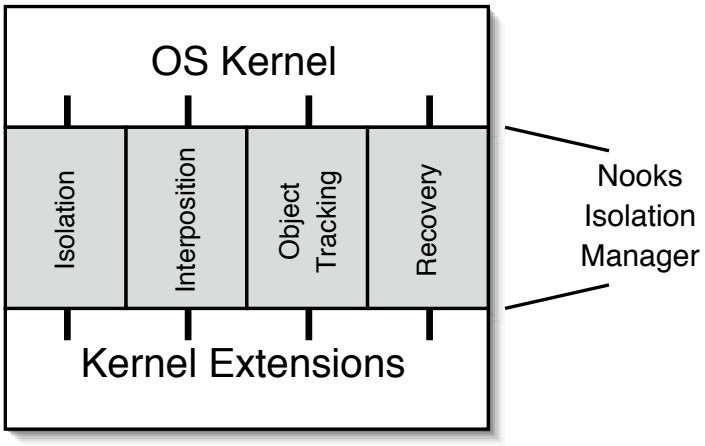

Fig. 1. **The Nooks Isolation Manager, a transparent OS layer inserted between the kernel and kernel extensions.**

function lines jutting down below the NIM represent the changes, if any, the extension writer makes to interface a specific extension or extension class to Nooks. In general, *no* modifications should be required at this level, since transparency for existing extensions is our major objective.

The NIM provides four major architectural functions, as shown in Figure 1: Isolation, Interposition, Object Tracking, and Recovery. We describe each function below.

2.2.1 *Isolation.* The Nooks *isolation mechanisms* prevent extension errors from damaging the kernel (or other isolated extensions). Every extension 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 a limited portion of the kernel's address space.

The major task of the isolation mechanism, then, 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 extension domains and the kernel domain.

Unlike system calls, which are always initiated by an application, the kernel frequently calls into extensions. These calls may generate callbacks into the kernel, which may then generate a call into the extension, and so on. This complex communication style is handled by a new kernel service, called the *Extension Procedure Call* (XPC) – a control transfer mechanism specifically tailored to isolating extensions within the kernel. This mechanism resembles Lightweight Remote Procedure Call (LRPC) [Bershad et al. 1990] and Protected Procedure Call (PPC) in capability systems [Dennis and Horn 1966]. 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 extension's domain

than vice versa).

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

The interface between the extension, the NIM, and the kernel is provided by a set of *wrapper stubs* that are part of the interposition mechanism. Wrappers resemble the stubs in an RPC system [Birrell and Nelson 1984] that provide transparent control and data transfer across address space (and machine) boundaries. Nooks' stubs provide transparent control and data transfer between the kernel domain and extension domains. Thus, from the extension'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 extension's function entry points.

2.2.3 *Object Tracking.* The NIM's *object-tracking functions* oversee all kernel resources used by extensions. In particular, object-tracking code: (1) maintains a list of kernel data structures that are manipulated by an extension, (2) controls all modifications to those structures, and (3) provides object information for cleanup when an extension fails. An extension's protection domain cannot modify kernel data structures directly. Therefore, object-tracking code must copy kernel objects into an extension domain so they can be modified and 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 extension and kernel. Kernel routines can then avoid scrutinizing parameters, executing checks only when called from unreliable extensions.

2.2.4 *Recovery.* Nooks' *recovery functions* detect and recover from a variety of extension faults. Nooks detects a *software fault* when an extension invokes a kernel service improperly (e.g., with invalid arguments) or when an extension consumes too many resources. In this case, recovery policy determines whether Nooks triggers recovery or returns an error code to the extension, which can already handle the failure of a kernel function. Triggering recovery prevents further corruption, but may degrade performance by recovering more frequently. Nooks detects a *hardware fault* when the processor raises an exception during extension execution, e.g., when an extension attempts to read unmapped memory or to write memory outside of its protection domain. Unmodified extensions are of course not in a position to handle their own hardware faults, so in such cases Nooks always triggers a higher level recovery.

Faulty behavior may also be detected from outside Nooks by a user or a program. The user or program can then trigger Nooks recovery explicitly.

Extensions executing in a Nooks domain only access domain-local memory directly. All extension access to kernel resources is managed and tracked through wrappers. Therefore, Nooks can successfully release extension-held kernel structures, such as memory objects or locks, during the recovery process.

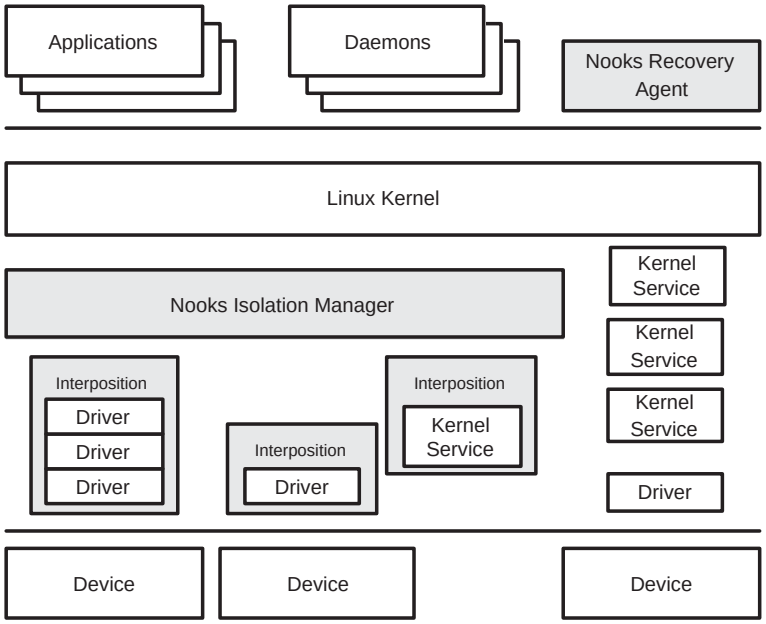

Fig. 2. **The Nooks layer (shaded) inside the Linux OS, showing wrapped Linux extensions executing in isolated protection domains. It is not necessary to wrap all extensions, as indicated by the unshaded extensions on the right.**

3. IMPLEMENTATION

We implemented Nooks inside the Linux 2.4.18 kernel on the Intel x86 architecture. We chose Linux as our platform because of its popularity and its wide support for kernel extensions in the form of loadable modules. Our experience working inside other operating systems, though, including Windows NT, DEC OSF/1, VMS, NetBSD, and Mach, suggests that Linux may be a *worst-case* Nooks target.[1] The Linux 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 extensions directly access fields in many kernel data structures. Despite these challenges, one developer brought the system from concept to function in about 18 months.

The Linux kernel supports standard interfaces for many extension classes. For example, there is a generic interface for block and character devices, 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 extensions take place through function calls, either from the kernel into extensions or from extensions into exported kernel routines. Some global data structures, such as the current task structure, are directly accessed by extensions. Fortunately, extensions modify few of these struc-

[1]Although we developed Nooks on Linux, we expect that the architecture and design could be ported to other operating systems, such as Windows XP or Solaris.

Source Components	# Lines
Memory Management	1,882
Object Tracking	1,454
Extension Procedure Call	770
Wrappers	14,396
Recovery	1,136
Linux Kernel Changes	924
Miscellaneous	2,074
Total number of lines of code	22,266

Table I. **The number of non-comment lines of source code in Nooks.**

tures, and frequently do so through preprocessor macros and inline functions. As a result, Nooks can interpose on most extension/kernel interactions by intercepting the function calls between the extensions and kernel.

Figure 2 shows the Nooks layer inside of Linux. Under the Nooks Isolation Manager are isolated kernel extensions: a single device driver, three stacked 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. Figure 2 also shows unwrapped kernel extensions that continue to interface directly to the Linux kernel.

The NIM exists as a Linux layer that implements the functions described in the previous section. To facilitate portability, we do not use the Intel x86 protection rings or memory segmentation mechanisms. Instead, extensions execute at the same (ring 0) privilege level as the rest of the kernel. Memory protection is provided through the conventional page table architecture and can be implemented both with hardware- and software-filled TLBs.

Table I shows the size of the Nooks implementation. Nooks is composed of about 22,000 lines of code. In contrast, the kernel itself has 2.4 million lines, and the Linux 2.4 distribution has about 30 million [Wheeler 2002]. Other commodity systems are of similar size. For example, various reports relate that the Microsoft Server 2003 operating system contains over 50 million lines of code [Thurrott 2003]. Clearly, relative to a base kernel and its extensions, Nooks' reliability layer introduces only a modest amount of additional system complexity.

In the following subsections we discuss implementation of Nooks' major components: isolation, interposition, wrappers, object tracking, and recovery. We describe wrappers separately because they make up the bulk of Nooks' code and complexity. Finally, we describe limitations of the Nooks implementation.

3.1 Isolation

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

Figure 3 shows the Linux kernel with two lightweight kernel protection domains, each containing a single extension. All components exist in the kernel's address space. However, memory access rights differ for each component: e.g., the kernel

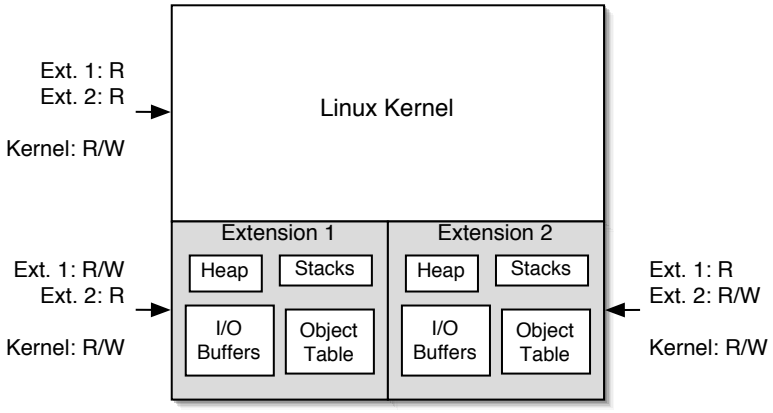

Fig. 3. **Protection of the kernel address space.**

has read-write access to the entire address space, while each extension 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 some single-address-space operating systems [Chase et al. 1994]).

To provide extensions 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 extension, 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 extension.

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

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 on the x86, both from the flush and from subsequent TLB misses. This cost could be mitigated in an architecture with a tagged TLB, such as the MIPS or Alpha, or with single-address-space protection support [Koldinger et al. 1994], such as the IA-64 or PA-RISC. However, because Nooks' lightweight protection domains are kernel tasks that share kernel address space, they minimize the costs of scheduling and data copying on a domain change when compared to normal cross-address space or kernel-user RPCs.

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 appropriate hardware support.

Nooks uses the XPC mechanism to transfer control between extension and kernel domains. XPC is transparent to both the kernel and its extensions, which continue

to interact through their original procedural interfaces. Transparency is provided by means of the wrapper mechanism, described in Section 3.3.

Control transfer in XPC is managed by two functions internal to Nooks: (1) `nooks_driver_call` transfers from the kernel into an extension, and (2) `nooks_kernel_call` transfers from extensions 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. The reverse operations are performed when the call returns.

The performance cost of an XPC is relatively high because it requires changing page tables and potentially flushing the TLB. To ameliorate this cost, XPC also supports *deferred calls*, which batches many calls into a single domain-crossing. Wrappers queue deferred function calls for later execution, either at the entry or exit of a future XPC. For example, we 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 extension calls a wrapper to pass the packet to the kernel. The wrapper queues the packet and batches it with the next few packets that arrive. Function calls such as this can be deferred because there are no visible side effects to the call. Two queues exist for each domain: an extension-domain queue holds delayed kernel calls, and a kernel-domain queue holds delayed extension calls.

In addition to deferring calls for performance reasons, Nooks also uses deferred XPC to synchronize extension modifications to objects explicitly passed from the kernel to extensions. In Linux, the kernel often returns a kernel structure pointer to an extension for structure modification, with no explicit synchronization of the update. The kernel assumes that the modification is atomic and that the extension will update it "in time." In such cases, the wrapper queues a deferred function call to copy the modified object back to the kernel at the extension's next XPC return to the kernel.

We made several one-time changes to the Linux kernel to support isolation. First, to maintain coherency between the kernel and extension page tables, we inserted code wherever the Linux kernel modifies the kernel page table. Second, we 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 directory pointer for the task. On return from exception, the code restores the stack pointer and page table for the extension. Finally, because Linux co-locates the task structure on the kernel stack (which changes as a result of isolation), we had to change its mechanism for locating the current task structure. We currently use a global variable to hold the task pointer, which is sufficient for uniprocessor systems. On a multiprocessor, we would use an otherwise unused x86 segment register, as is done in Windows.

3.2 Interposition

Interposition allows Nooks to intercept and control communication between extensions and the kernel. Nooks interposes on extension/kernel control transfers with

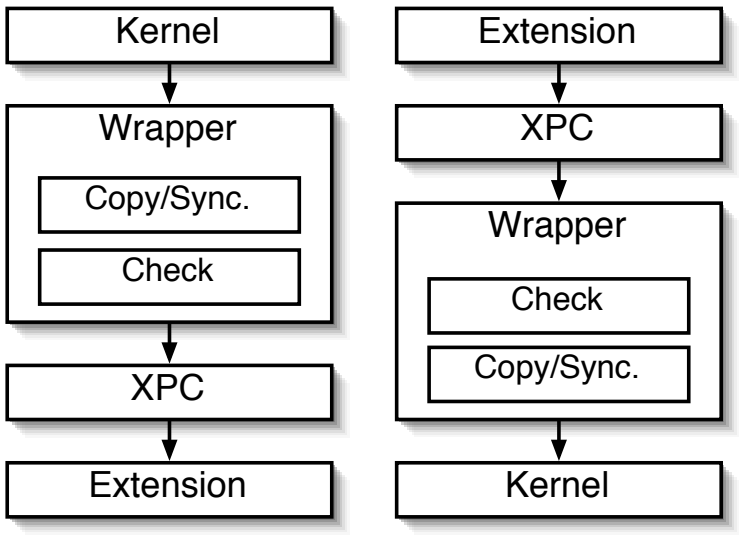

Fig. 4. **Control flow of extension and kernel wrappers.**

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, we modified the standard module loader to bind extensions to wrappers instead of kernel functions when the extensions are loaded. Second, we modified the kernel's module initialization code to explicitly interpose on the initialization call into an extension, enabling the extension to execute within its lightweight protection domain. Following initialization, all function pointers passed from the extension to the kernel are replaced by wrapper pointers. This causes the kernel to call wrappers rather than extension procedures directly.

In addition to interposing on control transfers, Nooks must interpose on some data references. The Linux kernel exports many objects that are only read by extensions (e.g., the current time). These objects are linked directly into the extension so they can be freely read. Other kernel objects are directly written by extensions. We changed macros and inline functions that directly modify kernel objects into wrapped function calls. For object modifications that are not performance critical, Nooks converts the access into an XPC into the kernel. For performance-critical data structures, we create a shadow copy of the kernel object within the extension's domain. The contents of the kernel object and the shadow object are synchronized before and after XPCs into the extension. This technique is used, for example, for the `softnet_data` structure, which contains a queue of the packets sent and received by a network device.

3.3 Wrappers

As noted above, Nooks inserts wrapper stubs between kernel and extension functions. There are two types of wrappers: *kernel wrappers* are called by extensions

to execute kernel-supplied functions; *extension wrappers* are called by the kernel to execute extension-supplied functions. In either case, 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. When an extension 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 (local) procedure call. In the opposite direction, when the kernel calls an extension 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 extension.

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, object-tracker code within wrappers implements *call-by-value-result* semantics for XPC, by creating a copy of kernel objects on the local heap or stack within the extension's protection domain. No marshalling or unmarshalling is necessary, because the extension and kernel share the kernel address space. For simple objects, the synchronization code is placed directly in the wrappers. For more complex objects, such as file system inodes or directory entries that have many pointers to other structures, we wrote explicit synchronization routines to copy objects between the kernel and an extension. Third, wrappers perform an XPC into the kernel or extension to execute the desired function, as shown in Figure 4.

Wrappers are relatively straightforward to write and integrate into the kernel. We developed a tool that automatically generates wrapper entry 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 through XPC. Similarly, for the kernel-to-extension interface, the tool takes a list of interfaces (C structures containing function pointers) and generates wrappers for the kernel to call.

We wrote the main wrapper body functions by hand. This is a one-time task required to support the kernel-extension interface for a specific OS. This code verifies that parameters are correct and moves parameters between protection domains. Once written, wrappers are automatically usable by all extensions that use the kernel's interface. Writing a wrapper requires knowing how parameters are used: whether parameters are alive across calls or are passed to other threads, and which parameters or fields of parameters can be modified. We performed this task by hand, but metacompilation [Engler et al. 2000] could be used to determine the characteristics of extensions by analyzing the set of existing drivers.

3.3.1 *Wrapper Code Sharing.* Section 4 describes the eight extensions we isolated for our Nooks experiments: two sound-card drivers (sb and es1371), four Ethernet drivers (pcnet32, e1000, 3c59x, and 3c90x), a file system (VFAT), and an in-kernel Web server (kHTTPd).

Previously, Table I showed that the Nooks implementation includes 14K lines of wrapper code, over half of the Nooks code base. We implemented 248 wrappers in

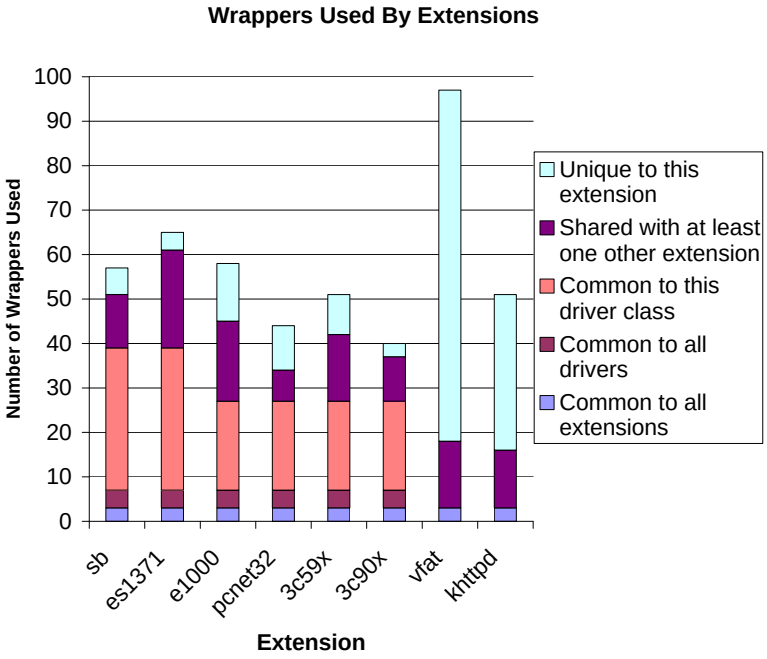

Fig. 5. **Code sharing among wrappers for different extensions.**

all, which we use to isolate 463 imported and exported functions. Wrapper code is thus often shared among multiple drivers in a class or across classes.

Figure 5 shows the total number of wrappers (both kernel and extension 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 adding fault resistance to a given extension. For example, of the 44 wrappers used by the pcnet32 Ethernet driver (31 kernel wrappers and 13 extension wrappers), 27 are shared among the four network drivers. Similarly, 39 wrappers are shared between the two sound-card drivers. Overall, of the 159 wrappers that are not shared, 114 are in the one-of-a-kind extensions VFAT and kHTTPd.

3.4 Object Tracking

The object tracker facilitiates the recovery of kernel objects following an extension failure. The Nooks object tracker performs two independent taskks. First, it records the *addresses* of all objects in use by an extension. Objects used only for the duration of a single XPC call are recorded in a table attached to the current task structure. Objects with long lifetimes are recorded in a per-protection-domain hash table. Second, for objects that may be written by an extension, the object tracker records an association between the kernel and extension versions of the object. This association is used by wrappers to pass parameters between the extension's protection domain and the kernel's protection domain.

The object tracker must know the lifetimes of objects to perform garbage col-

lection, when necessary, or to prevent extensions from using dangling references. Currently, this code can be written only by examining the kernel-extension interface. There are several common paradigms. For example, some objects are accessible to the extension only during the lifetime of a single XPC call from the kernel. In this case, we 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 extension, in which case we know their lifetimes exactly. In still other cases, we go by the semantics of the object and its use. For example, extensions allocate the `timer` data structure to suspend a task. We add this object to the object tracker when an extension calls `add_timer` and remove it when the timer fires, at which point we know that it is no longer used. In some cases, it may be necessary to modify the kernel to notify Nooks when an object is deleted.

Complex objects may be handled in different ways. In some cases, Nooks copies objects into the extension's protection domain, following embedded pointers as appropriate. In other cases, Nooks avoids copying, for example, by mapping network packets and disk blocks into and out of an extension. A "page tracker" mechanism within the object tracker remembers the state of these mapped pages and grants and revokes extension access to the pages.

Our 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, we inspected the interfaces between the kernel and our supported extensions and noted every object that passed through those interfaces. We then wrote object-tracking procedures for each of the 43 object types that we saw. For each object type, there is a unique type identifier and code to release instances of that type during recovery.

3.5 Recovery

Recovery in Nooks consists of two parts. After a fault occurs, the *recovery manager* releases resources in use by the extension. The *user-mode agent* coordinates recovery and determines what course of action to take.

Nooks triggers recovery when it detects a failure through software checks (e.g., parameter validation or livelock detection), processor exceptions, or explicit external signals. After a failure, Nooks suspends the running extension and notifies the recovery manager.

The Nooks recovery manager is tasked with returning the system, including the extension, to a clean state from which it can continue. The recovery manager executes in phases to ensure that resources are not used after they are released. The first phase of recovery is specific to device drivers: Nooks disables interrupt processing for the device controlled by the extension, preventing livelock that could occur if device interrupts are not properly dismissed. It then starts a user-mode recovery agent, which controls the subsequent recovery.

The user-mode recovery agent relies on a configuration database to define the recovery policy for specific extensions or classes of extensions. The agent can perform extension-specific recovery actions as well as notify system managers of the fault. It can also change configuration parameters, replace the extension, or even disable recovery if the extension fails too frequently. The agent requires that many kernel components, such as a file system and disk driver, function properly.

In designing the recovery mechanism, we assume that drivers fail due to transient faults, or "heisenbugs" [Gray 1996], that do not always reproduce. This is evidenced by the fact that drivers typically function correctly after rebooting. We take advantage of the non-reproducing nature of driver faults to simplify our recovery process. Rather than trying to roll back the driver to a previous safe state, we instead completely restart the driver.

By default, the recovery agent initiates full recovery of faulting extensions by unloading the extension, releasing all of its kernel and physical resources, and then reloading and restarting the extension. The agent first calls the recovery manager to release any resources that may be safely reused by the kernel.

The recovery manager signals tasks that are currently executing within the extension, or have called through the extension, to unwind. For a task in a non-interruptible state in either the kernel or another extension, complete recovery may be impossible if the sleeping task never wakes. In this case, partial recovery may still be possible, even though not all processes will continue to execute. Uninterruptible sleeps are infrequent in the Linux kernel, however, so we do not believe this to be a significant limitation. Nooks then releases any kernel resources the extension is using that will not be accessed by an external device. For example, a network device may continue to write to packet buffers in memory; therefore, those buffers cannot be released until the device has been reinitialized.

The recovery manager walks the list of objects known to the object tracker and releases, frees, or unregisters all objects that will not be accessed by external devices. Nooks associates each object type in the tracker with a recovery function. The function releases the object to the kernel and removes all references from the kernel into the extension. If new kernel-extension interfaces are added to Nooks, kernel developers need only add functions to recover new object types used by those interfaces.

Nooks ensures the correctness of kernel data structures after recovery both through the object tracker and through XPC. The use of call-by-value-result ensures that the kernel data structures are updated atomically. The object tracker records all references between extension and kernel data structures and can therefore remove all references to the extension.

After releasing kernel resources, the agent unloads the extension. It then consults policy and may choose to automatically reload the extension in a new lightweight protection domain. The agent then initializes the extension, using the startup scripts that accompany the extension. For device drivers, only after the driver has been reloaded does Nooks finally release all physical resources that could be accessed by the device, such as interrupt request levels (IRQs) and physical memory regions.

3.6 Implementation Limitations

Section 2 described the Nooks philosophy of designing for mistakes and for fault resistance. The Nooks implementation involves many trade-offs. As such, it does not provide complete isolation or fault tolerance for all possible extension errors. Nooks runs extensions in kernel mode to simplify backward compatibility, so we cannot prevent extensions from deliberately executing privileged instructions that corrupt system state. We do not prevent infinite loops inside of the extension, but

we do detect livelock between the extension and kernel with timeouts. Finally, we check parameters passed to the operating system, but we cannot do a complete job given Linux semantics (or lack thereof).

Our current implementation of recovery is limited to extensions that can be killed and restarted safely. This is true for device drivers, which are dynamically loaded when hardware devices are attached to a system. It may not be true for all extensions.

These limitations are not insignificant, and crashes may still occur. However, we believe that our implementation will allow a kernel to resist many crashes caused by extensions. Given the enormous number of such crashes, a fault-resistant solution can have a large impact on overall reliability.

3.7 Achieving Transparency

As mentioned previously, transparency for extensions is a critical goal: Nooks must be able to isolate existing extensions that reflect no knowledge of Nooks. Old extensions must run unchanged, yet benefit from the system's fault isolation and recovery mechanisms.

Nooks' code includes two key component types to facilitate transparency for the extensions we isolated.

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

When an extension is loaded, the loader automatically interposes Nooks' wrapper stubs at the extension-kernel interface. Thereafter, any function call made from the extension to the kernel (or vice versa) transfers instead to a Nooks wrapper. The wrapper intercepts the call, invoking object-tracking code to manage every parameter passed between the caller and callee. Finally, the wrapper transfers control from the caller's domain to the callee's domain using XPC.

Neither the extension nor the kernel is aware of the existence of the Nooks layer.[2] However, the Nooks code isolates the extension and tracks all resources it uses, allowing Nooks to catch errant behavior and to clean up during recovery.

4. EVALUATING RELIABILITY

The thesis of our work is that Nooks can significantly improve system reliability by isolating the kernel from extension failures. This section uses automated experiments to demonstrate that Nooks can detect and automatically recover from faults in extensions. In these tests, Nooks recovered from 99% of extension faults that would otherwise crash Linux.

4.1 Test Methodology

We tested Nooks on a variety of existing kernel extensions and artifically introduced bugs to induce faults.

[2]For the eight extensions we isolated for our experiments reported in the following sections, seven required no changes to run under Nooks, while the eighth (kHTTPd) required changes to only 13 lines of code.

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 tault	Flip a bit in an instruction
NOP fault	Elide an instruction

Table II. **The types of faults injected into extensions and the code transformations used to emulate these faults.**

4.1.1 *Fault Injection.* 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 types of faults into the system. First, we inject faults that emulate specific programming errors common to kernel code according to earlier studies [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 we delete a random instruction.

Table II 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). In our tests, we inject an equal number of each fault type.

4.1.2 *Types of Extensions Isolated.* In the experiments reported below, we used Nooks to isolate three types of extensions: device drivers, a kernel subsystem (VFAT), and an application-specific kernel extension (kHTTPd). The device drivers we chose were common network and sound card drivers, representative of the largest class of Linux drivers.[3] A device driver's interaction with the kernel is well matched to the Nooks isolation model for many reasons. First, drivers invoke the kernel and are invoked by the kernel 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.

[3]Linux has more than 420 sound card drivers and 270 network drivers.

Extension	Purpose
sb	SoundBlaster 16 driver
es1371	Ensoniq sound driver
e1000	Intel Pro/1000 Gigabit Ethernet driver
pcnet32	AMD PCnet32 10/100 Ethernet driver
3c59x	3COM 3c59x series 10/100 Ethernet driver
3c90x	3COM 3c90x series 10/100 Ethernet driver
VFAT	Win95 compatible file system
kHTTPd	In-kernel Web server

Table III. **The extensions isolated and the function that each performs. Measurements are reported for extensions shown in bold.**

In addition to device drivers, we isolated a loadable kernel subsystem. The subsystem we chose was the optional VFAT file system, which is compatible with the Windows 95 FAT32 file system [Microsoft Corporation 2000]. While drivers tend to have a small number of interfaces with relatively few functions, the VFAT interface is larger and more complex than the device drivers'. VFAT has six distinct interfaces that together export over 35 calls; by comparison, the sound and network devices each have one interface with 8 and 13 functions, respectively. 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.

Lastly, we isolated an application-specific kernel extension – the kHTTPd Web server [van de Ven 1999]. kHTTPd resides in the kernel so that it can access kernel network and file system data structures directly, avoiding otherwise expensive system calls. Our experience with kHTTPd demonstrates that Nooks can isolate even ad-hoc and unanticipated kernel extensions.

Overall, we have isolated eight extensions under Nooks, as shown in Table III. We present reliability and performance results for five of the extensions representing the three extension types: sb, e1000, pcnet32, VFAT and kHTTPd. Results for the remaining three drivers are consistent with those presented.

4.1.3 *Test Environment.* Our application-level workload consists of four programs that stress the sound card driver, the network driver, 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 our kernel-level Web server.

We ran our reliability experiments in the context of the VMware Virtual Machine [Sugerman et al. 2001]. The virtual machine allows us to perform thousands of tests remotely while quickly and easily returning the system to a clean state following each one. We spot-checked a number of the VMware trials against a base hardware configuration (i.e., no virtual machine) and discovered no anomalies. In addition, the e1000 tests were run directly on raw hardware, because VMware does not support the Intel Pro/1000 Gigabit Ethernet card.

To measure reliability, we conducted a series of trials in which we injected faults into extensions running under two different Linux configurations. In the first, called

"native," the Nooks isolation services were present but unused.[4] In the second, called "Nooks," the isolation services were enabled for the extension under test. For each extension, we ran 400 trials (50 of each fault type) on the native configuration. In each trial, we injected five random errors into the extension and exercised the system, observing the results. We then ran those same 400 trials, each with the same five errors, against Nooks. It is important to note that our native and Nooks configurations are identical binaries, allowing our automatic fault injector to introduce identical errors. We next describe the results of our experiments.

4.2 Test Results

As described above, we ran 400 fault-injection trials for each of the five measured extensions for 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. We now examine different types of failures that occurred.

4.2.1 *System Crashes.* A system crash is the most extreme and easiest problem to detect, as the operating system either 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. In practice, however, as previously discussed, Nooks may not detect or recover from certain failures caused by very bad programmers or very bad luck.

Figure 6 shows the number of system crashes caused by our fault-injection experiments for each of the extensions running on native Linux and Nooks. Of the 317 crashes observed with native Linux, Nooks eliminated 313, or 99%. In the remaining four crashes the system deadlocked, which Nooks does not handle.

Figure 6 also illustrates a substantial difference in the number of system crashes that occur for VFAT and sb extensions under Linux, compared to e1000, pcnet32 and kHTTPd. This difference reflects the way in which Linux responds to kernel failures. The e1000 and pcnet32 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 code as fatal and crashes the system, hence the large number of crashes in e1000, pcnet32, 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, VFAT and sb would cause system crashes.

4.2.2 *Non-Fatal Extension Failures.* While Nooks is designed to protect the OS from misbehaving extensions, it is not designed to detect erroneous extension behav-

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

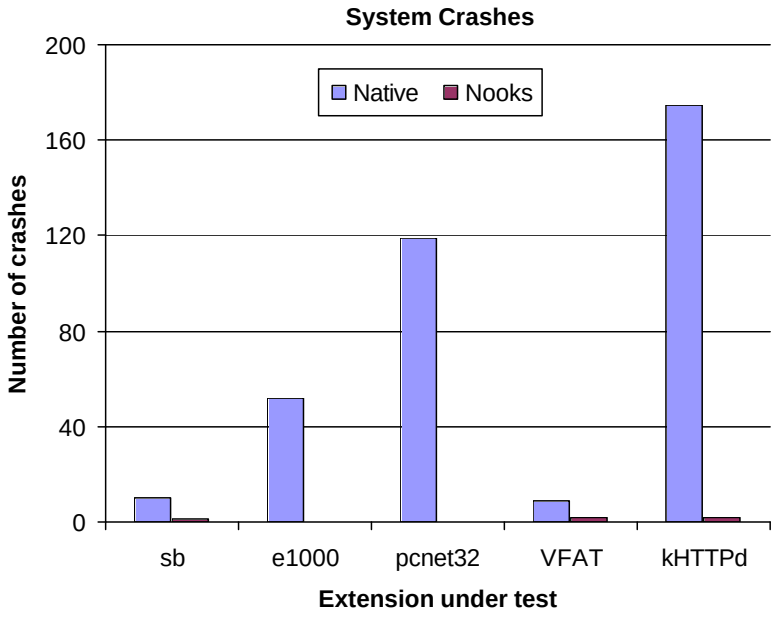

Fig. 6. **The reduction in system crashes in 2000 fault-injection trials (400 for each extension) observed using Nooks. In total, there were 317 system crashes in the native configuration and only four system crashes with Nooks.**

ior. 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.

Our fault-injection trials cause a number of non-fatal extension failures, allowing us to examine Nooks' effectiveness in dealing with these cases, as well. Figure 7 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 previously described. 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 the device was being manipulated by process-oriented code.

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 per-

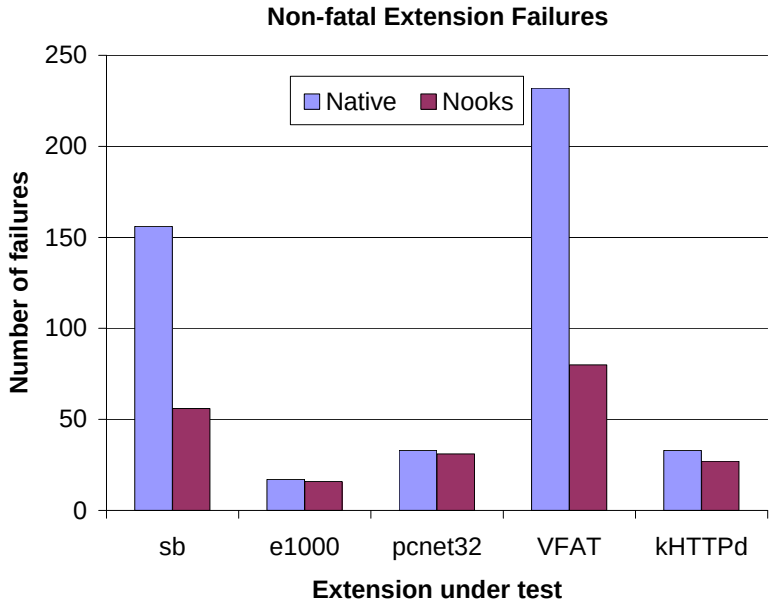

Fig. 7. **The reduction in non-fatal extension failures observed using Nooks. In total, there were 512 such failures in the native configuration and 212 with Nooks.**

formed an extension recovery, thereby allowing the application to continue. 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. These were caught by Nooks and an extension failure was avoided.

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. It is straightforward to develop a "nanny" service that probes for disabled extensions and invokes Nooks' recovery procedures, as appropriate. Alternatively, the failure could be detected by the user, who can then invoke Nooks to initiate a manual recovery.

4.2.3 *Recovery Errors.* The Nooks recovery procedure is straightforward – a faulting extension is unloaded, reloaded, and restarted. For network, sb, and kHTTPd extensions, this process improves reliability directly. For VFAT, however, which deals with persistent state stored on disk, there is some chance that the extension damaged critical on-disk structures *before* Nooks detected an error condition.

In practice, we found that in 90% of the cases, VFAT recovery 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, we 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%. While we would not expect Nooks to do this automatically, it suggests that there may be extensions to Nooks that could improve recovery through the use of application-specific recovery services.

4.2.4 *Manually Injected Errors.* In addition to the automatic fault-injection experiments, we inserted about 10 bugs by hand. Taking the most common fixes for faults reported on the Linux Kernel Mailing List and in the paper by Chou et al. [Chou et al. 2001], we "broke" extensions by removing checks for NULL pointers, failing to properly initialize stack and heap variables, dereferencing a user-level pointer, and freeing a resource multiple times. Nooks automatically detected and recovered from all such failures.

4.2.5 *Latent Bugs.* Nooks revealed several latent bugs in existing kernel extensions. For example, it discovered a bug in the 3COM 3c90x Ethernet driver that occurs during its initialization.[5] Nooks also discovered a bug in another extension, kHTTPd [van de Ven 1999], where an already freed object was referenced. In general, we found that Nooks could be a useful kernel development tool that provides a "fast restart" whenever an extension under development fails.

4.3 Summary of Synthetic Reliability Experiments

Nooks eliminated 99% of the system crashes that occurred with native Linux. The remaining failures directly reflect our 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 recover from many non-fatal extension failures. While Nooks cannot detect many kinds of erroneous behavior, it can trap extension exceptions and initiate recovery in many cases. Overall, Nooks eliminated nearly 60% of non-fatal extension failures caused by our fault injection trials. Finally, Nooks detected and recovered from all of the commonly occurring faults that we injected by hand.

5. PERFORMANCE

This section presents benchmark results that evaluate the performance cost of the Nooks isolation services. Our experiments use existing benchmarks and tools to compare the performance of a system using Nooks to one that does not. Our test machine is a Dell 1.7 GHz Pentium 4 PC running Linux 2.4.18. The machine includes 890 MB of RAM, a SoundBlaster 16 sound card, an Intel Pro/1000 Gigabit Ethernet adapter, and a single 7200 RPM, 41 GB IDE hard disk drive. Our network tests used two similarly equipped machines.[6] Unlike the reliability tests described previously, all performance tests were run on a bare machine, i.e., one without the VMware virtualization system.

[5]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.

[6]We do not report performance information for the slower network adapters to avoid unfairly biasing the results in favor of Nooks.

Benchmark	**Extension**	*XPC Rate (per sec)*	*Nooks Relative Perf. (%)*	*Native CPU Util. (%)*	*Nooks CPU Util. (%)*
Play-mp3	sb	150	100	4.8	4.6
Receive-stream	e1000 (receiver)	10,961	97	39.7	57.8
Send-stream	e1000 (sender)	58,373	97	38.8	81.8
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

Table IV. **The relative performance of Nooks compared to native Linux for six benchmark tests. CPU utilization is accurate to only a few percent. Relative performance is determined either by comparing latency (Play-mp3, Compile-local) or throughput (Send-stream, Receive-stream, Serve-simple-web-page, Serve-complex-web-page). The data reflects the average of three trials with a standard deviation of less than 2%.**

Table IV summarizes the benchmarks used to evaluate system performance. For each benchmark, we used Nooks to isolate a *single* extension, indicated in the second column of the table. We ran each benchmark on native Linux without Nooks and then again on a version of Linux with Nooks enabled. The table shows the relative change in performance for Nooks, either in wall clock time or throughput, depending on the benchmark. We also show CPU utilization measured during benchmark execution, as well as the rate of XPCs per second incurred during each test. The table shows that Nooks achieves between 44% and 100% of the performance of native Linux for these tests.

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). We now look at each benchmark in turn.

5.1 Sound Benchmark

The Play-mp3 benchmark plays an MP3 file at 128 kilobits per second through the system's sound card, generating only 150 XPCs per second. At this low rate, the additional XPC 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 [Haarsten 2000], modems, and sound cards, Nooks offers a clear benefit by improving driver reliability with almost no performance cost.

5.2 Network Benchmarks

The Receive-stream benchmark is an example of a moderate XPC-frequency test. Receive-stream was measured with the netperf [Jones 1995] 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 throughput reduction of about 3% and an overall CPU utilization increase of 18 percentage points.

In contrast, Send-stream (also measured using netperf) is a high XPC-frequency test that isolates the sending node's Ethernet driver. Unlike the Receive-stream test, which benefits from the batching of received packets, the OS does not batch outgoing packets that it sends. Therefore, although the total amount of data transmitted is the same, Send-stream executes nearly an order of magnitude more XPCs per second than Receive-stream. The overall CPU utilization on the sender thus increases from about 39% on native Linux to 81% with Nooks. As with the Receive-stream benchmark, throughput drops by about 3%. 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 Nooks overhead. Nevertheless, on slower processors or faster networks, it may be worthwhile to batch outgoing streaming packets as is done, for example, with network terminal protocols [Gettys et al. 1900].

5.3 Compile Benchmark

As an indication of application-level file system performance, we measured the time to untar and compile the Linux kernel on a local VFAT file system. Table IV shows that the compilation ran about 10% slower when VFAT was isolated with Nooks. In this case, the CPU was nearly 100% utilized in the native case, so the additional overhead due to Nooks is directly reflected in the end-to-end execution time.

We take this benchmark as an opportunity to analyze the causes of Nooks' slowdown. 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, we profiled the compile-local benchmark on the native Linux kernel and Linux with Nooks isolation.

We used statistical profiling of the kernel to break the execution time of the benchmark into components. The results are shown in Figure 8. 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 exeuted without Nooks. The new code in Nooks (shown by the upper bars in the Figure 8), accounts for 46 of the additional 72 seconds spent in the kernel under Nooks.

Of the 46 seconds spent in Nooks, more than half half (28 seconds) is due just to XPC. Object tracking is another large cost (6 seconds), because Nooks consults a hash table to lookup 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 appear 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' recovery support is reflected in terms of the time spent tracking

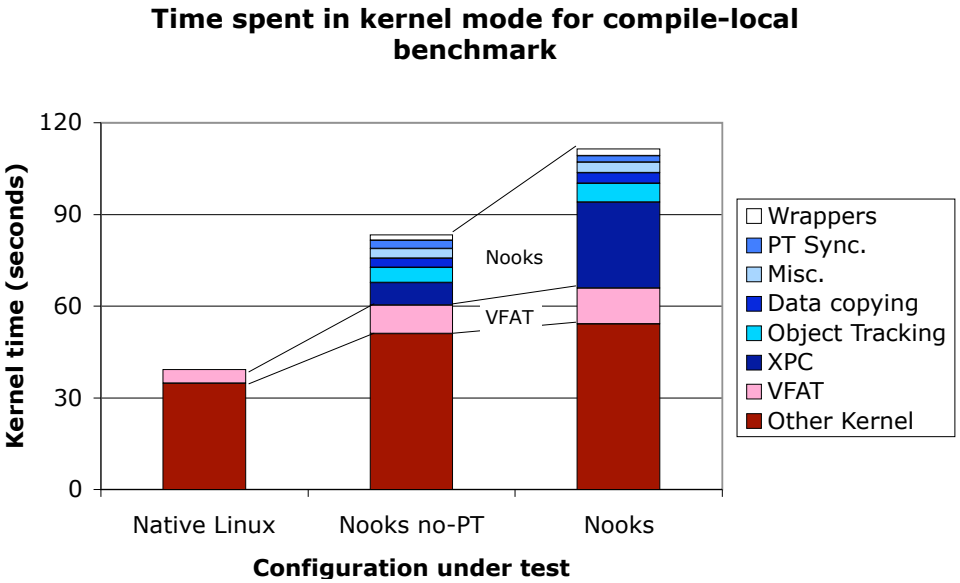

Fig. 8. **Comparative times spent in kernel mode for the Compile-local (VFAT) benchmark. During the run with Nooks, the system performed 10,934,567 XPCs into the kernel and 4,0086,586 XPCs into the extension. Time in user mode (not shown) was identical for both configurations (477 seconds).**

objects that occurs on every XPC in which a pointer parameter is passed. Tracking allows the Nooks recovery manager to correctly 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 8 show the execution time for native Linux code in the kernel (labelled *other kernel*) and in the VFAT filesytem. For native Linux, about 35 seconds were spent in the kernel proper and about 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 code execuing 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.

We 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 futher understand this effect, we 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 8, the Nooks kernel without a separate page table (Nooks no-PT) spent 51 seconds in the kernel proper and 9 seconds was in VFAT. 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, we speculate that the remaining performance difference between native Linux and Nooks is due to additional TLB and cache misses that occur because we copy data between the kernel and the filesystem and execute on a separate stack. The Pentium 4 has just an 8KB L1 data cache, so it is particularly sensitive to additional memory accesses.

We 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 more finely tuned data structures [Schroeder and Burrows 1989; Draves et al. 1991] and superpages to contain all the data for a protection domain.

5.4 Web Server Benchmarks

The final two benchmarks illustrate the impact 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. We used httperf [Mosberger and Jin 1998] 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 Send-stream benchmark, which exhibits roughly the same rate of XPCs but without saturating the CPU, degrades by only 10%. In addition, Send-stream 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 already a bottleneck 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. We 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 [Standard Performance Evaluation Corporation 1999] against the user-mode Apache 2.0 Web Server [Apache Project 2000], with and without Nooks isolation of the 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 [7]. With Nooks installed and the Ethernet driver isolated on the server, peak throughput dropped by about 3%, to 110 requests per second.

5.5 Summary

This section used a small set of benchmarks to quantify the performance cost of Nooks. For the sound and Ethernet drivers tested, Nooks imposed a performance penalty of less than 10%. For kHTTPd, an ad-hoc application extension, the penalty was nearly 60%. A key factor in the performance impact is the number of XPCs required, as XPCs impose a burden, particularly on the x86 TLB in our 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.

Overall, Nooks provides a substantial reliability improvement at a cost that depends on the extension being isolated. The reliability/performance trade-off is thus one that can be made on a case-by-case basis. For many computing environments, given the performance of modern systems, we believe that the benefits of Nooks' isolation and recovery services are well worth the costs.

6. RELATED WORK

Our work differs from the substantial body of research on extensibility and reliability in many dimensions. 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 recoverability, and to impose only a modest performance penalty.

Hardware support for modularity

The major hardware approaches to improve reliability include capability-based architectures [Houdek et al. 1981; Organick 1983; Levy 1984] and ring and segment architectures [Intel Corporation 2002; Saltzer 1974]. [8] These systems support fine-grained protection, enabling construction and isolation of privileged subsystems. The OS is extended by adding new privileged subsystems that exist in new domains or segments. Recovery is not specifically addressed in either architecture. In particular, capabilities support the fine-grained sharing of data. If one sharing component fails, recovery may be difficult for others sharing the same resource. Segmented architectures have been difficult to program and plagued by poor performance. In contrast, Nooks isolates existing code on commodity processors using standard virtual memory and runtime techniques, and it supports recovery through garbage collection of extension-allocated data.

[7] The test configuration is throughput limited due to its single IDE disk drive.
[8] [Witchel et al. 2002] presents a similar approach in a newer context.

Operating system support for isolation and recovery

Several projects have isolated kernel components through new operating system structures. Microkernels [Wulf 1975; Liedtke 1995; Young et al. 1986] and their derivatives [Engler et al. 1995; Ford et al. 1997; Hand 1999] promise another path to reliability. These systems isolate extensions into separate address spaces that interact with the OS through a kernel communication service, such as messages or remote procedure call [Bershad et al. 1990]. Therefore, the failure of an extension within an address space does not necessarily crash the system. However, as in capability-based systems, recovery has received little attention in microkernel systems. In Mach, for example, a user-level system service can fail without crashing the kernel, but rebooting is often the only way to restart the service. Despite much research in fast inter-process communication (IPC) [Bershad et al. 1990; Liedtke 1995], the reliance on separate address spaces raises performance concerns that have prevented adoption in commodity systems. Microkernel/monolithic hybrids, such as L^{4}Linux [Härtig et al. 1997], provide much of the isolation support needed for reliability, but are more difficult to integrate into existing code bases.

In the past, virtual memory techniques have been used to isolate specific components or data from corruption, e.g., in a database [Sullivan and Stonebraker 1991] or in the file system cache [Ng and Chen 1999]. Nooks uses similar techniques to protect the operating system from erroneous extension behavior.

Virtual machine technologies [Chapin et al. 1995; Chen and Noble 2001; Sugerman et al. 2001; Whitaker et al. 2002] have been proposed as a solution to the reliability problem. They can reduce the amount of code that can crash the whole machine. Virtualization techniques typically run several entire operating systems on top of a virtual machine, so faulty extensions in one operating system cause only a few applications to fail. However, if the extension executes in the virtual machine monitor, such as device drivers for physical devices, a fault causes all virtual machines *and* their 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. The challenge for reliable extensibility is not in virtualizing the underlying hardware; rather it lies in virtualizing only the *interface* between the kernel and extension. In fact, this is a major feature of the Nooks architecture.

A number of transaction-based systems [Schmuck and Wylie 1991; Seltzer et al. 1996] have applied recoverable database techniques 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 [Schmuck and Wylie 1991]. Like the language-based approaches, these strategies have limited applicability. In contrast, Nooks integrates transparently into existing hardware and operating systems.

Compiler and language support for reliability

An alternative to operating system-based isolation is the use of type-safe programming languages and run-time systems [Bershad et al. 1995] that prevent many faults from occurring. Such systems can provide performance advantages, since compile-time checking enables lightweight run-time structures (e.g., local procedure calls rather than cross-domain calls). To date, however, OS suppliers have been unwill-

ing 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. In contrast, Nooks requires no specialized programming language.

Recent years have seen the development of software techniques that enforce code correctness properties, e.g., software fault isolation [Wahbe et al. 1993] and self-verifying assembly code [Necula and Lee 1996]. These technologies are attractive and might replace or augment some of Nooks' isolation techniques. Nevertheless, in their proposed form, they deal only with the isolation problem, leaving unsolved the problems of transparent integration and recovery. Recently, techniques for verifying the integrity of extensions in existing operating systems have proven effective at revealing programming errors [Engler et al. 2000; DeLine and Fähndrich 2001; Ball and Rajamani 2001; Condit et al. 2003]. This static approach obviously complements our own dynamic one.

The Devil project [Mérillon et al. 2000] takes a different approach, ensuring that drivers interact with devices correctly. In Devil, a device vendor would specify 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 removes 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 is complementary to Nooks, in that it removes many of the bugs in drivers, but requires writing new drivers to use the generated interface.

Recovery

More recently, researchers have begun to focus on *recovery* as a general technique for dealing with failure in complex systems [Patterson et al. 2002]. For example, Candea, in Candea and Fox [2001], proposes a model of *recursive recovery*; in the model a complex software system is decomposed into a multi-level implementation where each layer can fail and recover independently. Nooks is complementary, although our focus to date has been limited to restarting portions of operating system kernels.

Other systems have focused on recovery from faults in existing code, such as discount checking [Lowell and Chen 1998; Lowell et al. 2000]. Discount checking recovers from faults in user-level programs automatically by snapshotting state periodically, and reverting back to a previous snapshot following a failure. Nooks, in contrast, complete restarts failed kernel extensions. Wrappers have been used for reliability and recovery in other systems. Fabre, in Fabre et al. [2000] and Fetzer, in the Healers project Fetzer and Xiao [2003] use wrappers similar to Nooks' around existing code to both tolerate and recover from faults automatically. These wrappers can verify pre- and post-conditions, catch exceptions, and retry function calls after a failure. Unlike Nooks, these systems do not incorporate memory isolation and hence do not prevent accidental memory corruption.

Table V shows the changes to hardware architecture, operating system architecture, or extension architecture required by previous approaches to reliability. Only Nooks, virtual machines, and static analysis techniques need no architectural changes.

In summary, Nooks brings to commodity operating systems the well-known re-

Approach	Required Modifications		
	Hardware	OS	Extension
Capabilities	yes	yes	yes
Microkernels	no	yes	yes
Languages	no	yes	yes
New Driver Architectures	no	yes	yes
Transactions	no	no	yes
Virtual Machines	no	no	no
Static Analysis	no	no	no
Nooks	no	no	no

Table V. **Components that require architectural changes for various approaches to reliability. A "yes" in a cell indicates that the reliability mechanism on that row requires architectural change to the component listed at the top of the column.**

quirements for fault toleratant operating systems [Denning 1976]: isolation, resource control, decision verification (checking), and error recovery. Nooks provides these features for extensions in a way that is compatible and transparent to existing code.

7. CONCLUSIONS

Kernel extensions are a major source of failure in modern operating systems. Nooks is a new reliability layer intended to significantly reduce extension-related failures. Nooks uses hardware and software techniques to isolate kernel extensions, trapping many common faults and permitting extension recovery. The Nooks system focuses on achieving *backward compatibility*, that is, it sacrifices complete isolation and fault tolerance for compatibility and transparency with existing kernels and extensions. Nevertheless, Nooks demonstrates that it is possible to realize an extremely high level of operating system reliability with a performance loss ranging from zero to just over 60%. Our fault-injection experiments reveal that Nooks recovered from 99% of the faults that caused native Linux to crash.

Our 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) extensions such as device drivers can be isolated without change to extension code, and (3) isolation and recovery can dramatically improve the system's ability to survive extension faults.

Overall, our experiments demonstrate that Nooks defines a new point in the reliability/performance space beyond simple kernel mode and user mode. In today's world, nearly all extensions run in the kernel and are potential threats to reliability. Nooks offers kernel developers a substantial degree of reliability with a cost ranging from negligible to significant. The decision to isolate a kernel extension should be made in light of that extension's native reliance on kernel services, its bottleneck potential, and the environment in which it will be used.

Clearly, for many device drivers and low XPC-frequency extensions, the decision is easy. For others, it is a question of requirements. Where performance matters more than reliability, isolation may not be appropriate. However, given the impres-

sive performance of current processors and the enormous rate at which performance is increasing, many devices are in the "easy decision" category today, and more will join that category with each passing year.

Acknowledgments

This work was supported in part by the National Science Foundation under grants ITR-0085670, CCR-0121341 and ITR-0326546. We appreciate the efforts of Steve Martin and Doug Buxton for their help in developing the wrapper-generating tool and testing Nooks, Leo Shum for adding sound card support, and Christophe Augier for his work on the recovery agent and on reliability testing. We would like to thank Intel and Microsoft for information on their respective products. We would also like to thank Frans Kaashoek and the many anonymous referees for their suggestions which have improved the content and presentation of the paper.

REFERENCES

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

Ball, T. and Rajamani, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*. LNCS, vol. 2057. 103–122.

Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. 1990. Lightweight remote procedure call. *ACM Transactions on Computer Systems 8,* 1 (Feb.), 37–55.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, Colorado, 267–284.

Birrell, A. D. and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2,* 1 (Feb.), 39–59.

Bovet, D. P. and Cesati, M. 2001. *Understanding the Linux Kernel*. O'Reilly.

Candea, G. and Fox, A. 2001. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the Eighth IEEE HOTOS*. 125–132.

Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., and Gupta, A. 1995. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, Colorado, 12–25.

Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. 1994. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems 12,* 4 (Nov.), 271–307.

Chen, P. and Noble, B. 2001. When virtual is better than real. In *Proceedings of the Eighth IEEE HOTOS*. 133–138.

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 1996 Symposium on Fault-Tolerant Computing (FTCS)*. IEEE, Sendai, Japan, 304 – 313.

Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W. 2003. CCured in the real world. In *Proceedings of the ACM SIGPLAN '03 ACM Conference on Programming Language Design and Implementation*. San Diego, California, USA, 232–244.

Custer, H. 1993. *Inside Windows NT*. Microsoft Press, Redmond, WA.

DeLine, R. and Fähndrich, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN '01 ACM Conference on Programming Language Design and Implementation*. Snowbird, Utah, 59–69.

Denning, P. J. 1976. Fault tolerant operating systems. *ACM Computing Surveys 8,* 4 (Dec.), 359–389.

Dennis, J. B. and Horn, E. V. 1966. Programming semantics for multiprogramming systems. *Communications of the ACM 9,* 3 (Mar.).

Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. 1991. Using continuations to implement thread management and communications in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, 122–136.

Engler, D., Chelf, B., Chou, A., and Hallem, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*. San Diego, CA, 1–16.

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.

Fabre, J.-C., Rodrí, M., Arlat, J., Salles, F., and Sizun, J.-M. 2000. Building dependable COTS microkernel-based systems using MAFALDA. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing (PRDC 00)*. Los Angeles, California, 85–94.

Fabry, R. S. 1974. Capability-based addressing. *Communications of the ACM 17,* 7 (July), 403–412.

Fetzer, C. and Xiao, Z. 2003. HEALERS: A toolkit for enhancing the robustness and security of existing applications. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN'03)*. San Francisco, California, 317–322.

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.

Forin, A., Golub, D., and Bershad, B. 1991. An I/O system for Mach. In *Proc. Usenix Mach Symposium*. 163–176.

Gettys, J., Carlton, P. L., and McGregor, S. 1900. The X window system version 11. Tech. Rep. CRL-90-08, Digital Equipment Corporation. Dec.

Gillen, A., Kusnetzky, D., and McLaron, S. 2002. The role of Linux in reducing the cost of enterprise computing. IDC white paper.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison-Wesley.

Gray, J. 1996. 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*. IEEE, Los Angeles, California, 3–12.

Haarsten, J. C. 2000. The Bluetooth radio system. *IEEE Personal Communications Magazine 7,* 1 (Feb.), 28–36.

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.

Härtig, H., Hohmuth, M., Liedtke, J., Schöberg, S., and Wolter, J. 1997. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, 66–77.

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

Houdek, M. E., Soltis, F. G., and Hoffman, R. L. 1981. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th ACM International Symposium on Computer Architecture*. ACM/IEEE, 341–348.

Hsueh, M., Tsai, T. K., and Iyer, R. K. 1997. Fault injection techniques and tools. *IEEE Computer 30,* 4 (Apr.), 75–82.

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

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

Koldinger, E. J., Chase, J. S., and Eggers, S. J. 1994. Architectural support for single address space operating systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 175–186.

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

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

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, CA.

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.

Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. 2000. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*. San Diego, CA, 17–30.

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

Mosberger, D. and Jin, T. 1998. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*. ACM, Madison, WI, 59—67.

Necula, G. C. and Lee, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*. Seattle, Washington, 229–243.

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

Organick, E. I. 1983. *A Programmer's View of the Intel 432 System*. McGraw Hill.

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. Technical Report CSD-02-1175, UC Berkeley Computer Science. Mar.

Project-UDI. 1999. Introduction to UDI version 1.0. Tech. rep., Project UDI. Aug.

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

Schmuck, F. and Wylie, J. 1991. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. Pacific Grove, California, 239–253.

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

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.

Short, R. 2003. Vice President of Windows Core Technology, Microsoft Corp. Private communication.

Standard Performance Evaluation Corporation. 1999. The SPECweb99 benchmark.

Sugerman, J., Venkitachalam, G., and Lim, B. 2001. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*. Boston, MA.

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 1991 Symposium on Fault-Tolerant Computing (FTCS-21)*. IEEE, Montreal, Que., Canada, 2–9.

Sullivan, M. and Stonebraker, M. 1991. 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*. Morgan Kaufman Publishing, 171–180.

Thurrott, P. 2003. Windows 2000 server: The road to gold, part two: Developing windows. *Paul Thurrott's SuperSite for Windows*.

TiVo Corporation. 2001. TiVo digital video recorder. www.tivo.com.

van de Ven, A. 1999. kHTTPd: Linux HTTP accelerator. Available at http://www.fenrus.demon.nl/.

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.

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

Whitaker, A., Shaw, M., and Gribble, S. D. 2002. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*. Boston, MA, 195–209.

Witchel, E., Cates, J., and Asanović, K. 2002. Mondrian memory protection. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 304–316.

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.