Published in ASPLOS XIII: Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008

The Design and Implementation of Microdrivers

Vinod Ganapathy [*], Matthew J. Renzelmann, Arini Balakrishnan[†], Michael M. Swift, Somesh Jha

University of Wisconsin-Madison [*]Rutgers University [†]Sun Microsystems

vinodg@cs.rutgers.edu, {mjr, arinib, swift, jha}@cs.wisc.edu

Abstract

Device drivers commonly execute in the kernel to achieve high performance and easy access to kernel services. However, this comes at the price of decreased reliability and increased programming difficulty. Driver programmers are unable to use user-mode development tools and must instead use cumbersome kernel tools. Faults in kernel drivers can cause the entire operating system to crash. User-mode drivers have long been seen as a solution to this problem, but suffer from either poor performance or new interfaces that require a rewrite of existing drivers.

This paper introduces the Microdrivers architecture that achieves high performance and compatibility by leaving critical path code in the kernel and moving the rest of the driver code to a user-mode process. This allows data-handling operations critical to I/O performance to run at full speed, while management operations such as initialization and configuration run at reduced speed in user-level. To achieve compatibility, we present DriverSlicer, a tool that splits existing kernel drivers into a kernel-level component and a user-level component using a small number of programmer annotations. Experiments show that as much as 65% of driver code can be removed from the kernel without affecting common-case performance, and that only 1-6 percent of the code requires annotations.

Categories and Subject Descriptors. D.4.5 [*Operating Systems*]: Reliability; D.4.7 [*Operating Systems*]: Organization and Design

General Terms. Management, Reliability

Keywords. Device Drivers, Reliability, Program Partitioning

1. Introduction

Recent systems such as Nooks [36], SafeDrive [43], and Xen [15] aim to improve operating system reliability by isolating the kernel from buggy device drivers. However, these systems do not address a key aspect of the driver reliability problem: writing kernel code is harder than writing user-mode code. Most advanced software engineering tools available today apply only to user-mode code, and debugging kernel code often requires remote debugging on a second machine. A large step toward addressing this problem is to move drivers out of the kernel [8, 11, 20, 25, 37].

Executing drivers in user mode provides several important benefits. First, driver developers can take advantage of user-mode tools such as profilers, libraries, scripting languages and advanced debuggers. While such tools are available aplenty for user-mode programming, kernel programming represents a smaller and a more challenging target; consequently, fewer and less polished tools are available. Second, the user/kernel boundary isolates the kernel from driver failures. For example, a null-pointer dereference or a deadlock in a user-mode driver will kill or hang its process but leave the kernel unaffected.

However, existing user-mode drivers frameworks suffer from one of two problems. First, previous attempts to execute unmodified drivers in user mode performed poorly [38]. The existing driver/kernel interface significantly limits the performance achievable in user mode. Moving the driver to user mode may require frequent domain transitions that move large data structures. The interface was written expecting local procedure calls, so communication is frequent and inefficient. For example, the networking stack calls into a driver separately to send each packet rather than batching a group of related packets. Similarly, it may pass in large data structures from which the driver only needs a single field.

Second, user-mode driver frameworks that achieve high performance require complete rewrites of the driver [18, 20, 25]. Each of these systems has a different interface than kernel drivers on the corresponding platforms. As a result, they provide little or no benefit to the tens of thousands of existing drivers and require driver writers to learn completely new programming models. Furthermore, even high-performance driver frameworks incur additional costs when entering user mode; consequently, drivers with demanding performance requirements, such as software modems, will remain in the kernel to avoid that overhead.

The path to user-mode drivers that perform well and are compatible with existing code comes from a better understanding of drivers. Conventional wisdom is that most driver code moves data between memory and an external device. In reality, data handling is a small fraction of the code in a driver. For example, in the e1000 gigabit Ethernet driver, only 10% of the code directly relates to sending and receiving packets [16]. Most of the driver code is device initialization and cleanup, management, and error handling. This code controls the driver and the device but is not on the critical path when moving data between memory and the device.

Thus, rather than consider drivers monolithically, we can partition them as a high-performance component that handles the data path and a low-performance component that handles the control path. Only the performance critical data-path code need run in the kernel. The remaining code may be moved into a separate user-level protection domain to aid development and improve fault isolation without significantly impacting performance.

The result is a new architecture for device drivers that we call *Microdrivers*. A microdriver is a device driver that is split into a kernel-level *k-driver* and a user-level *u-driver*. Critical path code, such as I/O, and high-priority functions, such as interrupt handling, are implemented in the k-driver. This code enjoys the full speed of a purely kernel driver. The remaining code, which is invoked infrequently, is implemented in the u-driver and executes outside the kernel in a user-mode process. When necessary, the k-driver may invoke the u-driver. This architecture yields four concrete benefits over traditional kernel drivers:

(**1**) *User-level programming.* The u-driver can be compiled and debugged with standard user-mode tools. We show that the u-driver

[* †] Work done while at the University of Wisconsin-Madison

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

can be debugged with standard source debuggers and memory profilers.

(**2**) *Good common-case performance.* Microdrivers have common-case performance comparable to traditional device drivers and better than kernel driver code executing fully in user mode. We show that performance of a network microdriver with 65% of the driver code in the u-driver has indistinguishable network performance from a monolithic kernel driver.

(**3**) *Fault isolation.* Microdrivers isolate faults better than traditional device drivers, because a buggy u-driver will not crash the entire system. For example, we show that the kernel survives a null-pointer dereference in a u-driver.

(**4**) *Compatibility.* Microdrivers are compatible with commodity operating systems. They expose the same interface to the kernel as traditional device drivers and can be created from existing drivers.

Thus, microdrivers achieve much of the programmability and isolation properties of user-mode drivers for a large fraction of code in a driver, yet provide the performance of kernel-mode drivers.

While we can apply this architecture to build new drivers, we would also like to allow existing drivers to benefit from the Microdrivers architecture. To this end, we present a tool called *DriverSlicer* for mostly automatic conversion of existing drivers to microdrivers. DriverSlicer has two parts, a *splitter* and a *code generator*.

(**1**) DriverSlicer's splitter partitions driver code such that performance critical functions remain in the kernel. The split aims to minimize the cost of moving data and control along the performance-critical path. Rarely-used functions, such as those for startup, shutdown and device configuration are relegated to the u-driver.

(**2**) DriverSlicer's code generator emits marshaling code to pass data between the microdriver components. Unlike traditional marshaling code for RPC, this tool employs static analysis to reduce the amount of data copied. When a complex structure crosses the user/kernel boundary, the analysis allows only the fields actually accessed on the other side to be copied. In addition, the analysis allows global variables to be automatically shared through copying. During code generation, this tool may prompt a programmer for annotations when it cannot automatically determine pointer semantics, such as the length of an array.

We evaluate the Microdrivers architecture and DriverSlicer on drivers from the Linux 2.6.18.1 kernel. We demonstrate with common benchmarks that (1) U-Drivers can be written and debugged with common user-level programming tools, (2) Microdrivers achieve common-case performance similar to native kernel drivers, (3) Microdrivers isolate failures in the u-driver from the kernel, and (4) Microdrivers reduce the amount of driver code running in the kernel.

In the next section we present background on device drivers. Following in Section 3 we present the Microdrivers architecture. Section 4 discusses the implementation of microdrivers, and Section 5 presents DriverSlicer. We evaluate Microdrivers in Section 6, present related work in Section 7 and finally conclude.

2. Background

A device driver is a module of code that translates requests from the kernel or an application into I/O requests to a specific device. The kernel or application may provide a standard interface that multiple drivers implement, or the driver interface may be unique. In addition, combinations are possible: a mix of common interfaces plus device-specific functionality, often hidden behind an `ioctl` function.

While device drivers have been an established part of operating systems since their origin, they have emerged as one of the greatest sources of system problems. Device drivers are one of the largest causes of system downtime. Microsoft reports that 89% of crashes of Windows XP are caused by device drivers [27], and a study of the Linux kernel showed that driver code has 2 to 7 times the bug density of other kernel code [7]. This may be due to the difficulty of writing and debugging kernel code, the unavailability of advanced software engineering tools for kernel programming, as well as the comparative lack of experience of driver writers as compared to core kernel developers. Driver programmability and system reliability can therefore be improved by removing driver code from the kernel.

Consider for instance the e1000 gigabit Ethernet driver, with 274 functions and 15,100 lines of code. For this driver, we identified (using DriverSlicer) that just 25 functions with 1,550 lines of code (10%) must execute in the kernel: these include functions that must execute at high priority, such as the interrupt handler, and major data handling routines, such as functions related to sending networking packets. The remaining functions are called only occasionally (*e.g.,* during device startup/shutdown, device configuration, and for diagnostics) and can be removed from the kernel without affecting common-case performance.

We also studied the revision history of this driver to evaluate whether the benefits of moving code out of the kernel are illusory, in that the effort in writing drivers remains in the k-driver. For each revision to the driver, we determined which portion of the driver changed. We found that of 703 patches, 125 were in the k-driver and 578, or 82%, were in the u-driver. We were not able to distinguish bug fixes from feature or performance enhancements. Nonetheless, these results indicate that splitting drivers has the potential to dramatically improve fault isolation and improve programmability. The following sections describe the Microdrivers architecture for achieving this goal.

3. Architecture of a microdriver

The Microdrivers architecture seeks, above all, a practical approach to improve system reliability. We identify four major goals for the architecture:

(**1**) *User-mode programming tools.* Microdrivers should allow programmers to use common tools such as source-level debuggers on drivers.

(**2**) *Good common-case performance.* Microdrivers should have common-case performance comparable to traditional device drivers, for example similar throughput, latency, and CPU overhead.

(**3**) *Fault isolation.* Microdrivers should isolate faults better than traditional device drivers. A fault in the user-level component should not trigger a system-wide failure.

(**4**) *Compatibility.* Microdrivers should be compatible with commodity operating systems. That is, they must expose the same interface to the operating system kernel as traditional device drivers.

These goals are at times contradictory, in that performance, user-level programming, and 100% compatibilty cannot all be achieved simultaneously. Instead, the Microdrivers architecture is a compromise that leaves some driver code in the kernel to achieve performance and compatibility. Previous systems, such as Nooks [36] and SafeDrive [43] achieve the latter three goals but do not support user-mode tools. User-mode driver frameworks, such as Fuse [11] and Microsoft's UMDF [25], achieve fault isolation and user-mode programming, but sacrifice some performance and all compatibility.

3.1 Splitting drivers

The approach we take is to factor device drivers into a user-mode component, called a *u-driver* and a kernel-mode portion, called a *k-driver*. The u-driver executes in a separate process from the caller, which may be shared between multiple drivers for efficiency. The

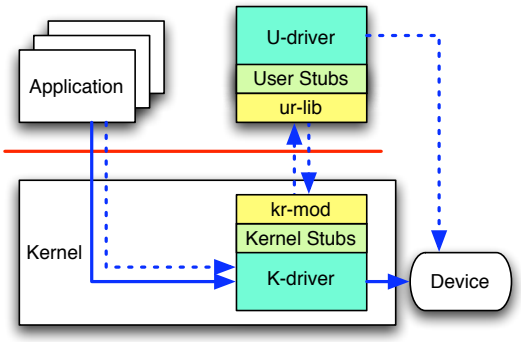

Figure 1. The Microdrivers architecture. The path for performance-critical functions is a solid line, and for non-critical functions is dashed.

kernel always invokes the k-driver, which may in turn invoke the u-driver for selected functions. These calls appear as local calls to the k-driver but are implemented with an RPC-like mechanism that calls up into the u-driver. The u-driver may also call down into the k-driver to invoke its functions. Figure 1 illustrates the architecture of a microdriver. The most important decision when applying this architecture is where to split the code between the k-driver and the u-driver. Ideally, the boundary should have few functions, be crossed infrequently, and pass little data.

The Microdrivers architecture splits drivers along performance and priority boundaries. Several operating systems already factor driver code into device-dependent code and device-independent code, such as miniport drivers in Windows [26] and families in the MacOS I/O Kit [1]. However, this separation does not account for the performance requirements of a driver. The components of these drivers communicate frequently and move large amounts of data. As a result, existing interfaces are not suitable for splitting drivers between kernel and user mode.

Code required for performance-critical operations, such as transmitting and receiving data, are left in the kernel portion. This code is often the *data path* of a driver. Non-critical operations, such as initializing the device, configuring the device, and handling errors, are placed in a user-mode process. This is often the *control path*. Other performance-critical functions may be left in the kernel as well to improve performance. Both components may access the physical device to avoid the need for frequent calls between the two components.

In the Microdrivers architecture, the kernel interface to a driver remains unchanged. This preserves compatibility with existing operating systems. Within the driver, individual functions may reside in the kernel or in user-mode. We take an RPC approach, in that client stubs in one portion invoke a runtime library to communicate with the other portion. This library is additionally responsible for synchronizing data structures.

This architecture can be realized by writing new drivers to fit this model if the programmer explicitly selects the code for the u-driver and k-driver. However, to preserve the investment in existing drivers, tools can assist this process. In Section 5, we describe DriverSlicer, a tool that largely automates the conversion of monolithic kernel drivers into microdrivers.

3.2 Runtime

When a driver is factored into a k-driver and a u-driver, the two portions must communicate and share data. In the Microdrivers architecture, runtime layers in the kernel and in user-mode provide these services. This runtime provides three key functions:

- *Communication*. Provide control and data transfer to allow each portion of the driver to invoke functions in the other. For example, upcalls enable the k-driver to invoke the u-driver.
- *Object tracking*. Synchronize kernel- and user-mode versions of driver data structures. For example, changes made in user mode must be propagated to kernel mode.
- *Recovery*. Restore system operation following a u-driver failure. For example, unload the k-driver when the u-driver crashes.

The runtime provides a mechanism for the k-driver to invoke the u-driver. This involves both signaling the u-driver to execute a particular function as well as providing any necessary data, including both arguments to that function as well as shared data accessed by that function. This need to synchronize shared variables distinguishes Microdrivers from existing communication mechanisms such as LRPC [4]. A similar downcall mechanism allows the u-driver to invoke the k-driver.

The communication mechanism copies data between the k-driver and the u-driver on an upcall. When the call returns, the runtime propagates any changes made to an object in user mode back to the kernel.

The object tracker manages and synchronizes the contents of data shared between a k-driver and a u-driver. Similar to the object tracker in Nooks [36], the object tracker records all objects shared between the k-driver and the u-driver in a table. When the k-driver invokes a function in the u-driver, the runtime locates the user-mode objects corresponding to parameters from the k-driver and updates their contents. On return to the kernel, the runtime looks up the kernel object associated with parameters to propagate changes made by the u-driver back to the kernel. To perform this task, the object tracker must track allocation and deallocation events, to know when to add or delete an object from the tracking table.

The object tracker must also manage kernel objects that are not pure data, such as locks, and I/O resources, such as ports or memory-mapped device registers. For these objects, synchronizing access is not simply copying data; rather, the behavior of the object in kernel mode must be preserved. Our approach is to make the u-driver and k-driver operate on a single object. In the case of I/O ports and memory-mapped device registers, we map the ports and physical addresses to the u-driver. For locks, we do all locking in the kernel and synchronize the user and kernel copies of data structures when locks are acquired or released.

To achieve fault resilience, the runtime must detect and recover when a u-driver fails. Detection occurs at the interface between the k-driver and the u-driver with timeouts and bad parameters checks. In addition, exit of the u-driver process signals a failure. Rather than adding a new recovery mechanism, the Microdrivers architecture is compatible with shadow drivers [35] or the SafeDrive [43] mechanism of logging kernel resource allocations.

4. Implementation of a microdriver in Linux

We implemented the Microdrivers architecture in the Linux 2.6.18.1 kernel to demonstrate its viability and performance.

Our implementation of a microdriver consists of six parts (as illustrated in Figure 1): a k-driver, a kernel runtime, a user runtime, a u-driver, user stubs, and kernel stubs. The k-driver is implemented as a loadable kernel module, much like traditional device drivers in Linux, while the u-driver is implemented as a user-level program. The kernel runtime, which interfaces with the k-driver and communicates with the user runtime, is implemented as a separate loadable kernel module. This module, kr-mod, registers as a device driver that implements the character device interface for communication with the user runtime. The kernel runtime implements functions for user/kernel communication, object tracking and recovery. The user runtime, ur-lib, is implemented as a multithreaded library

that links to the u-driver code. The interface that it exports to the u-driver is similar to the interface that the kernel runtime exports to the k-driver. Section 5 describes user and kernel stub generation. Figure 2 summarizes the mechanisms implemented in each of these components. Overall, the kernel runtime is 4,791 lines of C code and the user runtime is 1,959 lines.

Component	Mechanisms implemented in the component
k-driver	Driver code that executes in kernel mode.
u-driver	Driver code that executes in user mode.
kr-mod	(1) Object tracker (OT); (2) OT query interface for the k-driver; (3) Wrappers for kernel-mode deallocators; and (4) Wrappers for kernel-mode locking/unlocking.
ur-lib	(1) OT query interface for the u-driver; (2) Wrappers for user-mode allocators/deallocators; and (3) Wrappers for user-mode locking/unlocking.
User & kernel stubs	RPC stubs for user/kernel communication; described in Section 5.

Figure 2. Mechanisms implemented in various components in the Linux implementation of a microdriver.

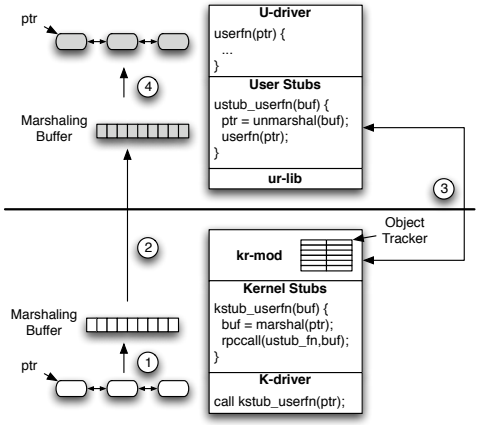

Figure 3. Data movement from the k-driver to the u-driver: (1) a call to the client stub of a u-driver function marshals the data structure; (2) kr-mod invokes the user thread and copies the marshaled data into user space; (3) ur-lib consults the object tracker and unmarshals the data structure; (4) ur-lib invokes the u-driver function on the unmarshaled data structure.

4.1 Communication

We use existing system call mechanisms to communicate between u-drivers and k-drivers because communication performance is of less concern than in previous user-mode driver systems. At startup, the ur-lib library calls an `ioctl` in the kr-mod module and waits for a request. When the k-driver makes an upcall, the `ioctl` returns and passes the request to user level.

A microdriver starts operation by first loading the k-driver and kr-mod modules, and starting ur-lib and the u-driver. The ur-lib library is implemented as a multithreaded program using a thread pool architecture. A master thread in ur-lib registers u-driver functions with kr-mod by invoking the `ioctl` system call. This call blocks in kr-mod until either the k-driver is uninstalled, in which case it returns with an error, or the k-driver invokes a function in the u-driver.

The k-driver invokes a function in the u-driver by calling the client stub. This stub marshals any data structures that may be read by the u-driver as it services the request, and invokes kr-mod to transmit the request to the u-driver. The u-driver pre-allocates a buffer to receive marshaled data; the size of this buffer is set to the upper bound of the number of bytes transmitted between the u-driver and the k-driver. In turn, kr-mod unblocks the master thread waiting in the kernel, copies the marshaled data into the marshaling buffer in the u-driver's address space, and signals ur-lib to invoke the appropriate function in the u-driver.

Upon receiving a request from the k-driver, the master thread of ur-lib dispatches the request to a worker thread, and returns to the kernel to await further requests. It is important to implement ur-lib as a multithreaded program because a request to execute a u-driver function may result in multiple control transfers between the user and the kernel. For example, the u-driver function may invoke a function that is implemented in the kernel, which in turn may call back into a second function in the u-driver.

The ur-lib library invokes a u-driver function by first updating the u-driver's data structures (via the user stub of the u-driver function) using data sent in the marshaling buffer. This buffer contains global variables, formal parameters, and structures accessed indirectly through pointers that are read by the u-driver function and its callees. The ur-lib library unmarshals this buffer using the object tracker to locate and update the user-mode objects corresponding to the kernel objects transmitted in the buffer. At this point, all the u-driver objects that will be affected by the request are synchronized with their k-driver counterparts, and ur-lib invokes the u-driver function with appropriate parameters. On return from this function, ur-lib marshals any data structures modified by the call into a buffer, and passes this back to kr-mod. The data structures modified by the call are determined by statically analyzing driver code, as described in Section 5. Much like ur-lib, kr-mod then unmarshals the data received, updates kernel data structures using the object tracker, and returns control to the k-driver.

The u-driver may also invoke the k-driver as it services a request, either to call a k-driver function, or to execute a function that is implemented in the kernel. A symmetric downcall mechanism in ur-lib and kr-mod handles this. The downcall mechanism invokes a kernel function via the `ioctl` system call. An `ioctl` handler in kr-mod receives this request and calls the appropriate kernel function. Downcalls execute on the thread that initially made the upcall, to ensure that references to user-mode addresses and per-thread structures are correct.

This mechanism is functionally similar to the LPC (lightweight procedure call) mechanism in Windows NT [33]. However, it does not incorporate scheduling support to ensure that the user-mode thread is awoken as soon as the kernel thread blocks on a request.

4.2 Object tracking

The main responsibility of the object tracker is to maintain and update the correspondence between kernel- and user-mode versions of shared data structures. In addition to in-memory objects, it is also responsible for tracking objects with additional semantics, such as I/O resources and locks.

The heart of the object tracker is a bi-directional table that stores the correspondence between kernel- and user-mode pointers. With each table entry, it also stores the size of the object referenced by the pointers. The object tracker exports an interface that supports the following classes of queries.

(**1**) *Translation.* Given a kernel pointer, return the corresponding user pointer (or vice-versa). Return "failed" if a corresponding pointer is not found.

(**2**) *Creation.* Enter a new pair of pointers, denoting a new correspondence, into the object tracker table.

(**3**) *Deletion.* Delete the table entry corresponding to a user-mode pointer or a kernel-mode pointer.

In our implementation, the object tracker resides in kr-mod; ur-lib queries the object tracker using `ioctl` system calls.

When ur-lib unmarshals the contents of the marshaling buffer upon receiving a request from the kernel, it queries the object tracker to translate each kernel pointer that it encounters in the buffer. The object tracker either returns the corresponding user pointer, which ur-lib uses to locate and update the u-driver data structure, or returns "failed." If no valid user pointer is found for a kernel pointer, ur-lib allocates a new memory object of the appropriate type and size (both type and size information are conveyed by the marshaling protocol) and creates a new entry in the object tracker.

Arrays require special treatment; in this case, even if a valid user pointer to the head of an array is found, the size of the array is checked. If its size has changed, the array is reallocated, the old array is freed, the correspondence between the kernel pointer and the old user pointer is deleted from the object tracker, and a new correspondence is added. To efficiently translate pointers into the middle of an array, the object tracker supports *range queries* that allow the head of an array to be found from the address of any of its elements.

4.2.1 Memory allocation and deallocation

Allocations and deallocations create and destroy objects in kernel and user memory. Each such event must update the object tracker appropriately. We handle these events by creating wrappers for allocators and deallocators.

Allocator wrappers forward each request by the u-driver to kr-mod which in turn allocates memory in kernel address space and returns the corresponding kernel pointer. The wrappers also allocate memory in user-mode and create a new entry in the object tracker. This approach ensures that each object in user memory will have a corresponding copy in kernel memory. It also has the advantage that memory allocation requests with special flags (*e.g.,* GFP_ATOMIC or GFP_DMA) are forwarded to the kernel, where memory can be allocated with these flags. Memory allocation requests in the kernel are unmodified, and do not create new entries in the object tracker. If a kernel object is shared with the u-driver, then it is allocated and initialized during unmarshaling, as discussed earlier. Thus, an object in kernel memory will have a copy in user memory only if it is accessed by the u-driver.

Deallocator wrappers in the u-driver intercept deallocation requests and free the object in user memory. This request is also forwarded to kr-mod, which looks up the corresponding kernel pointer, and deallocates the object in kernel memory. Deallocator wrappers in the k-driver are symmetric, but forward the deallocation request to the u-driver on the next upcall (or downcall return) when a copy of the freed object also exists in user memory.

4.2.2 Locking

The Microdrivers runtime must ensure that the u-driver and k-driver can never simultaneously acquire a lock on the object. It must also ensure that when a lock is released, the user and kernel copies of the object guarded by the lock are synchronized. We handle requests to acquire/release a lock using the protocol described below.

Requests by the u-driver to acquire a lock on an object are forwarded to the kr-mod module. In turn, kr-mod acquires a lock on the kernel version of the object and synchronizes the user version of the object with the kernel version, or blocks the locking request until the lock has been released. This approach ensures that the u-driver and k-driver can never simultaneously lock the object. A request by the u-driver to unlock an object that it has locked is similarly forwarded to kr-mod. Before releasing the lock, kr-mod first synchronizes the k-driver's copy of the object with the u-driver's version. This ensures that any changes to the object made by the u-driver are reflected in the k-driver's version. Code for object synchronization is generated using static analysis, as described in Section 5. Unlock requests by the k-driver are handled locally as the u-driver will synchronize its copy of the object when it next requests a lock.

While this approach works for several synchronization primitives offered by the Linux kernel (*e.g.,* semaphores), it fails to work for spinlocks. Spinlocks provide a lightweight synchronization mechanism, and are typically used by the kernel for short critical sections. Threads busy-wait on a spinlock until it is released. The above protocol is problematic for spinlocks for two reasons. First, it can lead to deadlocks. If a u-driver process acquires a spinlock on a kernel object and is descheduled, then kernel threads will busy wait until the u-driver is scheduled for execution again. Second, it offers poor fault isolation and hampers recovery. Several functions that acquire spinlocks disable interrupts. If a u-driver acquires a spinlock on a kernel object, disables interrupts, and crashes (*e.g.,* because of a memory error), then the kernel will be unable to initiate recovery mechanisms.

These reasons motivate a new locking primitive, the *combolock*, for driver objects guarded by spinlocks. We have also designed a protocol to acquire/release combolocks that offers the benefits of spinlocks without the disadvantages described above.

	Acquire *c*	**Release *c***

K-Driver	acquire(*c.splock*);
	if (*c.sem_required* $\neq 0$) {
	  c.sem_required++;
	  release(*c.splock*);
	  acquire(*c.sem*);
	}
		if (*c.sem_required* $\neq 0$) {
		  acquire(*c.splock*);
		  c.sem_required–;
		  release(*c.sem*);
		}
		release(*c.splock*);

U-Driver	acquire(*c.splock*);
	c.sem_required++;
	release(*c.splock*);
	acquire(*c.sem*);
		acquire(*c.splock*);
		c.sem_required–;
		release(*c.sem*);
		release(*c.splock*);

Figure 4. Acquiring and releasing combolocks.

Combolocks are similar to classical reader-writer locks, in which two locks protect the data, one for each class of access. A combolock consists of a semaphore *sem*, a spinlock *splock*, and an integer *sem_required*. The protocol that the k-driver and the u-driver use to acquire/release combolocks is shown in Figure 4. To acquire a combolock *c*, the k-driver first acquires the spinlock *c.splock*. If it determines that a u-driver thread has already aquired the combolock (as indicated by *c.sem_required* being non-zero), it releases *c.splock* and attempts to acquire *c.sem*. In contrast, u-driver requests to acquire a combolock *c* are forwarded to kr-mod, which tries to acquire *c.sem* after setting *c.sem_required* to a non-zero value. Releasing a combolock is symmetric, and is not explained for brevity.

As described earlier, several functions that acquire spinlocks also disable all interrupts. We handle such cases by augmenting the protocol in Figure 4 to disable interrupts just for the driver's device when the u-driver acquires the lock. Bottom half requests and timer interrupts that arrive when the the k-driver holds a combolock are deferred for later execution.

This protocol ensures lightweight locking in the k-driver if an active u-driver thread has not acquired a lock on the corresponding object. It also ensures that the u-driver and k-driver can never simultaneously lock the object guarded by the combolock. Finally, because the u-driver acquires a combolock *c* by acquiring the semaphore *c.sem*, the kernel will not busy wait if the u-driver has acquired the combolock, thus preventing deadlocks and allowing the kernel to initiate recovery mechanisms if the u-driver crashes.

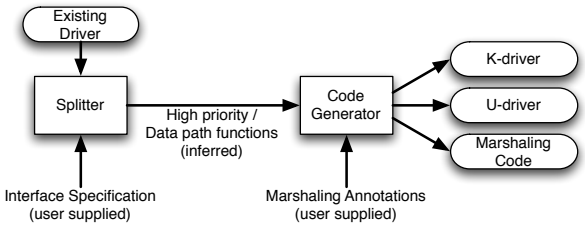

Figure 5. Architecture of DriverSlicer. The splitter infers a partition of the device driver, while the code generator produces the k-driver, the u-driver, and RPC code for data structures that cross the user/kernel boundary.

4.2.3 I/O resources

Functions implemented in the u-driver may access I/O resources such as ports and I/O memory. To prevent each such access from generating a request to the kernel, we allow the u-driver to access these resources directly.

We grant the u-driver permission to access all I/O ports using the `iopl` system call. Thus, the u-driver can access I/O ports directly with `in` and `out` instructions; it is not necessary to request the kernel to perform these operations instead.

To handle u-driver requests to access I/O memory, we map the device's I/O memory region into the u-driver's address space. We also create an entry in the object tracker to map this region of the u-driver's address space to the corresponding I/O memory addresses. We alter the marshaling protocol to avoid accessing objects in I/O memory. Fortunately, pointers to I/O memory objects in device drivers are often already annotated with the `__iomem` attribute. The marshaling code generator, described in the next section, avoids generating code to marshal/unmarshal objects referenced by IOMEM pointers. This ensures that the marshaling protocol will not alter the state of the device.

5. Refactoring device drivers with DriverSlicer

To preserve investment in existing drivers, we designed and implemented DriverSlicer, a tool to automate the creation of microdrivers from traditional device drivers. DriverSlicer is implemented as a source-to-source transformation using the CIL toolkit [29], and consists of about 10,000 lines of Objective Caml code. It interposes upon the build process, and analyzes preprocessed code. DriverSlicer is composed of two parts: a splitter, and a code generator.

5.1 The splitter

The splitter analyzes a device driver and identifies code on the data path and code that is executed at high priority levels. This code must be included in the k-driver to ensure good performance; the remaining code is relegated to the u-driver. Our current implementation infers such a split at the granularity of functions.

Given a device driver, the splitter begins by using programmer-supplied *interface specifications* to identify a set of *critical root functions* for the driver. Critical root functions are driver entry-points that must execute in the kernel and include high priority functions or functions called along the data path. Because these functions typically have a standard prototype, the programmer supplies interface specifications as type signatures. The splitter automatically marks functions that match these type signatures as critical root functions. For example, interrupt handlers in Linux are identified by their return type `irqreturn_t` and the types of their formal parameters. The programmer may also use domain knowledge specific to a driver family to provide interface specifications. For example, the type signature of the `hard_start_xmit` interface for network drivers identifies functions that send packets.

In addition to the above functions, we also include as critical roots driver functions that are called while the kernel holds a spinlock. The reason is because a user mode function cannot be executed within a spinlock critical section. Note, however, that if the kernel is modified to use other synchronization primitives (*e.g.,* semaphores or mutexes) for such critical sections, these functions need no longer be classified as critical roots. In the interest of compatibility with existing kernels, our current implementation works with an unmodified kernel.

The splitter then constructs the static call graph of the driver. It resolves calls via function pointers using a simple pointer analysis that is conservative in the absence of type casts: each function pointer resolves to any function whose address is taken, and has the same type signature as the pointer. It marks all functions that are called transitively from a critical root function as those that must be included in the k-driver. Functions implemented in the kernel are also constrained to execute in the kernel.

The splitter can optionally factor other constraints into the split. For example, functions that are called both by the k-driver and u-driver can be replicated if they can safely execute in user-mode. This replication avoids a request into the kernel each time such a function is invoked in the u-driver. Similarly, all functions that appear inside a critical section can be constrained to execute on the same side of the split.

Thus, the output of the splitter is a call graph, where each node is marked *kern* or *user*, based upon whether the corresponding function must execute in the k-driver or the u-driver.

5.2 The code generator

The code generator uses the results of the splitter and emits the k-driver, the u-driver, and RPC code for control and data transfer across protection domains. It uses static analysis, aided by user-supplied *marshaling annotations*, to identify data structures that cross the user/kernel boundary. This analysis also aims to minimize the amount of data transfered between the u-driver and the k-driver. The code generator also replaces calls to allocator/deallocator functions and lock/unlock functions both in the u-driver and the k-driver with their wrappers.

5.2.1 Generating code for interface functions

Given a call graph with nodes marked *kern* or *user*, the code generator identifies *interface functions* and generates code to transfer control. An interface function is a function marked *user* that can potentially be called by a function marked *kern*, or vice-versa. Non-interface functions are never called by functions implemented on the other side of the split, and thus do not need stubs for control or data transfer.

5.2.2 Generating code for data transfer

The code generator also produces code to marshal/unmarshal data structures accessed by the function. These data structures include all objects reachable via the formal parameters and the return value of the function and global variables accessed by the function.

Marshaling complex data structures for RPC is a well-studied topic, and several commercial tools are available for this purpose [28, 34, 41]. However, these tools are impractical for direct use because it is common practice in the Linux kernel to pass a complex data structure to a function that only accesses a few of its fields. In such cases, if the entire data structure is marshaled, then large amounts of data will cross the user/kernel boundary. This is especially the case with recursive data structures, which are all too common in the kernel. Our solution is to identify and marshal only the fields that are accessed by the function.

This is achieved by the code generator's *field access analysis*, described in Algorithm 1. It is a flow-insensitive, interprocedural analysis that identifies, for each driver function, the fields of complex data types accessed by that function. It traverses the static call graph of the driver bottom-up and computes for each function *f*, a set FieldAccess(f), denoting the set of fields accessed by *f*. Because device drivers typically do not make recursive function calls, Algorithm 1 does not consider cyclic call graphs. However, the algorithm can be extended to handle recursion by processing together all functions in each strongly connected component of the call graph. Algorithm 1 is conservative in the absence of pointer arithmetic to access fields of data structures, *i.e.,* it identifies all fields of data structures that are accessed by a function. To ensure completeness in the presence of pointer arithmetic, it relies on user-supplied marshaling annotations (specifically, the STOREFIELD annotation) described in Section 5.2.3. For each interface function, the code generator then produces marshaling/unmarshaling code using the results of the field access analysis.

```
   Input  : Source code of device driver.
   Output : FieldAccess(f) denoting the set of fields of each complex
            data structure accessed by function f and its callees.
1  G := Static call graph of the driver;
2  L := List of nodes in G, reverse topologically sorted;
3  foreach (f ∈ L) do
4      FieldAccess(f) := φ;
5      foreach (instruction s ∈ f) do
6          switch Instruction-kind(s) do
7              case Read/Write structure.field:
8                  FieldAccess(f) ∪= {Type-of(structure).field};
9              case Call g:
10                 FieldAccess(f) ∪= FieldAccess(g);
11 Return FieldAccess(f);
```

Algorithm 1: Field access analysis.

In our experiments, we found that DriverSlicer's field access analysis is critical for achieving good performance by reducing the amount of data that crosses the user/kernel boundary. For example, consider the 8139cp network driver. In the microdriver produced without field access analysis (*i.e.,* entire data structures are marshaled), 2,931,212 bytes cross the user/kernel boundary during device initialization; in contrast, just 1,729 bytes cross the user/kernel boundary when Algorithm 1 is applied during code generation.

In addition to field access analysis, the code generator also employs an analysis that resolves opaque (`void *`) pointers; it does so by examining how opaque pointers are cast before they are used in the driver. It uses the results of this analysis to resolve and marshal any objects referenced via opaque pointers. Occasionally, the analysis may report that an opaque pointer resolves to more than one type. In such cases, DriverSlicer expects the programmer to use an OPAQUE annotation, described below, to determine the target type of the pointer.

5.2.3 Marshaling annotations

When the code generator encounters a pointer, it may need annotations to clarify the semantics of the pointer to generate code appropriately. For example, it may need to determine whether a pointer refers to an array of objects, or a single instance of the base type. Such semantics cannot always be determined statically, and must be supplied by the programmer in the form of *marshaling annotations*. These annotations are applied to fields of structures, global variables, and formal parameters of interface functions. DriverSlicer implements annotations using the gcc type attributes framework. It currently supports seven kinds of annotations, described below.

(**1**) **NULLTERM**. This annotation is applied to variables that denote null-terminated character buffers. Figure 6 shows an example from the device driver for the AMD pcnet32 network card, and illustrates the use of the NULLTERM annotation on the `name` field of the `pcnet32_private` structure. NULLTERM annotations can also be applied to null-terminated global buffers and formal parameters of functions. A NULLTERM annotation instructs the code generator to compute the length of the buffer and marshal/unmarshal it using standard library routines (*e.g.,* `strlen`, and `strlcpy`).

```
struct pcnet32_private {
     const char * NULLTERM name;
     int rx_ring_size;
     struct pcnet32_rx_head * ARRAY(rx_ring_size) rx_ring;
     spinlock_t COMBOLOCK lock; ...
}
struct net_device {
     void * OPAQUE(struct pcnet32_private) priv;
     struct net_device * SENTINEL(next≠0) next; ...
}
```

Figure 6. Structure definition from the pcnet32 driver, illustrating the NULLTERM, ARRAY, COMBOLOCK, OPAQUE and SENTINEL annotations.

(**2**) **ARRAY**. This annotation identifies pointers that point to more than one instance of the base type. Each ARRAY annotation is parameterized by the length of the array. For example, in Figure 6, the field `rx_ring` of `pcnet32_private` points to an array of `pcnet32_rx_head` structures; the length of this array is stored in the `rx_ring_size` field. In the current implementation of DriverSlicer, array lengths can be specified as constants, other fields of the same C `struct`, or using any variable that is in scope at that program point. The code generator uses the ARRAY annotation to marshal/unmarshal the entire array.

(**3**) **COMBOLOCK**. This annotation denotes spinlocks that must be converted to combolocks (see Figure 6). Spinlocks must be converted to combolocks only if they are used to lock a data structure that is shared by the u-driver and the k-driver. The code generator uses the COMBOLOCK annotation to replace the declaration of the spinlock with a combolock and to replace calls to functions that acquire/release spinlocks with wrappers that acquire/release combolocks instead.

(**4**) **OPAQUE**. While DriverSlicer can resolve many `void *` pointers, it requires help when one may point at more than one type. It is common practice in the Linux kernel to include a `void *` pointer in a kernel data structure that resolves to driver-specific data structures. For example, the `priv` field of the `struct net_device` data structure is a `void *` pointer, as shown in Figure 6. It resolves to a data structure of type `struct pcnet32_private` in the pcnet32 driver. The `priv` field of `struct net_device` is thus annotated using OPAQUE(`struct pcnet32_private`), which informs the code generator that `priv` points to an object of this type. The code generator uses this information when it encounters the `priv` field during marshaling/unmarshaling, and interprets the referenced object as a `struct pcnet32_private`. The OPAQUE annotation can similarly be used to identify integers that are used as pointers.

(**5**) **SENTINEL**. This annotation identifies fields of data structures that can be used to access the structure recursively. When the marshaler encounters such a field, it uses the SENTINEL annotation to guide the traversal of this data structure. For instance, the `next` field of a `net_device` object points to another `net_device` object. In this case, the code generator uses a user-supplied predicate provided with the SENTINEL annotation to guide linked list traversal. In Figure 6, the `next`$\neq$0 predicate allows traversal of the linked list until a null pointer is encountered.

(**6**) **STOREFIELD**. As discussed earlier, Algorithm 1 may miss a field that is accessed only via pointer arithmetic. The STOREFIELD annotation is used to identify fields that are accessed via pointer arithmetic. The code generator always marshals/unmarshals fields with this annotation. For example, in the ne2000 network driver, the `priv` field of the `struct net_device` is accessed via a pointer arithmetic expression. We thus annotate `priv` with STOREFIELD.

(**7**) **CONTAINER**. Given a pointer to a field of a data structure, it is common practice in the Linux kernel to extract a pointer to the parent data structure using the `container_of` utility. For instance, consider the following code snippet, from the universal host controller interface driver for USB:

```
static void start_rh (struct uhci_hcd *
    CONTAINER(struct usb_hcd, hcd_priv) uhci) {
    struct usb_hcd *tmp;
    tmp = container_of(uhci, struct usb_hcd, hcd_priv);
    ...
}
```

In this example, `uhci` is a pointer to an object of type `struct uhci_hcd`, which is contained in an object of type `struct usb_hcd` (and is accessed via the field `hcd_priv`). The function `start_rh` obtains a pointer to the `usb_hcd` object via `container_of`. Because the container data structure is accessed, in this case, we annotate the formal parameter `uhci` using the CONTAINER annotation, as shown above. The code generator uses this annotation to marshal/unmarshal the container data structure as well.

Like other tools that depend on user-supplied annotations, DriverSlicer is sensitive to the correctness and completeness of annotations. An incorrect or a missed annotation can potentially result in an incorrectly marshaled data structure, which can crash the microdriver. While all the annotations discussed above are user-supplied, DriverSlicer employs several heuristics to prompt the programmer on where annotations may be needed. For example, if a variable is used in a pointer arithmetic expression, it is likely that the variable is being used to access an array, and must therefore have an ARRAY annotation. Similarly, opaque pointers are typically type cast before use; such type casts can be used to suggest possible resolutions for OPAQUE annotations. As discussed in Section 6.2.1, the current DriverSlicer prototype does not impose a significant manual overhead in terms of providing annotations. However, improvements to DriverSlicer's inference heuristics could further reduce the number of annotations required.

Some of the annotations described above (*e.g.,* NULLTERM and ARRAY) are closely related to the annotations proposed in SafeDrive [43]. However, the annotations used by DriverSlicer differ from those used by SafeDrive in two ways. First, DriverSlicer uses the annotations to guide the generation of marshaling code, while SafeDrive uses them to synthesize memory safety checks. Second, DriverSlicer only requires annotations on data structures that cross the user/kernel boundary. Specifically, in contrast to SafeDrive, this implies that local variables of functions need not be annotated.

Though we have not attempted to rigorously prove correctness of the transformations employed by DriverSlicer, we note that the code generator performs a sequence of simple transformations. Therefore, we believe that its correctness can either be proved or bugs fixed as it is widely applied. Note however that converting a device driver into a microdriver can often expose bugs in the driver's implementation. For instance, a latent race condition present in the original driver may manifest itself in a microdriver because of increased latencies.

6. Experimental results

In this section, we present the results of running DriverSlicer on four device drivers. We address two concerns:

Driver	Device
8139too	RealTek RTL-8139 Ethernet card
forcedeth	NVIDIA nForce Ethernet card
ens1371	Ensoniq sound card
uhci-hcd	Universal host controller interface

Figure 7. The Linux drivers evaluated.

(**1**) *Benefits.* What are the benefits of the Microdrivers architecture and the DriverSlicer tool, in terms of programmability, kernel-mode code reduction, and increased fault tolerance?

(**2**) *Costs.* What are the costs of Microdrivers, in terms of human effort to split drivers and performance?

To evaluate these questions and verify that DriverSlicer is applicable to different kinds of drivers, we considered drivers drawn from three major classes with widely differing interfaces to the kernel (Figure 7). We believe that these drivers are largely representative of the other drivers in their corresponding classes. The 8139too, ens1371, and uhci-hcd drivers were tested using a Pentium D 3.0GHz with 1024MB RAM running CentOS 4.2, while forcedeth was tested on an AMD Opteron 2.2GHz with 1024MB RAM running CentOS 4.4. All experiments were conducted using an unmodified Linux 2.6.18.1 kernel. In addition, we also created microdrivers for the pcnet32, ne2000 and 8139cp network drivers and ran them in a virtual machine testbed. The results were similar to the other drivers; we do not report them here.

6.1 Benefits

The Microdrivers architecture offers three benefits: (**1**) the ability to develop driver code at user level, with sophisticated programming tools; (**2**) reduction in the quantity of hand-written code in the kernel, which may have reliability or security bugs; and (**3**) improved reliability by isolating the kernel from bugs in user-level code.

6.1.1 User-level programming

Moving code from the kernel to a standard user-level process allows the use of user-level debugging and instrumentation aids such as GDB, Valgrind [30], and others. These tools are often more robust than their kernel-mode counterparts because of their larger user base. Consequently, when using the microdriver framework, developer productivity increases because these tools simplify debugging and performance analysis when the code under scrutiny is implemented in the u-driver.

During development, we used Valgrind and user-level GDB to eliminate bugs in the code generated by DriverSlicer. For example, one of the bugs that we discovered using Valgrind was a memory corruption error that manifested because insufficient memory was allocated for the marshaling buffer. This error, which resulted in a buffer overflow, did not immediately result in a crash, and would have been difficult to detect when the crash happened. Valgrind's memcheck tool immediately flagged this as a memory corruption error, which we then fixed.

Since DriverSlicer allows some flexibility in how the driver is split, an additional application of it may lie in debugging during driver development. For example, large parts of the driver can be moved to the u-driver using DriverSlicer, and debugged using user-level tools. Once the code is stable, performance-critical functionality can be moved back into the kernel for deployment.

6.1.2 Kernel-mode code reduction

One of the primary goals of the Microdrivers architecture is to reduce the amount of kernel-mode driver code to improve reliability and simplify debugging. To identify code that can potentially be removed from the kernel, we analyzed 297 device drivers, comprised of network, SCSI and sound drivers using DriverSlicer. We counted both the number of performance critical driver functions as well as

Driver family	Drivers analyzed	High priority and data path functions	Lines of code in these functions
Network	89	33.7%	37.2%
SCSI	33	32.9%	35.5%
Sound	175	26.3%	26.7%

Figure 8. Classification of functions in different families of Linux device drivers.

	K-Driver		U-Driver	
Driver	SLOC	# Functions	SLOC	# Functions
8139too	559 (34.6%)	12 (23.5%)	1056 (65.4%)	39 (76.5%)
forcedeth	1200 (34.7%)	28 (30.4%)	2260 (65.3%)	64 (69.6%)
ens1371	805 (53.8%)	25 (37.5%)	689 (46.1%)	40 (61.5%)
uhci-hcd	2028 (80.6%)	57 (80.3%)	489 (19.4%)	14 (19.7%)

Figure 9. Measurements comparing the amount of code in the k-driver and u-driver.

the number of lines of code in these functions. We only counted lines of code present within functions; specifically, we omitted global variables, preprocessor macros, and declarations that were not part of any function. As Figure 8 shows, DriverSlicer identified that fewer than 34% of the functions, comprising nearly 37% of the code of a driver are performance critical. While Figure 8 shows the potential for kernel-mode code reduction, Figure 9 presents the results of splitting the four drivers that we tested. As these results show, we were able to remove several non-critical functions from the kernel for each of the four drivers tested.

We also studied the revision history of these four drivers, and identified which portion of the corresponding microdriver would have been affected by each change. We found that the fraction of changes to the u-driver of each microdriver is roughly proportional to the relative size of the u-driver. Although we could not further classify changes as bug fixes or performance enhancements, this result indicates that splitting drivers can improve fault isolation and driver programmability.

In addition to the k-driver, kernel stubs execute in the kernel as well. DriverSlicer's code generator currently emits several thousand lines of kernel stub code: 14,700 for 8139too, 37,900 for forcedeth, 6,100 for ens1371 and 12,000 for uhci-hcd. While these constitute a large number of lines, they are highly-stylized and are automatically generated by DriverSlicer. They do not therefore contribute toward the number of hand-written lines of code in the kernel. Bugs in the stubs can be corrected by fixing the DriverSlicer's code generator rather than modifying each driver individually; this one time cost of verifying/testing the code generator is amortized over the drivers that it is applied to. In addition, existing techniques, such as table-driven marshaling, can greatly reduce the amount of code required [28].

6.1.3 Reliability

Our Microdrivers implementation has only rudimentary code for detecting and recovering from faults in a u-driver. We therefore restricted ourselves to injecting faults that would not require a full-fledged failure detector or recovery module. In particular, we tested the ability of the system to tolerate failures that would normally cause a kernel panic by injecting null pointer dereferences in the user-mode portions of each driver. Such bugs in a device driver kernel typically necessitate a reboot of the system to restore proper operation. In all cases, when the buggy code executed, the u-driver crashed, as expected, but when control returned to the k-driver, the error was detected. In our current implementation, control returns to the kernel when the u-driver crashes, but the data sent by the u-driver is not unmarshaled. Thus, the system behaves as if the function call that led to the crash never happened.

We injected the null pointer error in the sound driver in a function related to sound playback. When an application attempted to play a sound, the application hung when the u-driver crashed. When we terminated the application, control returned to the command prompt, and there was no other indication of a problem, other than the lack of sound (because the device was no longer functional). In the two network drivers, we introduced the null pointer dereferences in device initialization functions. In both cases, after the insmod command returned, the device was in an unusable state, but the system otherwise operated normally. We introduced a similar error in the initialization code for uhci-hcd; as with the network drivers, when the u-driver crashed, the USB device was unusable, but the system was otherwise functional.

This study demonstrates that our current implementation of Microdrivers can contain faults that crash the u-driver without affecting the rest of the system. However, because our implementation does not yet include a failure detection or a recovery subsystem, a buggy u-driver can still crash the entire system. For example, an erroneous deallocation request or a data structure corrupted by a u-driver can potentially crash the k-driver. Microdrivers augmented with a failure detection and recovery mechanism, such as shadow drivers [35] or the SafeDrive recovery mechanism [43], can further improve system reliability and availability.

Driver	Kernel header	Driver-specific
8139too	34	8
forcedeth	34	12
ens1371	7	7
uhci-hcd	27	146

Figure 10. Number of annotations.

6.2 Costs

The costs of the Microdrivers architecture are the burden on programmers to convert existing drivers to microdrivers, in the form of annotating driver and kernel code, and the performance cost of switching protection domains while drivers execute.

6.2.1 Manual overhead

To evaluate the manual overhead involved in creating a microdriver, we counted the number of manual annotations that we had to supply to DriverSlicer. Figure 10 summarizes these results; for each driver we counted both the number of annotations to kernel headers (which must be supplied just once) as well as driver-specific annotations (which must be supplied on a per-driver basis). The same set of kernel header annotations sufficed for both network drivers. Overall, only 1-6% of driver code required annotations.

6.2.2 Performance

To measure the performance impact of splitting device drivers, we measured common-case performance overheads using data-intensive benchmarks. The results are summarized in Figure 11.

We measured the performance (both throughput and CPU utilization) of the two network drivers using the netperf utility [10]. We installed each network driver on a test machine, and measured TCP throughput between the test machine and a client. All netperf tests used the default TCP receive and send buffer sizes of 87,380 bytes and 16,384 bytes, respectively. Netperf was run using a single process; all results were averaged over 3 runs. As expected, these results demonstrate that common-case performance is minimally impacted in a microdriver (the minor speedups that we observed with the forcedeth driver are within the margin of experimental error). For the sound driver, we measured CPU utilization (using the vmstat utility) while playing a 256-Kbps MP3 file. The CPU utilization with both the microdriver and the original driver was effectively zero. We measured the performance of the uhci-hcd driver by using it to copy the Linux-2.6.18.1 source tarball (230MB) from a hard drive to a USB flash drive. As with other drivers, overheads

		Original driver		Microdriver	
Driver	**Workload**	CPU (%)	Throughput	CPU (%)	Throughput
8139too	TCP-send	8.46%	94.13Mbps	8.43% (-0.39%)	94.10Mbps (-0.04%)
8139too	TCP-receive	10.69%	94.07Mbps	10.68% (-0.09%)	94.07Mbps (-0.00%)
forcedeth	TCP-send	20.68%	940.19Mbps	21.64% (+4.66%)	940.06Mbps (-0.01%)
forcedeth	TCP-receive	65.24%	939.27Mbps	66.44% (+1.84%)	939.75Mbps (+0.05%)
uhci-hcd	Copy	0.576%	914.64KBps	0.592% (+2.78%)	913.19KBps (-0.16%)

Figure 11. Measurements comparing the performance of network and uhci-hcd microdrivers with unmodified device drivers.

	8139too	**forcedeth**	**ens1371**	**uhci-hcd**
KBytes sent/received	127	50	7.6	88
Requests to user	42	11	198	61
Requests to kernel	45	50	402	238

Figure 12. Measurements comparing the amount of data copied between the k-driver and u-driver.

for throughput and CPU utilization were near zero for the uhci-hcd microdriver.

We also examined the number of user/kernel transitions and the amount of data transferred between the two domains to form a more accurate understanding of how frequently such transitions are necessary. In each case, we counted the number of transitions to install the driver and run the data-intensive workload. In all drivers, the user/kernel transitions reported in Figure 12 happened during startup. In particular, we observed *no user/kernel transitions* along the data-intensive path. For the network drivers, this corresponds to sending/receiving packets, while for the sound driver, it refers to playing a sound file. Driver startup, which resulted in a number of user/kernel transitions, took almost thrice as long.

	8139too	**forcedeth**	**ens1371**	**uhci-hcd**
Unmodified driver	9.9	48	29	29
k-driver+kern stubs	184	557	87	142
kr-mod	54	51	54	54
User components	136	48	28	48

Figure 13. Measurements comparing the memory usage (in Kbytes) of microdrivers with unmodified drivers.

Figure 13 compares the memory usage of unmodified device drivers and the corresponding microdrivers. We measured the memory utilization of the unmodified driver, k-driver, kernel stubs, and kr-mod using the lsmod utility. For each microdriver, we also measured the steady-state resident set size of user space components (the u-driver, user stubs and ur-lib). As discussed earlier, DriverSlicer emits several thousand lines of stub code, which contribute toward the memory utilization of microdrivers.

7. Related work

Prior projects on improving system reliability by isolating device drivers fall under three categories: hardware-based isolation, language-based isolation, and user-mode driver frameworks. We compare the Microdrivers architecture against prior work along four axes, as shown in Figure 14: user-level programming, fault isolation, common-case performance, and compatibility.

Hardware-based isolation. Several projects use hardware-based mechanisms to isolate device drivers. For example, Nooks [36] adds a reliability subsystem to commodity operating systems that allows each driver to run in its own protection domain, while virtual machine-based techniques (*e.g.,* [13, 15, 21]) run device drivers within their own virtual machines. Mondrix [40] is a hybrid hardware/software approach that also offers memory protection for kernel extensions, including device drivers. Such schemes offer good fault isolation, report low performance overheads and are compatible with commodity operating systems. However, they do not offer

Approach	**ULP**	**Isol.**	**Perf.**	**Comp.**
Microdrivers	√	√	√	√
Nooks [36]	×	√	√	√
VM-based [13, 15, 21]	×	√	√	√
Mondrix [40]	×	√	√	√
SafeDrive [43]	×	√	√	√
User f/w [8, 11, 20, 25]	√	√	√	×
User compat. f/w [2, 38]	√	√	×	√
Microkernels [22, 41]	√	√	√	×

ULP = User-level programmability; **Isol.** = Fault isolation; **Perf.** = Good common-case performance; **Comp.** = Compatibility.

Figure 14. Comparison with related work.

the benefits of user-level programming because drivers execute in the kernel. In contrast, microdrivers offer many of the benefits of hardware-based isolation for a large fraction of driver code, while allowing driver writers to avail of user-level programming tools.

Language-based isolation. SafeDrive [43] uses programmer-supplied annotations and a type-inference engine to insert memory safety checks. It also uses a recovery mechanism that is triggered when a safety check fails. While SafeDrive offers low-performance overhead and compatibility, device drivers protected with SafeDrive still execute in the kernel. Further, SafeDrive currently only protects against memory safety violations, such as null pointer dereferences, and does not protect against synchronization errors. Microdrivers, in contrast, can offer protection against synchronization errors (*e.g.,* deadlocks) in the u-driver.

User-mode driver frameworks. There have been several attempts to remove device driver code from the kernel. Examples include Microkernels [22, 41], and several user-mode driver frameworks [8, 11, 20, 25, 37]. While these frameworks allow user-level programming and good fault isolation, they all suffer from one of two problems. They either offer poor performance [2, 38] because they transmit large amounts of data frequently across the user/kernel boundary, or they are incompatible with commodity operating systems, often requiring complete rewrites of drivers and modifications to the kernel [8, 20, 25, 37]. Microdrivers offer poorer fault isolation than these frameworks because they do not isolate faults in the k-driver. However, because they are compatible with commodity operating systems and offer good performance, they offer a path to execute existing drivers in user mode.

In addition to the above projects on fault isolation, there has also been work on user-level network interfaces [39], which take exactly the opposite approach of Microdrivers and leave control code in the kernel while moving the data path to user-level. There have been other approaches at simplifying driver programming beyond moving code to user level, such as domain-specific languages for device access [9, 23, 24]. While useful, these approaches do not allow the use of user-mode programming tools nor do they reduce the kernel footprint of drivers.

Program partitioning. Program partitioning techniques, much like those in DriverSlicer, have also been used to improve application security [5, 42], create secure web applications [6], and to improve performance of distributed components and data-intensive applications [17, 31]. However, these techniques have not been applied to

device driver code. Several key components of Microdrivers, such as object tracking and synchronization, are unique to device drivers and have not been addressed in prior work.

Static analysis tools. Bug-finding tools, such as SLAM [3] and MC [12], have been developed to find bugs in device drivers. While these tools help improve system reliability, they do not make it easier to write driver code. In contrast, Microdrivers simplify driver development and improve programmability, *e.g.,* by allowing the use of memory leak detectors during development. Static analysis tools complement Microdrivers and would be helpful in improving the quality of the k-driver and the u-driver.

8. Conclusions and future work

Driver frameworks today offer an all-or-nothing choice between user mode and the kernel. Past microkernel developers have found the temptation to move user-mode code into the kernel to improve performance too strong [19, 14, 32], indicating the need for a hybrid model that allows *some* code in the kernel. The Microdrivers architecture is one such model that offers the benefits of user-level code development, together with good common-case performance and fault isolation. Further, because it is compatible with commodity operating systems, existing drivers can be ported to microdrivers.

Several enhancements to the current implementation of DriverSlicer can move even more code out of the kernel. For example, DriverSlicer could use profile information to identify performance-critical program paths (rather than functions) and split the driver at a finer level of granularity. Similarly, with minor modifications to the kernel, DriverSlicer can potentially move more functions to user mode. If critical sections in the kernel from which driver functions are invoked can be modified to use locking primitives other than spinlocks, then DriverSlicer will classify fewer functions as critical root functions, thus moving more code out of the kernel. We plan to explore these enhancements in future work.

While this paper has focused on porting existing drivers, the Microdrivers architecture can be applied to new device drivers as well. We expect that developing a microdriver as a u-driver and a k-driver will be akin to programming a distributed application. In this approach, a driver developer must manually identify performance-critical functions and program them in the k-driver. The developer must also identify objects shared by the u-driver and the k-driver and ensure that they are accessed atomically. An alternative approach that avoids these challenges is to program the driver monolithically with marshaling annotations, and use DriverSlicer during the development process to identify performance-critical functions and partition the code into a microdriver.

Acknowledgments. This work is supported in part by the NSF with grant CCF-0621487.

References

[1] Apple Inc. Introduction to I/O kit fundamentals, 2006.

[2] Francois Armand. Give a process to your drivers! In *EurOpen Autumn 1991*, September 1991.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[4] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM TOCS*, 8(1), 1990.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security*, 2004.

[6] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *SOSP*, 2007.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[8] P. Chubb. Get more device drivers out of the kernel! In *Ottawa Linux Symp.*, 2004.

[9] C. L. Conway and S. A. Edwards. NDL: A domain-specific language for device drivers. In *LCTES*, 2004.

[10] Information Networks Division. Netperf: A network performance benchmark. http://www.netperf.org.

[11] J. Elson. FUSD: A Linux framework for user-space devices, 2004. User manual for FUSD 1.0.

[12] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[13] Ú. Erlingsson, T. Roeder, and T. Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-05-82, Microsoft Research, June 2005.

[14] M. Rozier *et al.*. Overview of the Chorus distributed operating system. In *USENIX Symp. on Microkernels & other kernel architectures*, 1992.

[15] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[16] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. In *HotOS*, 2007.

[17] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *OSDI*, 1999.

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

[19] J. Lepreau, M. Hibler, B. Ford, J. Law, and D. B. Orr. In-kernel servers on Mach 3.0: Implementation and performance. In *USENIX Mach III Symp.*, 1993.

[20] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 20(5), 2005.

[21] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, 2004.

[22] J. Liedtke. On μ-kernel construction. In *SOSP*, 1995.

[23] A. Manolitzas. A specific domain language for network interface cards, 2001.

[24] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *OSDI*, 2000.

[25] Microsoft. Architecture of the user-mode driver framework, May 2006. Version 0.7.

[26] Microsoft Corp. Wdm: Introduction to windows driver model. http://www.microsoft.com/whdc/archive/wdm.mspx.

[27] Microsoft Corporation. Windows XP embedded with service pack 1 reliability. http://msdn2.microsoft.com/en-us/library/ms838661.aspx, January 2003.

[28] Microsoft Inc. Microsoft interface definition language.

[29] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Constr.*, 2002.

[30] N. Nethercode and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[31] A. Purohit, C. P. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in user-land, run in kernel-mode. In *HotOS*, 2003.

[32] M. E. Russionvich and D. A. Solomon. *Inside Windows 2000*. Microsoft Press, Redmond, Washington, 2000.

[33] David A. Solomon. *Inside Windows NT*. Microsoft Press, 1998.

[34] Sun Microsystems. Java idl. http://java.sun.com/products/jdk/idl.

[35] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, 2004.

[36] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM TOCS*, 23(1), February 2005.

[37] L. Torvalds. UIO: Linux patch for user-mode I/O, July 2007.

[38] K. T. Van Maren. The Fluke device driver framework. Master's thesis, Dept. of Computer Science, Univ. of Utah, December 1999.

[39] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *SOSP*, 1995.

[40] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *SOSP*, 2005.

[41] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian. Mach: A new kernel foundation for UNIX development. In *Summer USENIX Conference*, 1986.

[42] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM TOCS*, 20(3), August 2002.

[43] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.