Published in Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS XI), San Diego, California, May 2007

Microdrivers: A New Architecture for Device Drivers

Vinod Ganapathy, Arini Balakrishnan, Michael M. Swift and Somesh Jha
Computer Sciences Department, University of Wisconsin-Madison

Abstract

Commodity operating systems achieve good performance by running device drivers in-kernel. Unfortunately, this architecture offers poor fault isolation. This paper introduces microdrivers, which reduce the amount of driver code running in the kernel by splitting driver functionality between a small kernel-mode component and a larger user-mode component. This paper presents the microdriver architecture and techniques to refactor existing device drivers into microdrivers, achieving most of the benefits of user-mode drivers with the performance of kernel-mode drivers. Experiments on a network driver show that 75% of its code can be removed from the kernel without affecting common-case performance.

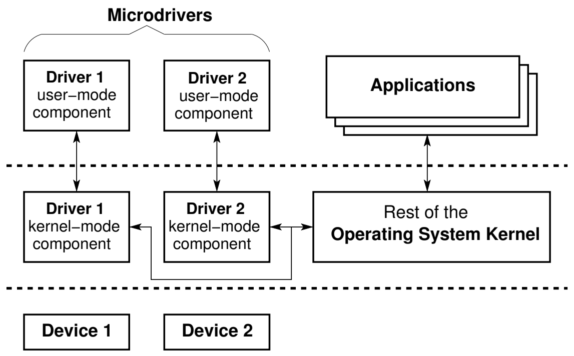
Microdrivers
Driver 1 user–mode component
Driver 2 user–mode component
Applications
Driver 1 kernel–mode component
Driver 2 kernel–mode component
Rest of the Operating System Kernel
Device 1
Device 2

Figure 1: Microdrivers split device driver functionality between a kernel-mode component and a user-mode component.

1 Introduction

Bugs in device drivers are a major source of reliability problems in commodity operating systems. For instance, a recent Microsoft report indicates that as many as 85% of failures in Windows XP stem from buggy device drivers [19].

The root of the problem is the architecture of commodity operating systems. They are designed as *monolithic kernels* with all device drivers (and other kernel extensions), residing in the same address space as the kernel. This architecture results in good performance because invoking driver functionality is as easy and efficient as a function call. Unfortunately, this also results in bloated operating systems and poor fault isolation. For example, kernel extensions constitute over 70% of Linux kernel code [6] (a large fraction of these are device drivers), while over $35,000$ drivers exist on Windows XP desktops [18]. A single bug exercised in any one of these extensions suffices to crash the entire operating system.

Several architectures have been proposed to isolate faults in device drivers [1, 9, 10, 16, 17, 22, 25]. For example, *microkernels* run device drivers as user-mode processes. A bug exercised in a device driver only results in the failure of the user-mode process running that driver. This approach, however, has two key problems. First, this architecture is not compatible with commodity operating systems, which are designed as monolithic kernels. Providing support for user-mode device drivers in commodity operating systems thus requires kernel modifications and rewriting device drivers [7, 14]. Second, switching between the kernel and a user-mode device driver involves the costly overhead of changing protection domains. For devices such as high-throughput network cards, this can result in significant latencies and performance penalties [22, 23]. A common escape hatch employed in such cases is to implement drivers within the kernel, which defeats the benefit afforded by microkernels.

This paper presents a new architecture for device drivers called *microdrivers*. Microdrivers seek the middle ground between monolithic kernels and microkernels, and improve reliability while maximizing performance. In a microdriver, the functionality of a device driver is *split* between a kernel-mode component and a user-mode component (Figure 1). The kernel-mode component contains critical and frequently used functionality, such as interrupt handling and performance-critical operations (*e.g.,* sending and receiving network packets and processing SCSI commands), while the user-mode component contains non-critical and infrequently used functionality (*e.g.,* startup/shutdown and error-handling). The user-mode component is implemented as a standalone process that is called from the kernel-mode component. Together, they provide the functionality of a traditional device driver.

Microdrivers are motivated by a simple mantra: *reduce the amount of code running in the kernel to improve* *its reliability*. As discussed in Section 2, more than 70% of device driver code contains non-critical functionality. By relegating this code to a user-mode process, a microdriver reduces the amount of code running in the kernel and improves the reliability of the system as a whole. In addition, because the kernel-mode component of a microdriver is much smaller than the entire driver, it is amenable to verification and code audits.

Perhaps the most important aspect of microdrivers is compatibility with commodity operating system architectures—device drivers written for monolithic kernels can be *refactored nearly automatically* into microdrivers. This provides a path to execute device drivers written for commodity operating systems in user-mode without sacrificing performance. Prior efforts at user-mode device drivers have either required rewriting the driver completely [7, 14] or impose significant performance penalty [23]. We show that program analysis techniques can automatically identify critical functions in a device driver, following which a semantics-preserving transformation can split the driver into a user-mode and a kernel-mode component. We discuss the design and implementation of such a refactoring tool in Section 4. We used this tool to refactor the e1000 device driver for the Intel PRO/1000 gigabit network adapter into a microdriver. The kernel-mode component of this microdriver contains just 25% of the code of the entire microdriver. Our preliminary experience with this microdriver indicates that overheads for common-case performance and CPU utilization are negligible.

2 The case for microdrivers

Because device drivers communicate with I/O devices, their performance is critical to ensure fast I/O. Conventional wisdom holds that performance-critical functionality must be implemented in the kernel. Even undergraduate texts preach that I/O algorithms must be implemented in the kernel for good performance [21, page 427]. Unfortunately, a popular interpretation of this tenet is that device drivers must reside in the kernel. This has lead to the monolithic and unreliable operating systems that we see today.

Surely, the *entire driver* does not reside on the performance-critical path? To answer this question, we conducted a study of 455 device drivers, comprising network, SCSI and sound drivers from the Linux 2.6.18 kernel, and identified performance-critical functions in each of them. To do so, we extracted the static call-graph of each driver—this graph has an edge $f \rightarrow g$ if function f can potentially call function g. We resolved calls via function pointers using a simple pointer analysis that is conservative in the absence of type-casts—each function pointer can resolve to any function whose address is taken, and whose type signature matches that of the function pointer.

Driver family	Drivers analyzed	Critical funcs
Network	134	27.8%
SCSI	49	26.1%
Sound	272	7.8%

Figure 2: Classification of functions in different families of Linux device drivers.

We then identified a set of *critical root functions* that are driver entrypoints that must execute in the kernel for high performance. Critical root functions are those that handle interrupts or execute at other high-priority levels (*e.g.,* tasklets, bottom-halves), and functions that supply data to or receive data from a device. We define performance-critical functions to be critical root functions plus the functions that they transitively call. Given a template of the entrypoints, critical root functions can be identified automatically for each family of drivers: *e.g.,* functions that transmit network packets are critical for network drivers, while functions that process SCSI commands are critical for SCSI device drivers. We wrote a tool to automatically identify critical root functions (based upon their type signatures) and the functions that they transitively call.

Figure 2 shows the results of our study. We found that fewer than 30% of the functions in a driver are performance critical. The remaining functions are called only occasionally, *e.g.,* during device startup/shutdown, to configure device parameters, and to obtain diagnostic information. Consider, for example, the e1000 driver for the Intel PRO/1000 gigabit network adapter, one of the drivers considered in our study. Critical root functions for this driver include the interrupt handler, the function to transmit network packets, and callback functions registered with the kernel to poll the device. This driver contains 274 functions containing approximately 15,100 lines of source code. Of these, just 25 functions containing approximately 1,550 lines of source code were classified as critical. It suffices to execute just these functions in the kernel for good performance. Relegating the remaining functions, which handle startup/shutdown and get/set device parameters, to a user-mode process will greatly reduce the amount of code running in kernel space without adversely impacting common-case performance. Note that our estimate of critical device driver code is conservative, because we only identify critical functions. It is likely that a finer-grained approach will show that even less code is on the critical path.

Three factors lead us to believe that implementing non-critical functionality as a user-mode process will also improve system reliability and availability as a whole.

First, fault isolation improves. Any bugs that crash the user-mode process of a microdriver will potentially render the corresponding device unusable but will not affect the rest of the operating system. The device driver can then be restarted in isolation to restore operation of the device. Note that because the kernel-mode component of a microdriver implements critical device functionality, such as interrupt processing, it is possible to keep the device operational even if the user-mode process crashes. For example, the kernel-mode component can implement error-checking code that detects that the user-mode process has crashed, and wait until the process restarts. However, as it does so, it can still serve other requests to/from the device.

Second, because the kernel-mode component of the microdriver implements critical and heavily-used functionality, this code is likely more heavily tested than the user-mode component. Further, because the kernel-mode component is a relatively small entity, it can either be verified, subject to thorough code audits, or be protected with mechanisms such as SFI [24].

Third, because the kernel-mode component and the user-mode component of a microdriver communicate via a narrow interface (as desribed in Section 3), data passed between the kernel- and user-mode components can be sanity-checked for errors. For example, a bug in the user-mode component may introduce a dangling pointer in a data structure that it then passes to the kernel. However, the corrupted data structure can be detected using error-checking code implemented at the interface, thus potentially preventing corruption of kernel data structures.

Indeed, the tenet that reduced code in the kernel means improved reliability has also been recognized by many others [4, 7, 9, 13], and is an important motivation for microkernels. This has resulted in several services, that were previously implemented in the kernel, being implemented in user-mode (*e.g.,* kernel module loaders, AFS). There have also been several recent efforts to redesign device drivers (in particular, graphics drivers) with the goal of reducing the amount of code running in the kernel [4, 13].

Finally, microdrivers also allow vendors to take advantage of user-level tools such as profilers and debuggers during the driver development process. Comparable tools for developing kernel code are fewer in number and not as advanced because kernel programming represents a smaller market and a more challenging target.

Of course, microdrivers are not a panacea for device driver reliability problems. A bug in the kernel-mode component of a microdriver could still crash the operating system. It is also possible that by splitting functionality between a user-mode and kernel-mode component, microdrivers can expose otherwise latent bugs. For example, a latent race condition in a device driver could potentially be exposed in its microdriver implementation.

3 Architecture of a microdriver

A microdriver consists of a kernel-mode component that implements critical functionality and a user-mode process that implements non-critical functionality. Device driver functionality is split between the kernel-mode and user-mode components at function boundaries. The two components communicate using an LRPC-like mechanism [3]. In the kernel-mode component, direct calls to functions implemented in the user-mode component are replaced with upcalls through stubs. Stubs marshal data structures accessed by the called function and unmarshal them when the call returns. A symmetric downcall mechanism enables the user-mode component to invoke kernel functions. To handle multiple requests from the kernel-mode component, the user-mode process is multithreaded.

An object tracker, similar to the one used by Nooks [22], synchronizes copies of a data structure in the kernel's address space and the user-mode process' address space. It has three main functions.

First, the object tracker is invoked during marshaling/unmarshaling to translate pointers between address spaces. This ensures that updates to a data structure in one address space are reflected on its copy in the other address space. Doing so is challenging for complex data structures such as arrays, whose elements are accessed as offsets from the start of the data structure. The object tracker must explicitly store the range of such data structures and ensure that accesses via offsets are translated correctly between address spaces.

Second, the object tracker ensures that data structures allocated/deallocated in one address space are also allocated/deallocated in the other. Allocations are dealt with during pointer translation—a new data structure is allocated and initialized in an address space if no corresponding copy is found in that address space. Dealing with deallocations is more challenging. Deallocator functions must update the object tracker's database by removing the entry for the data structure being deallocated.

Third, the object tracker manages synchronization of shared data structures. Two copies of a shared data structure can exist in a microdriver, one in each address space, only one of which must be modified at any time. To support concurrent accesses to such data structures, the user-mode process must synchronize with the kernel to acquire a lock on a shared data structure. The object tracker must ensure that any updates to a shared data structures in one address space are reflected to its copy in the other address space.

In addition to the basic functions described above, the object tracker can optionally include error-checking code to check for a variety of common data structure corruptions, such as dangling pointers and potential null-pointer dereferences.

Several enhancements are possible to the basic architecture of a microdriver. Functions that are repeatedly called from both the kernel-mode component and user-mode component can potentially be replicated in both components, thus avoiding the overhead of an address-space change each time the function is accessed. Similarly, a frequently-accessed data structure can be allocated in a shared memory region that is accessible both to the kernel and the user-mode process. Finally, to ensure fast operation of the user-mode process, the operating system can pin the process' pages to memory.

4 Refactoring device drivers to microdrivers

Microdrivers present the same interface to the kernel as traditional device drivers, and are thus compatible with commodity operating systems. Moreover, code to implement upcalls, downcalls, marshaling and unmarshaling follows a standard template and can be automatically generated. This section presents the design of a tool that statically refactors traditional device drivers into microdrivers (see Figure 3). Such a tool preserves the investment in existing device drivers and provides a migration path to create microdrivers.

The refactoring tool has two functions. First, it must analyze the device driver and determine which functions are critical. This is achieved by the *splitter*. Second, it must move the remaining (non-critical) functions to a user-mode component, and generate code for communication between the kernel-mode and user-mode components. This is achieved by the *code generator*.

The splitter. The splitter analyzes the device driver and determines how functions implemented in the driver must be split between kernel-mode and user-mode. It builds a static call-graph of the driver (including edges for indirect calls), identifies critical root functions, and classifies functions transitively called by them as critical, as described in the study in Section 2. Critical root functions need to be identified just once for each family of device drivers.

While this simple propagation-based approach to identify critical functions has worked well for drivers that we have considered so far, the splitter can employ more sophisticated algorithms that use dynamically gathered profile information. For example, critical functions can be inferred by solving an optimization problem on the static call-graph modeled as a *flow network* [2] with weights on edges and nodes. Edge weights denote call frequencies (obtained by profiling) and node weights are proportional to the number of lines of code in the function denoted by the node. The goal is to find a cut in the graph under the constraint that all nodes representing critical root functions must appear on one side of the cut (the critical side). Additional constraints can also be imposed, *e.g.,* that a critical section must not be split between the kernel-mode and user-mode components. Further, the cut should be optimal: it should mimimize both the weight of edges crossing the cut and the weight of nodes on the critical side of the cut. All nodes on the critical side of the cut are marked critical, and the remaining nodes are non-critical. Intuitively, such a cut minimizes the number of switches between protection domains and also the amount of code running in the kernel.

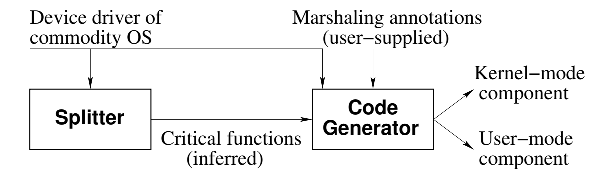
Device driver of commodity OS
Marshaling annotations (user–supplied)
Kernel–mode component
Splitter
Code Generator
Critical functions (inferred)
User–mode component

Figure 3: Design of a tool to refactor traditional device drivers into microdrivers.

The code generator. The code generator uses the critical functions identified and emits code for the kernel-mode and user-mode components. It also generates all the code to handle upcalls and downcalls, including stubs and code to marshal/unmarshal data structures. The object tracker and threadpool implementation (for the multithreaded user-mode component) are common to all microdrivers and need to be written just once.

The code generator needs *marshaling annotations* to guide the generation of marshaling/unmarshaling code. These annotations are used to specify the length of dynamically allocated arrays, linked lists and other complex data structures. The code generator employs a conservative static analysis algorithm to identify variables that represent such data structures and prompts the user to provide these annotations. This potentially reduces the traditional burden associated with annotation, because the user does not have to provide annotations beforehand, but only as guided by the code generator, and only for data structures that cross the user/kernel boundary. For example, for the e1000 device driver, the code generator automatically infers that variables of type `struct e1000_rx_ring` and `struct e1000_tx_ring` (among others) are arrays. These denote ring buffers that are allocated by the e1000 driver at startup. It requests marshaling annotations for each function call that crosses address-spaces and potentially modifies these data structures. The user must supply marshaling annotations that determine the length of these data structures (or supply predicates, *e.g.,* those that determine how to stop traversing a linked list).

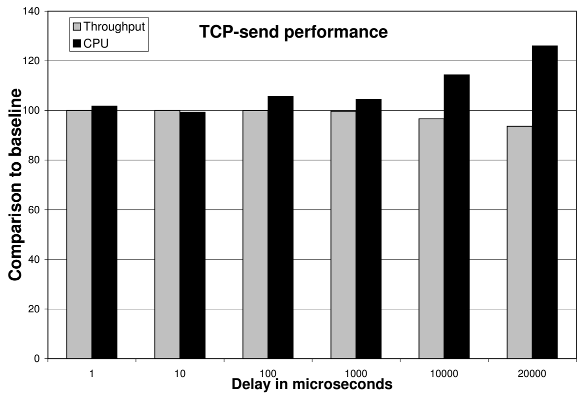

Figure 4: Performance of an e1000 microdriver.

5 Implementation and experiments

We have implemented several portions of the microdriver architecture and the refactoring tool. In particular, the refactoring tool automatically identifies a split and generates code for the kernel- and user-mode components. We have also implemented a static analysis algorithm to infer where marshaling annotations are necessary and are currently in the process of integrating this with the code-generator for marshaling/unmarshaling. To date, we have applied the tool to several network drivers. Because our infrastructure is still in development, we report our experience simulating the operation of an e1000 microdriver. In particular, we used our tool to generate code for the kernel- and user-mode components, and ran both the components in the kernel address-space, using delays to simulate change of protection domains.

The kernel-mode component of our e1000 microdriver contains just 25% of the code of the entire microdriver. In our experiments, we ran the e1000 microdriver on a dual-core 3Ghz Pentium-D machine running Linux-2.6.18. We measured network throughput and CPU utilization using `netperf` to send TCP packets (results for TCP/receive were similar and are omitted). We used buffers of size 256KB on both the sending and receiving side and transmitted 32KB messages. Figure 4 compares the network throughput and CPU utilization of the e1000 microdriver (with different values for delays) against a traditional e1000 device driver running under the same conditions. We observed that the microdriver has negligible overheads for network throughput and CPU utilization for delays under $10\mu s$. Even with a 20ms delay $(60,000,000$ machine cycles) we only observed a 6.3% drop in network throughput and 26% increase in CPU utilization. These results show that microdrivers reduce the amount of driver code running in the kernel without affecting common-case performance, and are thus a viable alternative to traditional device drivers.

6 Related work

Hardware-based isolation. Several architectures use hardware-based mechanisms to isolate faults in kernel extensions, in particular device drivers. These include Nooks [22] and VMM-based mechanisms [11, 15] that run each driver in its own protection domain. Microdrivers also use hardware, in particular, the process boundary, to isolate large parts (but not the entire) device driver. However, microdrivers can potentially perform better than these hardware-based isolation mechanisms because performance-critical code resides and executes in kernel address-space. In addition, microdrivers also reduce the amount of code running in the kernel. Microkernels (*e.g.,* [16, 23, 25]) also use the process boundary to isolate device drivers, and explicitly aim to reduce the amount of code executing with kernel privilege, but do so at the cost of reduced performance. Microdrivers offer poorer isolation than microkernels, but promise better performance.

Several recent efforts have focused on reducing the amount of driver code running in commodity operating system kernels [4, 7, 8, 13, 14, 17]. The FUSD framework [8] and the Microsoft user-mode driver framework [17] offer APIs to program user-mode device drivers that communicate with a kernel module. Chubb [7] and Leslie *et al.* [14] report user-mode driver performance comparable to in-kernel device drivers. However, unlike microdrivers, they require both kernel support, and rewriting device drivers, making them incompatible with existing operating systems.

Language-based isolation. SafeDrive [27] is a recent effort to improve device driver reliability by preventing type safety violations (and is similar in spirit to SFI [24]). SafeDrive reports good performance and is compatible with commodity operating systems. However, unlike microdrivers, SafeDrive does not reduce the amount of in-kernel code. Moreover, it does not offer protection against bugs that do not violate type safety (*e.g.,* violation of the locking protocol or other kernel API usage rules).

Program partitioning. Automatic and semi-automatic program partitioning techniques, much like the ones in Section 4, have also been applied to improve application security [5, 26] and to improve the performance of distributed components [12] and data-intensive user applications [20].

7 Conclusions

Microdrivers are a promising alternative to existing architectures for device drivers. They can improve system reliability by reducing the amount of code running in the kernel without adversely affecting common-case performance. Because microdrivers are compatible with commodity operating systems, they offer a path for running existing device drivers in user-mode with good common-case performance. This paper also shows that program analysis and transformation techniques can refactor existing drivers nearly automatically into microdrivers.

References

[1] The Minix3 operating system. `www.minix3.org`.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, February 1993.

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

[4] D. Blythe. Windows graphics overview. In *Windows Hardware Engineering Conference*, 2005.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *13th USENIX Security Symposium*, 2004.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *18th ACM SOSP*, 2001.

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

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

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th ACM SOSP*, 1995.

[10] A. Forin, D. Golub, and B. Bershad. An I/O system for Mach. In *USENIX Mach Symposium*, 1991.

[11] 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.

[12] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *4th ACM/USENIX OSDI*, 1999.

[13] B. Langley. Windows "Longhorn" display driver model—details and requirements. In *Windows Hardware Engineering Conference*, 2004.

[14] 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. *Journal of Computer Science and Technology*, 20(5), September 2005.

[15] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th ACM/USENIX OSDI*, 2004.

[16] J. Liedtke. On μ-kernel construction. In *15th ACM SOSP*, 1995.

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

[18] B. Murphy and M. R. Garzia. Software reliability engineering for mass market products. *The DoD Software Tech News*, 8(1), 2004.

[19] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft WinHec 2004 Presentation DDT301, 2003.

[20] A. Purohit, C. P. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in user-land, run in kernel-mode. In *9th HotOS*, 2003.

[21] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison Wesley, fifth edition, 1999.

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

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

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM SOSP*, 1993.

[25] 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.

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

[27] 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 *7th OSDI*, 2006.