A Sledgehammer Approach to Reuse of Legacy Device Drivers

Joshua LeVasseur Volkmar Uhlig

System Architecture Group
University of Karlsruhe, Germany

{jtl, volkmar}@ira.uka.de

Abstract

Device drivers account for the majority of an operating system's code base, and reuse of the existing driver infrastructure is a pragmatic requirement of any new OS project. New operating systems should benefit from the existing device driver code base without demanding legacy support from the kernel.

Instead of trying to directly integrate existing device drivers we propose a more radical approach. We run the unmodified device driver, with its complete original OS, isolated in a virtual machine. Our flexible approach, requiring only minimal support infrastructure, allows us to run any existing device driver, independently of the OS or driver vendor.

1 Introduction

In today's operating systems device drivers account for the largest part of the code base[1] [5]. The support of a sufficiently wide variety of legacy devices is a tremendous development effort for every OS project. Rewriting all device drivers can be considered impossible, both due to the development and testing effort as well as lack of documentation and accessibility of hardware.

A technique to reuse binary drivers is via cohosting [14], where the processor is multiplexed between two collaborating operating systems, with one providing device support. An alternative approach, typically applied at the source code level, is to reuse device drivers from the rich driver base of such systems as Linux or BSD with minimal or no changes [8, 10, 4, 2, 17, 9]. A software adaptation layer in the new OS provides the expected execution environment of the driver's original kernel. However, the adaptation layer requires a significant engineering and validation effort.

The engineering effort is influenced by the interface complexity. Typical monolithic construction permits device drivers to access their original OS's internals, and they may freely modify data structures and invoke kernel functions. Thus the interface surface area between the device driver and its original OS may be large. When emulated in a new environment, full semantics must be maintained. In many cases the semantics are undocumented, are not specified, or even diverge from their specifications. Device drivers within the same class of devices may not even share the same interface surface. Each re-used driver needs validation in the new environment, and revalidation in response to changes and updates to the original OS.

We propose an alternative approach for reuse of legacy device drivers. We use the complete original OS itself as the compatibility wrapper. The original OS effectively becomes an execution container for the driver. Using software partitioning techniques we strictly isolate this driver container and the new OS from each other. The partitioning can be performed by machine-virtualization [11] or paravirtualization [11, 19, 3].

To access devices driven by the driver container, we add a request interface which injects externally generated device requests. As in a traditional client-server scenario, the new OS acts as a client to the driver container.

At first our approach seems excessive, but it effectively eliminates most of the adaptation layer problems, and if carefully implemented it performs well and can be resource efficient. A necessity is that the driver OS can be extended with the request interface, which is usually trivial, for example by adding a loadable kernel module. By maintaining the complete execution environment we (almost) preserve all implicit semantics. When running in a full virtual machine the necessary code modification to the driver OS is limited to the interface extension. Thus, our approach is also applicable to proprietary operating systems. This particularly opens an update path to newer operating systems on irreplaceable legacy hardware (e.g., use Windows 95 drivers under Windows XP for obsolete hardware).

The remainder of the paper is structured as follows. In Section 2 we explain our architecture in more detail. Section 3 addresses virtualization, resource management, and

[1] For example, Linux 2.4.1 drivers account for 70% of the IA32 code base.

This paper was originally published in the
Proceedings of the 11th ACM European SIGOPS
Workshop, Leuven, Belgium, September 19–22, 2004.

trust issues of device drivers in virtual machines. In Section 4 we briefly evaluate our approach, and Section 5 discusses future work and concludes.

2 Architecture

The architecture supports several device driver reuse scenarios, from unmodified binary device drivers to recompiled device drivers in a paravirtualization environment. In each scenario, the device driver operating system (DD/OS) executes deprivileged within a virtual machine.

A translation module is added to the DD/OS, which behaves as a server in a client-server model, and maps client requests into sequences of DD/OS primitives. It likewise monitors for completion of the requests, and sends appropriate responses to the client (see Figure 1).

Via co-existing virtual machines, device drivers can execute in separate yet collaborating device driver operating systems. Device drivers from incompatible operating systems can coexist. If one device driver relies on another (e.g., a device needs bus services), and separate virtual machines host the device drivers, then a virtualized device module in the client DD/OS establishes a client-server relationship between two device driver VMs.

The scope of the software engineering effort to reuse device drivers, other than the virtual machine implementation itself, consists of the translation modules. For each class of devices in a particular flavor of DD/OS, a dedicated translation module must be developed. The translation module enables reuse of all devices within the device class.

The translation module may interface with the DD/OS at several layers of abstraction. Available interfaces include the user level API of the DD/OS (e.g., file access to emulate a raw disk), raw device access from user level (e.g., Linux raw sockets), abstracted kernel module interfaces such as the buffer cache, or the kernel primitives of the device drivers in the DD/OS.

3 Virtualization Issues

The isolation of the DD/OS via machine-virtualization and paravirtualization introduces several issues: the DD/OS performs DMA operations, it can violate the special timing needs of physical hardware, and it consumes resources beyond those which a device driver requires. Likewise, legacy operating systems are not designed to collaborate with other operating systems to control the devices within the system.

3.1 DMA Address Translation

DMA in commodity systems operates in the physical address space. Before issuing a DMA request to hardware the

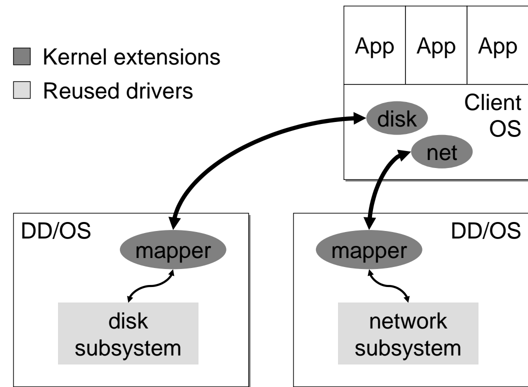

Figure 1. Interaction of a client with multiple device driver OS's. Each DD/OS has direct access to its managed device.

OS translates its kernel memory addresses into the physical address space. However, the additional memory translation indirection introduced by the virtual machine monitor renders the OS's addresses invalid, and thus referencing the wrong physical pages.

DMA address remapping, as supported by the AMD Opteron [1], HP's ZX1 chipset [6] and the Alpha 21172 chipset [7], provides address translation on DMA addresses, similarly to virtual memory for the main processor. The device is known as an IO-MMU. The virtual machine monitor (VMM), using the IO-MMU, can transparently remap the addresses on the I/O bus so that DMA works as expected by the DD/OS[2].

In systems without such hardware support DMA addresses have to be translated in software. Since knowledge to operate the DMA engines is integrated in the device drivers, the virtual machine monitor would have to understand each device's semantics to transparently rewrite the DMA addresses—which would effectively render our approach useless.

In a paravirtualized environment we have the ability to modify the address translation infrastructure in the DD/OS, taking the VM monitor's additional memory indirection into account[3]. Before issuing a device request the DD/OS consults the VMM for the current memory translation and pins it for the duration of the DMA operation.

In a fully virtualized environment, however, we cannot modify the DD/OS. Instead, we use a memory mapping

[2]To support concurrent DMA operations by multiple DD/OS's with different memory translations requires the IO-MMU to support distinct address spaces. We are currently investigating time-multiplexing of the IO-MMU between DD/OS's when the IO-MMU lacks support for address spaces.

[3]The changes required for a Linux 2.4.21 kernel are limited to three functions: `virt_to_bus`, `bus_to_virt`, and `page_to_bus`.

Figure 2. Remapping of physical memory for DMA. All driver VMs share code and read-only data. The page pool gets dynamically reallocated between the different OS instances. Memory areas in the VMs marked in white are valid for DMA.

scheme known from single address space OS's. By cautiously placing the DD/OS, the VMM ensures that memory which is accessed via DMA is mapped idempotently to physical memory. If an OS only performs DMA on read-only memory and dynamically allocated memory from the page pool then we can run multiple instances of the same DD/OS image concurrently. The OS code and read-only data are shared by all OS instances. The dynamic page pool is mapped to all DD/OS's, whereby each instance has its distinct allocated partition as shown in Figure 2. Memory ballooning [18] supports cooperative re-allocation of physical memory between DD/OS's. The single address space method allows multiple, concurrent DD/OS instances, thereby providing increased reliability and concurrency; however it is limited insofar that we can only use a single kernel image.

3.2 Trust and Reliability

Code with unrestricted access to DMA-capable hardware devices can circumvent standard memory protection mechanisms [12]. A malicious driver can potentially elevate its privileges by using DMA to replace OS code or data. In any system without explicit hardware support to restrict DMA accesses, we have to consider device drivers as part of the trusted computing base.

Isolating device drivers in separate virtual machines can still be beneficial. In [15] Swift et al. use very weak protection by leaving device drivers fully privileged, but still report a successful recovery rate of 99% for synthetically injected driver bugs. The fundamental assumption is that device drivers are usually faulty, but not malicious.

We differentiate between three trust scenarios. In the first scenario only the client of the DD/OS is untrusted. In the second scenario both the client as well as the DD/OS are untrusted by the VMM. In the third scenario the client and DD/OS additionally distrust each other. Note that the latter two scenarios can only be enforced with DMA restriction.

During a DMA operation page translations targeted by DMA have to stay constant, i.e. pinned. If the DD/OS's memory is not statically allocated it has to explicitly pin the memory. If the DD/OS initiates DMA in or out of the client's memory to eliminate copying overhead, that memory has to be pinned as well. In case the DD/OS is untrusted, the VMM has to enable DMA permissions to the memory and to ensure that the DD/OS cannot run denial-of-service attacks by pinning excessive amounts of physical memory.

If the DD/OS and client distrust each other, further provisions are required. If the DD/OS gets accounted for pinning memory, a malicious client could run a DoS attack against the driver. A similar attack by the DD/OS is possible if the client gets accounted for pinning. The solution is a cooperative approach with both untrusted parties involved. The client performs the pin operation on its own memory which eliminates a potential DoS attack by the DD/OS. Then, the DD/OS validates with the VMM that the pages are sufficiently pinned. By using timely bound pinning [13] guaranteed by the VMM, the DD/OS can safely perform the DMA operation.

3.3 Timing

Traditional OS's usually assume exclusive access to the processor with guarantees about execution time and timing behavior. With multiple VMs sharing resources this assumption does not hold.

If a VM is completely isolated from an external time base, it is sufficient to introduce a virtual time base and to run the virtual clock slower. However, since hardware devices are not adapted to this virtual time base, timing constraints may be violated. A slower clock is problematic when operations on the device have to be executed within a distinct time period, such as consecutive register accesses or interrupt handling.

The DD/OS is subject to the scheduling regime of the VMM. Arbitrary preemption of the DD/OS may lead to faulty behavior of the device. By using a heuristic we try to avoid preemption in time critical sections. Similarly to the scheme we described in [16] we differentiate between safe and unsafe preemption states for time-critical sections. A safe state is when preemption can take place because no critical section is active, an unsafe preemption state otherwise.

To ensure uninterrupted atomic execution of time critical sections, OS's usually disable interrupts on the local processor. Based on this observation we define an unsafe state to be when interrupts are disabled on a virtual CPU.

When the VMM decides to preempt a DD/OS in an unsafe state, the preemption is postponed until the DD/OS re-enables interrupts or voluntarily releases the CPU. If the architecture allows to selectively receive a trap upon reactivation of interrupts, as possible with IA32's virtual interrupt pending flag, the additional overhead is minimal, since limited to the rare cases when preemptions are postponed. An

upper time bound guarantees that the DD/OS cannot monopolize the system.

3.4 Resource Consumption

An OS is designed to control the complete system hardware, and so includes a data and code memory footprint for handling services unrelated to the reused device driver. It expects sufficient physical resources to satisfy classic workloads, and will try to optimally preallocate the resources in expectation of running classic workloads. The OS can consume resources even when the device driver is idle, such as CPU time to service a periodic timer interrupt.

The resource costs of a device driver and its OS extend beyond the resources inherently associated with a device driver. The device driver OS provides a compatibility layer at the cost of increased resource consumption. Every instance of a device driver OS will contribute to increased resource demands.

A virtual machine monitor can forcibly limit the resources consumed by its guest OS, and transparently reallocate resources on demand, as with traditional OS memory sharing techniques. Memory sharing and copy-on-write can additionally amortize the costs of multiple device driver instances. Page sharing is detectable via the process of page scanning [18]. Likewise, memory can be provided to the device driver OS when necessary, and later reclaimed via memory ballooning [18].

Some legacy operating systems provide control parameters to tune the system for space efficiency or speed efficiency. These parameters can be configured for a device driver workload to help reduce resource consumption.

With paravirtualization, the functionality of the device driver OS can be restricted in scope to the services necessary for executing a device driver, to the extent of disabling user-level application support. Modifications to its behavior can favor the device driver workload, rather than classic workloads.

3.5 Shared Hardware and Recursion

When running multiple DD/OS's some hardware resources are shared. This includes for example the PCI configuration space, interrupt controllers and processor configuration registers such as MTRRs on IA32. Different DD/OS's may even have conflicting configurations, such as for PCI bridges or interrupt controller modes.

If the driver for the shared device can be replaced we can use a simple recursive model. Instead of accessing the hardware directly by the DD/OS, we forward the request to yet another DD/OS. For drivers where replacement is impossible, we use full virtualization by trapping and translating device access (see Figure 3). The VMM then either synthesizes the device request and forwards it to the corresponding DD/OS or handles the request directly.

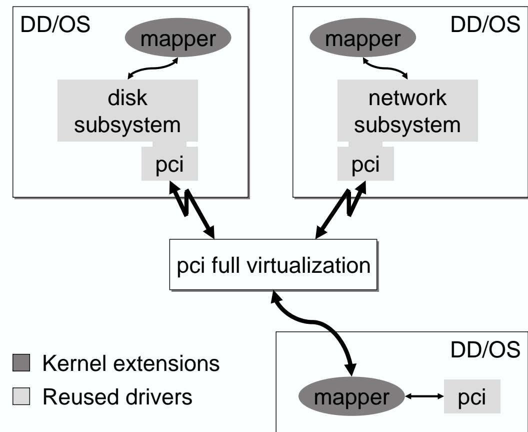

Figure 3. A disk and a network driver OS, unable to replace their PCI drivers, use full virtualization of PCI to achieve centralized reuse of an isolated PCI driver. The virtualization layer traps accesses to PCI, and transforms them into requests to the PCI DD/OS.

Commodity OS's usually scan the system busses for available devices and load the corresponding drivers. When giving unrestricted access to busses like PCI, each driver OS would potentially instantiate its own device driver. To avoid conflicts we partition devices and only report those which should be managed by a specific driver OS. By only reporting a single PCI bus with no bridges we try to avoid bus reconfiguration which could lead to conflicting configurations between multiple DD/OS's[4].

4 Evaluation

For an initial evaluation we use a set of paravirtualized Linux kernels adapted to our VMM, which we consider to approximate a fully virtualized solution, albeit with superior performance [3]. The first Linux kernel serves as the device driver OS and has access to most platform resources, but runs deprivileged. It exports a device driver request interface for the network device class, implemented as a loadable device driver module. The DD/OS reuses unmodified Linux network drivers, but recompiled to suit the paravirtualization environment.

The other Linux kernels lack direct device access and instead interface with the DD/OS for network services. They each use a specialized network device driver which com-

[4]For example Linux does not relocate devices if the BIOS correctly enabled and placed them.

municates with the DD/OS. The client Linux kernels, while adapted to the VMM, run unmodified Linux applications such as the Apache web server.

4.1 Network Interface

We implement a network model which treats the clients as direct participants of the external physical network. The DD/OS serves as a network bridge. The VMM allocates virtualized LAN addresses for the clients to accompany packets on the external network.

Network interfaces manage two types of packet flows: synchronous outbound packets, and asynchronous inbound packets. The clients own the pages backing outbound packets while the DD/OS owns the pages backing inbound packets. The network device driver interface uses two mechanisms to transfer packets between the DD/OS and client, to honor the semantics which accompany the two classes of page ownership.

For efficient outbound packet transfer, the client and server communicate enough information to support DMA directly from the client's packets. The server validates and translates the client's addresses, and validates pinning rights. The client and server communicate via a shared descriptor ring on a set of pages mapped into both address spaces.

The Linux network bridging module in the DD/OS directs relevant inbound packets to our translation layer. The translation layer queues the packets to the appropriate clients, and eventually copies the data to pages owned by the clients.

4.2 Initial Results

We have successfully tested the functionality of our translation layer reusing Linux's e1000 and e100 device drivers, which control the Intel gigabit and 100Mbit network controllers respectively.

The number of source lines of code for the translation layer in the DD/OS is 1802. Another 770 lines compose the virtual device driver in the client Linux. In contrast, the actual Linux e1000 device driver consists of 7903 source lines of code, and the Linux e100 device driver consists of 5633.

We measured the throughput of the TTCP benchmark executing in a paravirtualized Linux, and isolated from the DD/OS which provides the reused e1000 adapter. Throughput and CPU utilization were measured with the performance counters of the 2.8GHz Pentium 4, and compared to the performance of native Linux on the raw hardware. The reused device driver achieved throughput within 2.5% of native Linux, but with 1.46x more CPU utilization for sending, and 2.3x more CPU utilization for receiving (using packet copying). A detailed analysis, along with a study of CPU load, is planned for a future paper.

5 Conclusion and Outlook

We propose a solution for unmodified device driver reuse, which disentangles the goal of reuse from the task of designing a new operating system kernel. By partitioning the device driver at user level, supported by its original OS, we achieve unmodified driver reuse with minimal support infrastructure to interface a device driver with the system. The isolation further supports coexistence of device drivers from incompatible operating systems, and improves dependability.

We are working on determining the limits of the solution's performance, and how to further reduce code development for the interface layers. We also study how separation of device drivers improves dependability, and the associated performance costs.

References

[1] Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, April 2004.

[2] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. Wisniewski, and J. Xenidis. Utilizing Linux kernel components in K42. Technical report, IBM Watson Research, August 2002.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, October 19–22 2003.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 143–156, Saint Malo, France, October 5–8 1997.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Canada, October 21–24 2001.

[6] HP Technical Computing Devision. *HP zx1 mio ERS, Rev. 1.0.* Hewlett Packard, March 2003.

[7] Digital Equipment Corporation. *Digital Semiconductor 21172 Core Logic Chipset, Technical Reference Manual*, April 1996.

[8] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 38–51, Saint-Malo, France, October 5–8 1997.

[9] A. Forin, D. Golub, and B. Bershad. An I/O system for Mach 3.0. In *Proceedings of the Second USENIX Mach Symposium*, pages 163–176, Monterey, CA, November 20–22 1991.

[10] S. Goel and D. Duchamp. Linux device driver emulation in Mach. In *USENIX Annual Technical Conference*, pages 65–74, San Diego, CA, January 22–26 1996.

[11] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[12] B. Leslie and G. Heiser. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, School of Computer Science and Engineering, March 2003.

[13] J. Liedtke, V. Uhlig, K. Elphinstone, T. Jaeger, and Y. Park. How to schedule unlimited memory pinning of untrusted processes Or provisional ideas about service-neutrality. In *7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 153–159, Rio Rico, Arizona, March 29–30 1999.

[14] J. Sugerman, G. Venkitachalam, and B.H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 25–30 2001.

[15] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, Bolton Landing, NY, October 19–22 2003.

[16] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 6–7 2004.

[17] VMware. *VMware ESX Server I/O Adapter Compatibility Guide*, January 2003.

[18] C. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, MA, December 9–11 2002.

[19] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 195–209, Boston, MA, December 9–11 2002.