Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines

Joshua LeVasseur Volkmar Uhlig Jan Stoess Stefan Götz

University of Karlsruhe, Germany

Abstract

We propose a method to reuse unmodified device drivers and to improve system dependability using virtual machines. We run the unmodified device driver, with its original operating system, in a virtual machine. This approach enables extensive reuse of existing and unmodified drivers, independent of the OS or device vendor, significantly reducing the barrier to building new OS endeavors. By allowing distinct device drivers to reside in separate virtual machines, this technique isolates faults caused by defective or malicious drivers, thus improving a system's dependability.

We show that our technique requires minimal support infrastructure and provides strong fault isolation. Our prototype's network performance is within 3–8% of a native Linux system. Each additional virtual machine increases the CPU utilization by about 0.12%. We have successfully reused a wide variety of unmodified Linux network, disk, and PCI device drivers.

1 Introduction

The majority of today's operating system code base is accounted for by device drivers.[1] This has two major implications. First, any OS project that aims for even a reasonable breadth of device drivers faces either a major development and testing effort or has to support and integrate device drivers from a driver-rich OS (e.g., Linux or Windows). Even though almost all research OS projects reuse device drivers to a certain extent, full reuse for a significant driver base has remained an elusive goal and so far can be considered unachieved. The availability of drivers solely in binary format from the Windows driver base shows the limitations of integration and wrapping approaches as advocated by the OS-Kit project [10]. Also, implicit, undocumented, or in the worst case incorrectly documented OS behavior makes driver reuse with a fully emulated execution environment questionable.

The second implication of the large fraction of driver code in mature OS's is the extent of programming errors [7]. This is particularly problematic since testing requires accessibility to sometimes exotic or outdated hardware. The likelihood of programming errors in commonly used device drivers is probably much lower than in application code; however, such errors are often fatal. Device drivers, traditionally executing in privileged mode, can potentially propagate faults to other parts of the operating system, leading to sporadic system crashes.

In this paper we propose a pragmatic approach for full reuse and strong isolation of legacy device drivers. Instead of integrating device driver code we leave all drivers in their original and fully compatible execution environment—the original operating system. We run the device driver wrapped in the original operating system in a dedicated virtual machine (VM). Thus we can (almost) guarantee that semantics are preserved and that incompatibilities are limited to timing behavior introduced by virtual machine multiplexing.

The virtual machine environment also strongly isolates device drivers from the rest of the system to achieve fault containment. The isolation granularity depends on the number of collocated drivers in a single VM. By instantiating multiple collaborating VMs we can efficiently isolate device drivers with minimal resource overhead.

Reuse of device drivers and driver isolation are two important aspects of operating systems; however, they are usually discussed independently. With virtual machines, we propose to use a single abstraction to solve both problems in an extremely flexible, elegant, and efficient way.

2 Related Work

Our work uses known principles of hardware-based isolation to achieve driver reuse and improved system dependability. It is unique in the manner and the extent to which it accomplishes unmodified driver reuse, and how it improves system dependability, in terms of drivers, without system modification.

2.1 Reuse

Binary driver reuse has been achieved with cohosting, as used in VMware Workstation [32]. Cohosting multiplexes the processor between two collaborating operating systems, e.g., the driver OS and the VM monitor.

[1] Linux 2.4.1 drivers cover 70% of its IA32 code base [7].

Originally published in the Proceedings of the 6th Symposium on Operating System Design and Implementation, December 2004

When device activity is necessary, processor control is transfered to the driver OS in a world switch (which restores the interrupt handlers of the driver OS, etc.). The driver OS releases ownership of the processor upon device activity completion. The cohosting method offers no trust guarantees; both operating systems run fully privileged in supervisor mode and can interfere with each other.

Device drivers are commonly reused by transplanting source modules from a donor OS into the new OS [2, 4, 11, 15, 28, 35]. In contrast to cohosting, the new OS dominates the transplanted drivers. The transplant merges two independently developed code bases, glued together with support infrastructure. Ideally the two subsystems enjoy independence, such that the design of one does not interfere with the design of the other. Past work demonstrates that, despite great effort, conflicts are unavoidable and lead to compromises in the structure of the new OS. Transplantation has several categories of reuse issues, which we further describe.

Semantic Resource Conflicts

The transplanted driver obtains resources (memory, locks, CPU, etc.) from its new OS, subject to normal obligations and limitations, creating a new and risky relationship between the two components. In the reused driver's raw state, its manner of resource use could violate the resource's constraints. The misuse can cause accidental denial of service (e.g., the reused driver's non-preemptible interrupt handler consumes enough CPU to reduce the response latency for other subsystems), can cause corruption of a manager's state machine (e.g., invoking a non-reentrant memory allocator at interrupt time [15]), or can dead-lock in a multiprocessor system.

These semantic conflicts are due to the nature of OS design. A traditional OS divides bulk platform resources such as memory, processor time, and interrupts between an assortment of subsystems. The OS refines the bulk resources into linked lists, timers, hash tables, top-halves and bottom-halves, and other units acceptable for distributing and multiplexing between the subsystems. The resource refinements impose rules on the use of the resources, and depend on cooperation in maintaining the integrity of the state machines. Modules of independent origin substitute a glue layer for the cooperative design. For example, when a Linux driver waits for I/O, it removes the current thread from the run queue. To capture the intended thread operation and to map it into an operation appropriate for the new OS, the glue layer allocates a Linux thread control block when entering a reused Linux component [2,28]. In systems that use asynchronous I/O, the glue layer converts the thread operations into I/O continuation objects [15].

Sharing Conflicts

A transplanted driver shares the address space and privilege domain with the new OS. Their independently developed structures contend for the same resources in these two domains, and are subject to each other's faults.

Due to picky device drivers and non-modular code, a solution for fair address space sharing may be unachievable. The older Linux device drivers, dedicated to the IA32 platform, assumed virtual memory was idempotently mapped to physical memory. Reuse of these drivers requires modifications to the drivers or loss in flexibility of the address space layout. The authors in [28] decided not to support such device drivers, because the costs conflicted with their goals. The authors of [15] opted to support the drivers by remapping their OS.

Privileged operations generally have global side effects. When a device driver executes a privileged operation for the purposes of its local module, it likely affects the entire system. A device driver that disables processor interrupts disables them for all devices. Cooperatively designed components plan for the problem; driver reuse spoils cooperative design.

Engineering Effort

Device driver reuse reduces engineering effort in OS construction by avoiding reimplementation of the device drivers. Preserving confidence in the correctness of the original drivers is also important. When given device drivers that are already considered to be reliable and correct (error counts tend to reduce over time [7]), it is hoped that their reuse will carry along the same properties. Confidence in the new system follows from thorough knowledge of the principles behind the system's construction, accompanied by testing.

Reusing device drivers through transplantation reduces the overall engineering effort for constructing a new OS, but it still involves substantial work. In [10] Ford et al. report 12% of the OS-Kit code as glue code.

Engineering effort is necessary to extract the reused device drivers from their source operating systems, and to compile and link with the new operating system. The transplant requires glue layers to handle semantic differences and interface translation.

For implementation of a glue layer that gives us confidence in its reliability, intimate knowledge is required about the functionality, interfaces, and semantics of the reused device drivers. The authors in [2, 15, 28] all demonstrate intimate knowledge of their source operating systems.

The problems of semantic and resource conflicts multiply as device drivers from several source operating systems are transplanted into the new OS. Intimate knowledge of the internals of each source operating system

is indispensable. Driver update tracking can necessitate adaptation effort as well.

2.2 Dependability

The use of virtual machines to enhance reliability has been long known [16]. A variety of other techniques for enhancing system dependability also exist, such as safe languages and software isolation, and are complementary to our approach. The orthogonal design provided by our solution permits coexistence with incompatible subsystems and development methodologies.

User-level device driver frameworks [9, 11, 17, 20, 26, 31] are a known technique to improve dependability. They are typically deployed in a microkernel environment. Our approach also executes the device drivers at user level; however, we use the platform interface rather than a specialized and potentially more efficient API.

The recent Nooks project [33] shares our goal of retrofitting dependability enhancements in commodity systems. Their solution isolates drivers within protection domains, yet still executes them within the kernel with complete privileges. Without privilege isolation, complete fault isolation is not achieved, nor is detection of malicious drivers possible.

Nooks collocates with the target kernel, adding 22,000 lines of code to the Linux kernel's large footprint, all privileged. The Nooks approach is similar to second generation microkernels (such as L4, EROS, or K42) in providing address space services and synchronous communication across protection domains, but it doesn't take the next step to deprivilege the isolation domains (and thus exit to user-level, which is a minuscule overhead compared to the cost of address space switching on IA32).

To compensate for Linux's intricate subsystem entanglement, Nooks includes interposition services to maintain the integrity of resources shared between drivers. In our approach, we connect drivers at a high abstraction level—the request—and thus avoid the possibility of corrupting one driver by the actions of another driver.

Like us, another contemporary project [12, 13] uses paravirtualization for user-level device drivers, but focuses on achieving a unified device API and driver isolation. Our approach specifically leaves driver interfaces undefined and thus open for specializations and layer-cutting optimizations. Their work argues for a set of universal common-denominator interfaces per device class.

3 Approach

The traditional approach to device driver construction favors intimate relationships between the drivers and their kernel environments, interfering with easy reuse of drivers. On the other hand, *applications* in the same environments interface with their kernels via well defined APIs, permitting redeployment on similar kernels. Applications enjoy the benefits of orthogonal design.

To achieve reuse of device drivers from a wide selection of operating systems, we classify drivers as applications subject to orthogonal design, based on the following principles:

Resource delegation: The driver receives only bulk resources, such as memory at page granularity. The responsibility to further refine the bulk resources lies on the device driver. The device driver converts its memory into linked lists and hash tables, it manages its stack layout to support reentrant interrupts, and divides its CPU time between its threads.

Separation of name spaces: The device driver executes within its own address space. This requirement avoids naming conflicts between driver instances, and helps prevent faulty accesses to other memory.

Separation of privilege: Like applications, the device driver executes in unprivileged mode. It is unable to interfere with other OS components via privileged instructions.

Secure isolation: The device driver lacks access to the memory of non-trusting components. Likewise, the device driver is unable to affect the flow of execution in non-trusting components. These same properties also protect the device driver from the other system components. When non-trusting components share memory with the drivers, they are expected to protect their internal integrity; sensitive information is not stored on shared pages, or when it is, shadow copies are maintained in protected areas of the clients [14].

Common API: The driver allocates resources and controls devices with an API common to all device drivers. The API is well documented, well understood, powerfully expressive, and relatively static.

Most legacy device drivers in their native state violate these orthogonal design principles. They use internal interfaces of their native operating systems, expect refined resources, execute privileged instructions, and share a global address space. Their native operating systems partially satisfy our requirements. Operating systems provide resource delegation and refinement, and use a common API–the system's instruction set and platform architecture. By running the OS with the device driver in a virtual machine, we satisfy all of the principles and thus achieve orthogonal design.

3.1 Architecture

To reuse and isolate a device driver, we execute it and its native OS within a virtual machine. The driver directly controls its device via a pass-through enhancement to the virtual machine, which permits the device driver OS (DD/OS) to access the device's registers, ports, and receive hardware interrupts. The VM, however, inhibits the DD/OS from seeing and accessing devices which belong to other VMs.

The driver is reused by a client, which is any process in the system external to the VM, at a privileged or user level. The client interfaces with the driver via a translation module added to the device driver's OS. This module behaves as a server in a client-server model. It maps client requests into sequences of DD/OS primitives for accessing the device, and converts completed requests into appropriate responses to the client.

The translation module controls the DD/OS at one of several layers of abstraction: potentially 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., raw sockets), abstracted kernel module interfaces such as the buffer cache, or the kernel primitives of the device drivers in the DD/OS. It is important to choose the correct abstraction layer to achieve the full advantages of our device driver reuse approach; it enables a single translation module to reuse a wide variety of devices, hopefully without a serious performance penalty. For example, a translation module that interfaces with the block layer can reuse hard disks, floppy disks, optical media, etc., as opposed to reusing only a single device driver.

To isolate device drivers from each other, we execute the drivers in separate and co-existing virtual machines. This also enables simultaneous reuse of drivers from incompatible operating systems. When an isolated driver relies on another (e.g., a device needs bus services), then the two DD/OS's are assembled into a client-server relationship. See Figure 1 for a diagram of the architecture.

The requirement for a complete virtual machine implementation is avoidable by substituting a paravirtualized DD/OS for the unmodified DD/OS. In the paravirtualized model [3,16], the device driver's OS is modified to interface directly with the underlying system. However, most importantly, the device drivers in general remain unmodified; they only need to be recompiled.

3.2 Virtual Machine Environment

In our virtualization architecture we differentiate between five entities:

- The hypervisor is the privileged kernel, which securely multiplexes the processor between the virtual machines. It runs in privileged mode and enforces protection for memory and IO ports.

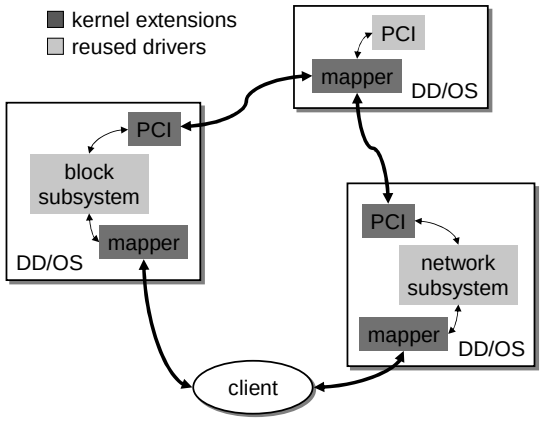

Figure 1: Device driver reuse and isolation. The kernel extensions represent the components loaded into the DD/OS's to coordinate device driver reuse. The block and network DD/OS's recursively use the PCI DD/OS.

- The virtual machine monitor (VMM) allocates and manages resources and implements the virtualization layer, such as translating access faults into device emulations. The VMM can be either collocated with the hypervisor in privileged mode or unprivileged and interacting with the hypervisor through a specialized interface.

- Device driver OS's host unmodified legacy device drivers and have pass-through access to the device. They control the device via either port IO or memory mapped IO and can initiate DMA. However, the VMM restricts access to only those devices that are managed by each particular DD/OS.

- Clients use device services exported by the DD/OS's, in a traditional client-server scenario. Recursive usage of driver OS's is possible; i.e. a client can act as a DD/OS for another client. The client could be the hypervisor itself.

- Translation modules are added to DD/OS's to provide device services to the clients. They provide the interface for the client-to-DD/OS communication, and map client requests into DD/OS primitives.

The hypervisor features a low-overhead communication mechanism for inter-virtual-machine communication. For message notification, each VM can raise a communication interrupt in another VM and thereby signal a pending request. Similarly, on request completion the DD/OS can raise a completion interrupt in the client OS.

The hypervisor provides a mechanism to share memory between multiple virtual machines. The VMM can

register memory areas of one VM in another VM's physical memory space, similarly to memory-mapped device drivers.

3.3 Client Requests

To provide access to its devices, the driver OS exports a virtual device interface that can be accessed by the client. The interface for client-to-DD/OS device communication is not defined by the hypervisor or the VMM but rather left to the specific translation module implementation. This allows for optimizations such as virtual interrupt coalescing, scatter-gather copying, shared buffers, and producer-consumer rings as used in Xen [3].

The translation module makes one or more memory pages accessible to the client OS and uses interrupts for signalling, subject to the particular interface and request requirements. This is very similar to interaction with real hardware devices. When the client signals the DD/OS, the VMM injects a virtual interrupt to cause invocation of the translation module. When the translation module signals the client in response, it invokes a method of the VMM, which can be implemented as a trap due to a specific privileged instruction, due to an access to an IO port, or due to a memory access.

3.4 Enhancing Dependability

Commodity operating systems continue to employ system construction techniques that favor performance over dependability [29]. If their authors intend to improve system dependability, they face the challenge of enhancing the large existing device driver base, potentially without source code access to all drivers.

Our architecture improves system availability and reliability, while avoiding modifications to the device drivers, via driver isolation within virtual machines. The VM provides a hardware protection domain, deprivileges the driver, and inhibits its access to the remainder of the system (while also protecting the driver from the rest of the system). The use of the virtual machine supports today's systems and is practical in that it avoids a large engineering effort.

The device driver isolation helps to improve *reliability* by preventing fault propagation between independent components. It improves driver *availability* by supporting fine grained driver restart (virtual machine reboot). Improved driver availability leads to increased system reliability when clients of the drivers promote fault containment. Proactive restart of drivers, to reset latent errors or to upgrade drivers, reduces dependence on recursive fault containment, thus helping to improve overall system reliability.

The DD/OS solution supports a continuum of configurations for device driver isolation, from individual driver isolation within dedicated VMs to grouping of all drivers within a single VM. Grouping drivers within the same DD/OS reduces the availability of the DD/OS to that of the least stable driver (if not further). Even with driver grouping, the system enjoys the benefits of fault isolation and driver restart.

Driver restart is a response to one of two event types: asynchronous (e.g., in response to fault detection [33], or in response to a malicious driver), or synchronous (e.g., live upgrades [23] or proactive restart [5]). The reboot response to driver failure returns the driver to a known good state: its initial state. The synchronous variant has the advantage of being able to quiesce the DD/OS prior to rebooting, and to negotiate with clients to complete sensitive tasks. Our solution permits restart of any driver via a VM reboot. However, drivers that rely on a hardware reset to reinitialize their devices may not be able to recover their devices.

The interface between the DD/OS and its clients provides a natural layer of indirection to handle the discontinuity in service due to restarts. The indirection *captures* accesses to a restarting driver. The access is either delayed until the connection is transparently restarted [23] (requiring the DD/OS or the VMM to preserve canonical cached client state across the restart), or reflected back to the client as a fault.

4 Virtualization Issues

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

4.1 DMA Address Translation

DMA operates on physical addresses of the machine. In a VM, memory addresses are subject to another address translation: from guest physical to host physical addresses. Since devices are not subject to TLB address translation, DMA addresses calculated inside the VM and fed to a hardware device reference incorrect host memory addresses.

Virtual machine monitors usually run device drivers at kernel privilege level [3, 21, 35]. The VMM exports virtual hardware devices to the VM, which may or may not resemble the real hardware in the system. On device access the monitor intercepts and translates requests and

DMA addresses to the machine's real hardware. Since all hardware accesses including DMA requests are intercepted, the VM is confined to its compartment.

When giving a VM unrestricted access to DMA-capable devices, the VM-to-host memory translation has to either be incorporated into all device requests or the DMA address translation has to be preserved. The particular approach depends on available hardware features and the virtualization method (full virtualization vs. paravirtualization).

In a paravirtualized environment the DD/OS can incorporate the VMM page mappings into the DMA address translation. For the Linux kernel this requires modification to only a few functions. The hypervisor also has to support an interface for querying and pinning the VM's memory translations.

When DMA address translation functions can't be overridden, the DD/OS's have to be mapped idempotently to physical memory. Apparently, this would restrict the system to a single DD/OS instance. But by borrowing ideas from single-address-space OS's we can overcome this restriction under certain circumstances. In many cases device drivers only issue DMA operations on dynamically allocated memory, such as the heap or page pool. Hence, only those pages require the restriction of being mapped idempotently. Using a memory balloon driver [36], pages can be reclaimed for use in other DD/OS's, effectively sharing DMA-capable pages between all DD/OS's (see Figure 2).

DMA from static data pages, such as microcode for SCSI controllers, further requires idempotent mapping of data pages. However, dynamic driver instantiation usually places drivers into memory allocated from the page pool anyway. Alternatively, one DD/OS can run completely unrelocated; multiple instances of the same OS can potentially share the read-only parts.

It is important to note that all solutions assume well-behaving DD/OS's. Without special hardware support, DD/OS's can still bypass memory protection by performing DMA to physical memory outside their compartments.

4.2 DMA and Trust

Code with unrestricted access to DMA-capable hardware devices can circumvent standard memory protection mechanisms. A malicious driver can potentially elevate its privileges by using DMA to replace hypervisor 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. Nooks [33] only offers very weak protection by leaving device drivers fully privileged, but still reports a successful recovery rate of 99% for synthetically injected driver bugs. The fundamental assumption is that device drivers may fault, but are not malicious.

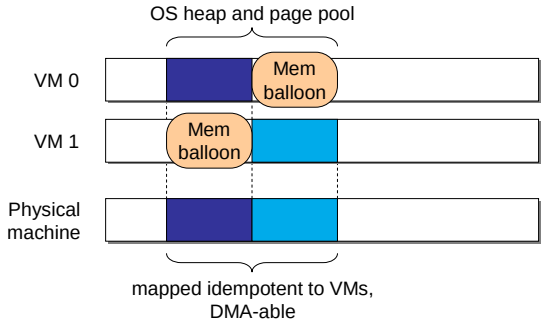

Figure 2: DMA memory allocation for two VMs. The balloon driver enables reallocation of the memory.

We differentiate between three trust scenarios. In the first scenario only the client of the DD/OS is untrusted. In the second case both the client as well as the DD/OS are untrusted by the hypervisor. In the third scenario the client and DD/OS also distrust each other. Note that the latter two cases can only be enforced with DMA restrictions as described in the next section.

During a DMA operation, page translations targeted by DMA have to stay constant. If the DD/OS's memory is not statically allocated it has to explicitly pin the memory. When the DD/OS initiates DMA in or out of the client's memory to eliminate copying overhead, it must pin that memory as well. In the case that the DD/OS is untrusted, the hypervisor 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.

When the DD/OS and client distrust each other, further provisions are required. If the DD/OS gets charged for pinning memory, a malicious client could run a DoS attack against the driver. A similar attack by the DD/OS against the client is possible when the DD/OS performs the pinning on behalf of the client. 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 hypervisor that the pages are sufficiently pinned. By using time-bound pinning [27] guaranteed by the hypervisor, the DD/OS can safely perform the DMA operation.

Page translations also have to stay pinned during a VM restart, since a faulting DD/OS may leave a device actively using DMA. All potentially targeted memory thus cannot be reclaimed until the VMM is sure that outstanding DMA operations have either completed or aborted.

Likewise, client OS's must not use memory handed out to the faulted DD/OS until its restart has completed.

4.3 IO-MMU and IO Contexts

The IO-MMU, initially designed to overcome the 32-bit address limitation for DMA in 64-bit systems, enables remapping bus addresses to host addresses at page granularity. IO-MMUs are, amongst others, available in AMD Opteron [1], Alpha 21172 [8], and HP Itanium systems [22]. They can be used to enforce access permissions for DMA operations and to translate DMA addresses. Thus, DD/OS's can be fully hardware-isolated from the VMM and other VMs, removing device drivers from the trusted computing base [24].

Tailored towards monolithic OS designs, IO-MMUs usually don't support multiple address contexts, such as per device, per slot, or per bus translations. The conflicting sets of virtual to physical mappings of isolated device drivers prevent simultaneous use of these IO-MMUs. We emulate multiple IO address contexts by time-multiplexing the IO-MMU between PCI devices. Resembling task scheduling, we periodically schedule IO-MMU contexts and enable bus access for only those devices that are associated with the active context.

The PCI specification [30] does not define a maximum access latency to the PCI bus, but only requires fair arbitration preventing deadlocks. Devices therefore have to be designed for potentially long bus access latencies—up to multiple milliseconds—which makes a coarse-grained scheduling approach feasible. The scheduling period has to be within the bounds of each device's timing tolerances; the particular handling of timeouts is specific to the device class. For example network cards simply start dropping packets when the card's internal buffers overflow, whereas the IDE DMA controller signals an error condition.[2]

A downside of time multiplexing is that the average available bus bandwidth for a device decreases and delivery latency increases. Benchmarks with a gigabit Ethernet NIC show a throughput decrease that is proportional to the allocated bus share. We further reduce the impact of time multiplexing by dynamically adapting bus allocations based on device utilization, preferring active and asynchronously operating devices.

The IO-MMU time multiplexing is a performance compromise to support device driver isolation on inadequate hardware, and is a proof-of-concept for our reuse and isolation goals. Future hardware solutions could eliminate the need for time multiplexing.

4.4 Resource Consumption

Each DD/OS consumes resources that extend beyond the inherent needs of the driver itself. The DD/OS needs a minimum amount of memory for code and data. Furthermore, each DD/OS has a certain dynamic processing overhead for periodic timers and housekeeping, such as page aging and cleaning. Periodic tasks in DD/OS's lead to cache and TLB footprints, imposing overhead on the clients even when not using any device drivers.

Page sharing as described in [36] significantly reduces the memory and cache footprint induced by individual DD/OS's. The sharing level can be very high when the same DD/OS kernel image is used multiple times and customized with loadable device drivers. In particular, the steady-state cache footprint of concurrent DD/OS's is reduced since the same housekeeping code is executed. It is important to note that memory sharing not only reduces overall memory consumption but also the cache footprint for physically tagged caches.

The VMM can further reduce the memory consumption of a VM by swapping unused pages to disk. However, this approach is infeasible for the DD/OS running the swap device itself (and its dependency chain). Hence, standard page swapping is permitted to all but the swap DD/OS. When treating the DD/OS as a black box, we cannot swap unused parts of the swap DD/OS via working set analysis. All parts of the OS must always be in main memory to guarantee full functionality even for rare corner cases.

Besides memory sharing and swapping, we use three methods to further reduce the memory footprint. Firstly, memory ballooning actively allocates memory in the DD/OS, leading to self-paging [18, 36]. The freed memory is handed back to the VMM. Secondly, we treat zero pages specially since they can be trivially restored. Finally, we compress [6] the remaining pages that do not belong to the active working set and that are not safe to swap, and uncompress them on access.

Page swapping and compression are limited to machines with DMA hardware that can fault on accesses to unmapped pages. Otherwise, a DMA operation could access invalid data (it must be assumed that all pages of a DD/OS are pinned and available for DMA when treating the DD/OS as a black box).

[2] IO-MMU time multiplexing is not fully transparent for all device classes. For example, the IDE DMA controller in our experimental AMD Opteron system requires dedicated handling. The IDE controller's behavior changes based on its DMA state: DMA startup or in-progress DMA. For DMA startup it can accept a multi-millisecond latency until its first bus access is permitted to proceed. But if its bus master access is rescinded for a multi-millisecond duration during an active DMA operation, it aborts instead of retrying the operation. The problem is that the millisecond scheduling period exceeds the device's latency. We therefore additionally check for in-progress DMA directly at the IDE controller and delay the preemption until DMA completion. However, to perform this test we need specific device knowledge—even though it is for a whole device class—compromising the transparency of our approach.

Periodic tasks like timers can create a non-negligible steady-state runtime overhead. In some cases the requirements on the runtime environment for a DD/OS whose sole purpose is to encapsulate a device driver can be weakened in favor of less resource consumption. For example, a certain clock drift is acceptable for an idle VM as long as it does not lead to malfunction of the driver itself, allowing us to schedule OS's less frequently or to simply drop their timer ticks.

4.5 Timing

Time multiplexing of multiple VMs can violate timing assumptions made in the operating system code. OS's assume linear time and non-interrupted execution. Introducing a virtual time base and slowing down the VM only works if there is no dependence on real time. Hardware devices, however, are not subject to this virtual time base. Violating the timing assumptions of device drivers, such as short delays using busy waiting or bound response times, can potentially lead to malfunctioning of the device.[3]

We use a scheduling heuristic to avoid preemption within time critical sections, very similar to our approach to lock-holder preemption avoidance described in [34]. When consecutive operations are time-bound, operating systems usually disable preemption, for example by disabling hardware interrupts. When the VMM scheduler would preempt a virtual processor but interrupts are disabled, we postpone the preemption until interrupts are re-enabled, thereby preserving the timing assumptions of the OS. This requires the VMM to trap the re-enable operation. Hard preemption after a maximum period avoids potential DoS attacks by malicious VMs.

4.6 Shared Hardware and Recursion

Device drivers assume exclusive access to the hardware device. In many cases exclusiveness can be guaranteed by partitioning the system and only giving device access to a single DD/OS. Inherently shared resources, such as the PCI bus and PCI configuration space, are incompatible with partitioning and require shared and synchronized access for multiple DD/OS's. Following our reuse approach, we give one DD/OS full access to the shared device; all other DD/OS's use driver stubs to access the shared device. The server part in the controlling DD/OS can then apply a fine-grained partitioning policy. For example, our PCI DD/OS partitions devices based on a configuration file, but makes PCI bridges read-only accessible to all client DD/OS's. To simplify VM device discovery, additional virtual devices can be registered.

In a fully virtualized environment, some device drivers cannot be replaced dynamically. Linux, for example, does not allow substituting the PCI bus driver. In those cases, full hardware emulation is required by the VMM. The number of such devices is quite limited. In the case of Linux the limitations include PCI, the interrupt controller, keyboard, mouse, and real-time clock.

5 Evaluation

We implemented a driver reuse system according to the architecture described in the prior sections, and assessed the architecture's performance, resource, and engineering costs. We evaluated reused drivers for the network, disk and PCI subsystems. We limit our evaluation to a paravirtualization environment.

To support a comparative performance analysis, we constructed a baseline system and a device driver reuse system that closely resemble each other. They use identical device driver code. They run the same benchmarks, utilizing the same protocol stacks and the same OS infrastructure. They differ in their architectures: the baseline uses its native device driver environment, while our system uses the driver reuse environment and is paravirtualized. The baseline OS is a standard Linux operating system. The device driver reuse system is constructed from a set of paravirtualized Linux OS's configured as DD/OS components and client components. The client OS communicates with the reused device drivers via special kernel modules installed into the client OS.

5.1 Virtualization Environment

The paravirtualization environment is based on the L4 microkernel [25]. L4 serves as a small privileged-mode hypervisor. It offers minimal abstractions and mechanisms to support isolation and communication for the virtual machines. Fewer than 13,000 lines of code run privileged.

The DD/OS and the client OS are provided by two different generations of the Linux kernel: versions 2.4.22 and 2.6.8.1. The 2.4 kernel was ported to the L4 environment in the tradition of the original L4Linux adaptation [19]. In contrast, we used a very lightweight adaptation of the 2.6 kernel to L4, with roughly 3000 additional lines of code (and only 450 lines intrusive). The paravirtualized Linux kernels use L4 mechanisms to receive interrupts, to schedule, to manage application memory, and to handle application system calls and exceptions.

The VMM, a user-level L4 task, coordinates resources

[3] Busy waiting, which relies on correct calibration at boot time, is particularly problematic when the calibration period exceeds a VM scheduling time slice and thus reports a slower processor. A device driver using busy waiting will then undershoot a device's minimal timing requirements.

such as memory, device mappings, and I/O port mappings for the DD/OS instances and the client OS.

All components communicate via L4 mechanisms. These mechanisms include the ability to establish shared pages, perform high-speed IPC, and to efficiently copy memory between address spaces. The mechanisms are coordinated by object interfaces defined in a high-level IDL, which are converted to optimized inlined assembler with an IDL compiler.

5.2 Translation Modules

For efficient data transfer, the client and DD/OS communicate enough information to support DMA directly from the client's pages via a shared producer-consumer command ring. In a typical sequence, the client adds device commands to the ring and activates the DD/OS via a virtual interrupt, and then the DD/OS services the command. Before performing the device DMA operation, the DD/OS validates the legality of the client's addresses and the client's pinning privileges.

The DD/OS does not generate virtual addresses for the client's pages; Linux device drivers are designed to support DMA operations on pages that are not addressable within the Linux kernel's virtual address space (by default, Linux can only address about 940MB of memory in its kernel space). The Linux drivers refer to pages indirectly via a page map. To leverage Linux's page map, we configure Linux with knowledge of all physical pages on the machine, but reserved from use (any attempts to access memory outside the DD/OS's VM causes page permission faults), and then convert client request addresses into page map offsets. In case a driver or subsystem places restrictions on acceptable addresses, it may be necessary to first copy the data.

Disk Interface The disk interface communicates with Linux's block layer, and is added to the DD/OS as a kernel module. It converts client disk operations into Linux block requests, and injects the block requests into the Linux kernel. Linux invokes the translation layer upon completion of the requests via a callback associated with each request. The block layer additionally supports the ability for the DD/OS to process requests out-of-order. The client and DD/OS share a set of request ID's to identify the reordered commands.

Network Interface The network interface has the additional feature of asynchronous inbound packet delivery. We developed our system to support multiple clients, and thus the DD/OS accepts the inbound packets into its own memory for demultiplexing. While outbound packets are transmitted from the client via DMA, inbound packets are securely copied from the DD/OS to the client by the L4 microkernel, thus protecting the client memory from the DD/OS (and requires agreement from the client to receive the packets). The L4 kernel creates temporary CPU-local memory mappings, within the L4 kernel space, to achieve an optimized copy.

The translation layer is added to the DD/OS as a device driver module. It represents itself to the DD/OS as a Linux network device, attached to a virtual interconnect. But it doesn't behave as a standard network device; instead it appends outbound packets directly to the real adapter's kernel packet queue (in the manner of network filters), where they are automatically rate controlled via the real device's driver feedback to the Linux kernel.

To participate directly on the physical network, the translation layer accepts inbound packets using the Linux ISO layer-two bridging module hook. The translation layer queues the packets to the appropriate client OS, and eventually copies to the client.[4]

PCI Interface When the PCI driver is isolated, it helps the other DD/OS instances discover their appropriate devices on the bus, and restricts device access to only the appropriate DD/OS instances.

The PCI interface is not performance critical. We forward all client PCI configuration-space read and write requests to the PCI DD/OS. It will perform write requests only for authorized clients. For read requests, it provides accurate information to the device's DD/OS, and contrived information to other clients.

We execute the PCI DD/OS at a lower priority than all other system components. With no timing requirements, it can tolerate severe clock drift.

5.3 Resource Consumption

For memory, we measured the active and steady-state page working set sizes of DD/OS instances, and considered the effect of page sharing and memory compression for all pages allocated to the DD/OS instances. For CPU, we focused on the idle cycle consumption (later sections explore the CPU costs of active workloads).

To avoid unnecessary resource consumption in the DD/OS, we configured the Linux kernel, via its build configuration, to include only the device drivers and functionality essential to handle the devices intended to be used in the benchmarks. The runtime environment of each DD/OS is a tiny ROM image which initializes into a single-user mode with almost no application presence.

[4] An alternative to packet copying, page remapping, has a prohibitively expensive TLB flush penalty on SMPs when maintaining TLB coherence. A future alternative is to use a spare hyperthread to copy the packets. If the network DD/OS has only a single client, then the client can provide the pages backing the inbound packets, avoiding the copy.

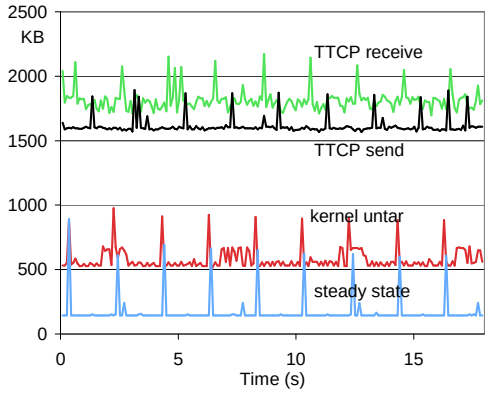

Figure 3: 90ms aggregate samples of Linux 2.6.8.1 DD/OS memory working sets when idle and for various disk and network benchmarks.

The data was collected while using Linux 2.6.8.1. The numbers are generally similar for Linux 2.4.22.

Working Set Figure 3 is a plot of memory page working sets of disk and network DD/OS's, where each sample covers 90ms of events. The "steady state" graph shows the inherent fixed cost of an idle DD/OS, usually around 144KB, with a housekeeping spike about every two seconds. The remaining graphs provide an idea of working set sizes during activity. The "ttcp receive" and "ttcp send" tests show the working set sizes during intense network activity. The "untar" test shows the working set response to the process of unarchiving a Linux kernel source tree to disk. The worst-case working set size reaches 2200KB, corresponding to high network activity. Our configuration is susceptible to a large working set for network activity because the DD/OS buffers incoming packets within its own memory. However, due to Linux's reuse of packet buffers the DD/OS working set size remains bounded.

Memory Compression To test the possibility of sharing and compressing the pages that back the DD/OS instances, we performed an offline analysis of a snapshot of a particular DD/OS configuration. The tested configuration included three DD/OS instances, one each for PCI, IDE, and the Intel e1000 gigabit. The PCI VM was configured with 12MB and the others with 20MB memory each. We ran the PostMark benchmark stressing a VM with Linux 2.6 serving files via NFS from the local IDE disk over the network. The active memory working set for all DD/OS's was 2.5MB.

For systems without an IO-MMU, the memory consumption can only be reduced by cooperative memory ballooning [36]. With the balloon driver in the DD/OS's we can reclaim 33% of the memory.

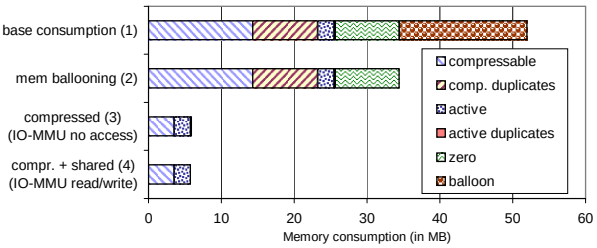

Figure 4: (1) Combined memory consumption of disk, network, and PCI DD/OS's with 20MB, 20MB, and 12MB VMs, (2) after memory ballooning, (3) with memory compression, and (4) memory compression and sharing.

Using an IO-MMU that can recover from page faults, we can revoke page access rights and compress memory that is not part of the active working set. Support of read-only page access rights by the IO-MMUs furthermore enables sharing of identical pages of the active working set via copy-on-write. We searched for duplicate pages among the three DD/OS instances. Any duplicate page is shareable whether it is in an active working set or not. A page in any DD/OS instance is additionally upgraded to an active page if it has a duplicate in any working set, to avoid having a compressed as well as an uncompressed copy. Finally, the IO-MMU enables us to reclaim all zero pages uncooperatively. For the given setup, up to 89% of the allocated memory can be reclaimed, reducing the overall memory footprint of three concurrent DD/OS's to 6MB (see Figure 4).

Without an IO-MMU, gray-box knowledge enables DD/OS paging. For example, the memory of Linux's page map is never used for a DMA operation, and is thus pageable. Furthermore, the network and block DD/OS each had a contiguous 6.9 MB identical region in their page maps, suitable for sharing.

CPU Utilization The steady state of a DD/OS has an inherent CPU utilization cost, not just influenced by internal activities, but also by the number of DD/OS's in the system. We measured the DD/OS CPU utilization response to additional DD/OS instances; the first eight DD/OS's each consume 0.12% of the CPU, and then the ninth consumes 0.15%, and the tenth consumes 0.23% (see Figure 5).

The DD/OS's were idle with no device activity. Only the first DD/OS was attached to a device—the PCI bus. The others contained a single device driver (the e1000).

The machine was a Pentium 4 2.8 GHz with a 1MB L2 cache, which can almost fit the steady-state memory working sets of seven DD/OS instances (at 144KB each, see Figure 3). The L2 cache miss rate began to rapidly rise with the eighth DD/OS, leading to an inflection in the CPU utilization curve.

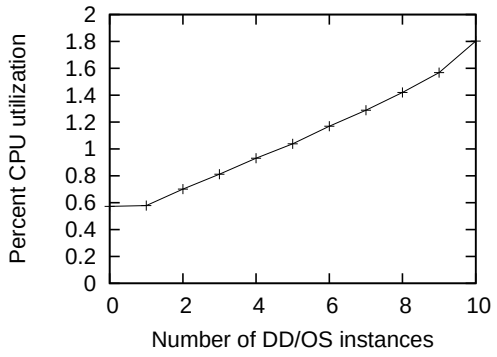

Figure 5: Incremental CPU utilization for additional steady-state DD/OS instances, representing the fi xed cost of executing a DD/OS.

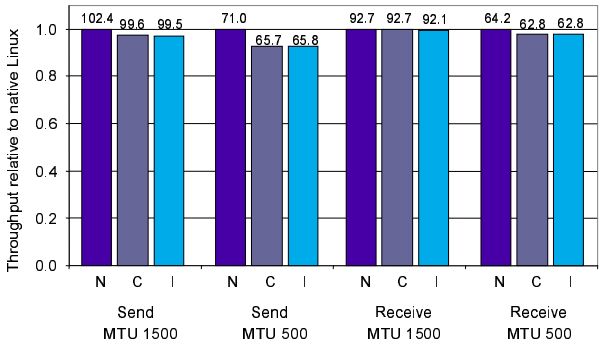

Figure 6: Normalized TTCP throughput results for native Linux (N), consolidated (C), and isolated (I) DD/OS's. Absolute throughput given in MB/s.

5.4 Performance

A set of benchmarks allowed us to explore the performance costs of the DD/OS approach to device driver reuse, stressing one driver at a time, and then using network and disk drivers together. The networking benchmarks were selected to help provide a point of comparison with recent literature.

We executed our benchmarks with two device driver reuse scenarios: (1) with all drivers consolidated in a single DD/OS, and (2) with the devices isolated in dedicated DD/OS instances. For a baseline, the benchmarks are also executed within the original, native device driver environment.

The benchmark OS ran Debian Sarge with the Linux 2.6 kernels, constrained to 768MB. When using the Linux 2.4 kernels, performance numbers were very similar. The hardware used in the test system was a Pentium 4 2.8 GHz processor, with an Intel 82540 gigabit network PCI card, and a desktop SATA disk (Maxtor 6Y120M0).

TTCP Figure 6 presents the throughput of the TTCP benchmark relative to the native throughput, using two packet sizes. Throughput at the 1500-byte packet size remains within 3% of native, and drops to 8% of native for 500-byte packets. Linux performs the packet sizing within the kernel, rather than within TTCP, via use of Linux's maximum transmission unit (MTU) parameter, avoiding a per-packet address space transition. The CPU utilization relative to native Linux was 1.6x for send, 2.06x for receive with 1500-byte MTU, and 2.22x for receive with 500-byte MTU. As expected, network receive generated a larger CPU load than network send due to extra packet copies. TTCP was configured for a 128KB socket buffer size.

Netperf The Netperf benchmark confirmed the TTCP MTU 1500 results; throughput with driver reuse remained within 3% of native, with 1.6x CPU utilization for sending, and up to 2.03x CPU utilization for receiving. The native throughput was 98.5 MB/s. A substantial increase in TLB and L2 cache misses led to higher CPU utilization. These misses are inherent to our test-platform; the Pentium 4 flushes TLBs and L1 caches on every context switch between the client and DD/OS. The Netperf benchmark transfered one gigabyte, with a 32KB send and receive size, and a 256KB socket buffer size.

Disk Figure 7 presents the results of our *streaming* disk benchmark for the isolated DD/OS's (consolidated results are identical). The benchmark highlights the overhead of our solution, as opposed to masking it with random-access disk latency. The benchmark bypasses the client's buffer cache (using a Linux raw device) and file system (by directly accessing the disk partition). We thus avoid timing the behavior of the file system. Native throughput averaged 50.75 MB/s with a standard deviation of 0.46 MB/s. For driver reuse, the throughput was nearly identical and the difference less than half the standard deviation, with CPU utilization ranging from 1.2x to 1.9x native.

Application-Level We studied application-level performance with the PostMark benchmark, run over NFS. This benchmark emulates the file transaction behavior of an Internet electronic mail server, and in our scenario, the file storage is provided by an NFS server machine. The benchmark itself executes on a client machine. The NFS server used our driver reuse framework, and was configured as in the microbenchmarks. The client had a 1.4 GHz Pentium 4, 256MB memory, a 64MB Debian RAM disk, an Intel 82540 gigabit Ethernet PCI card, and executed a native Linux 2.6.8.1 kernel. The performance of

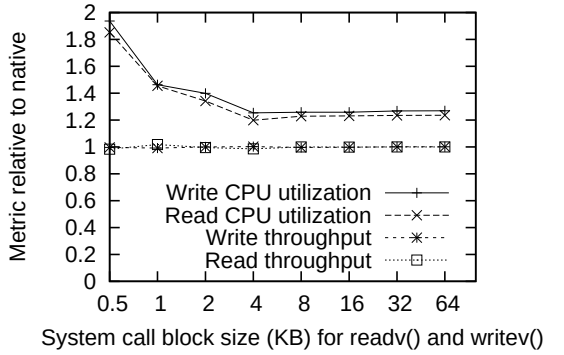

Figure 7: Throughput and CPU use relative to native Linux for disk streaming read and write.

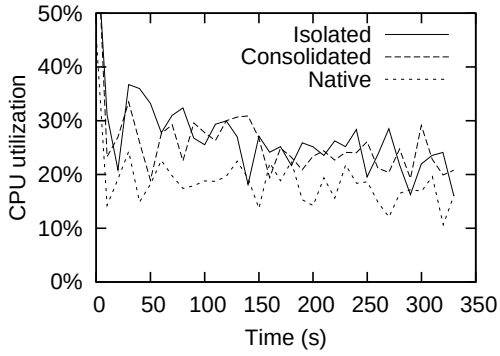

Figure 8: CPU utilization for the NFS server machine while handling the PostMark benchmark.

the NFS server was nearly identical for all driver scenarios, for native Linux and for driver reuse, with an average runtime of 343.4 seconds. The standard deviation, 2.4%, was over twice the loss in performance for driver reuse. Both the isolated and consolidated driver reuse configurations had higher CPU utilization than native Linux; see Figure 8 for CPU utilization traces of the NFS server machine covering the duration of the benchmark. The benchmark starts with a large CPU spike due to file creation. Postmark was configured for file sizes ranging from 500-bytes to 1MB, a working set of 1000 files, and 10000 file transactions.

5.5 IO-MMU

We used a 1.6 GHz AMD Opteron system with an AMD 8111 chipset to evaluate IO-MMU time multiplexing. The chipset's graphics aperture relocation table mechanism relocates up to 2GB of the 4GB DMA space at a 4KB granularity [1]. The chipset only supports read-write and no-access permissions.

Each virtual machine running a DD/OS has a dedicated IO-MMU page table which is synchronized with

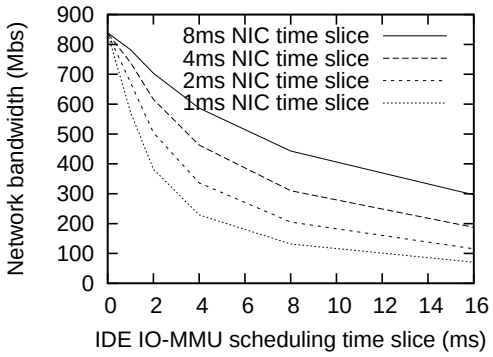

Figure 9: Network bandwidth in response to various IO-MMU context scheduling rates.

the guest-physical-to-host-physical mappings of the VM. When clients grant the DD/OS access to parts of their memory, appropriate entries are added to the IO-MMU page table as well.

The VMM connects the managed PCI devices of each DD/OS with their respective IO-MMU contexts. Periodically, but independent of the processor scheduler, we switch between the IO contexts. On context switch, the hypervisor enables and disables bus master access in the PCI configuration space for the respective devices. Our shortest scheduling granularity of 1ms is limited by the frequency of the periodic timer.

We evaluated the performance and overhead of scheduling IO-MMU contexts, as well as the bounds of the scheduling period for hardware devices. The test system contained two DD/OS's, one driving an Intel e1000 gigabit Ethernet adapter and the other handling the IDE disk controller.

First, and most importantly, we can completely isolate the physical memory covered by the IO-MMU and transparently relocate both VMs. Neither VM is able to perform DMA to memory outside its compartment. We ran the TTCP benchmark and varied the bus allocation for the NIC and disk controller. The network throughput scaled almost linearly with the bus allocation. The NIC started dropping packets when it lost access to the bus for more than 8ms. Figure 9 shows the achieved network bandwidth for various scheduling configurations.

The IDE controller is less bandwidth-sensitive since the throughput is bounded by disk latency. However, our scheduling granularity of 1ms exceeds the timeout for in-progress transactions. When disabling bus master we therefore postpone IDE deactivation when operations are still in-flight. The overhead for IO-MMU context switching was a 1% increase in CPU utilization.

	server	client	common	total
network	1152	770	244	2166
block 2.4	805	659	108	1572
block 2.6	751	546	0	1297
PCI	596	209	52	857
common	0	0	620	620
total	3304	2184	1024	

Figure 10: Itemization of source lines of code used to implement our evaluation environment. Common lines are counted once.

5.6 Engineering Effort

We estimate engineering effort in man hours and in lines of code. The translation modules and client device drivers for the block and network, along with the user-level VMM, were written by a single student over roughly a two month period, originally for L4Linux 2.4. This student already had experience with Linux network driver development for a paravirtualized Linux on L4. A second student implemented the PCI support within one week.

The 2.4 network translation module was easily upgraded to serve as the translation module for Linux 2.6, with minor changes. However the 2.4 block translation module was mostly incompatible with 2.6's internal API (Linux 2.6 introduced a new block subsystem). We thus wrote new block translation and client device drivers for 2.6. We successfully reused the 2.6 block and network drivers with the 2.4 client, and vice versa.

See Figure 10 for an itemization of the lines of code. The figure distinguishes between lines specific to the translation modules added to the server, lines specific to the virtual device drivers added to the client, and additional lines that are common (and are counted once).

The achieved code reuse ratio is 99.9% for NIC drivers in Linux; the translation modules add 0.1% to their code base. When we additionally include all code required for the virtualization—the L4 microkernel, the VMM, and the paravirtualization modifications—we still achieve a reuse ratio of 91% just for Linux's NIC driver base.

The engineering effort enabled us to successfully reuse Linux device drivers with all of our tested lab hardware. The following drivers were tested: Intel gigabit, Intel 100 Mbit, Tulip (with a variety of Tulip compatible hardware), Broadcom gigabit, pcnet32, ATA and SATA IDE, and a variety of uniprocessor and SMP chipsets for Intel Pentium 3/4 and AMD Opteron processors.

6 Discussion and Future Work

We presented a new approach to reusing unmodified device drivers and enhancing system dependability using virtual machines, but evaluated only a paravirtualized implementation. Paravirtualization is an enhanced machine API that relocates some functionality from the guest OS to the VMM and hypervisor [16]. For example, it permits our DD/OS instances to directly translate their virtual addresses into bus addresses for DMA. It also provides performance benefits [3, 16] compared to use of the real machine API. We have discussed the issues related to device driver pass-through with full virtualization, and consider our paravirtualization implementation to be an approximation. In terms of correctness, the primary difference relates to proper address translation for DMA operations, which becomes irrelevant with hardware device isolation (such as the IO-MMU). In terms of performance, the paravirtualization numbers underestimate the costs of a fully-virtualized solution.

Our system currently supports a sufficiently large subset of device classes to be self-hosting in a server environment. We have not addressed the desktop environment, which requires support for the graphics console, USB, Firewire, etc.

Generic driver stubs only provide access to the least common denominator, thereby hiding more advanced hardware features. Our client-server model enables device access at any level in the software hierarchy of the DD/OS, even allowing programming against richer OS APIs like TWAIN, or enabling vendor-specific features such as DVD burning. Using the appropriate software engineering methods, e.g., an IDL compiler, one can quickly generate cross-address-space interfaces that support APIs with rich feature sets.

7 Conclusion

Widely used operating systems support a variety of devices; for example, in Linux 2.4 on IA32, 70% of 1.6 million lines of kernel code implement device support [7]. New operating system endeavors have the choice of either leveraging the existing device drivers, or expending effort to replicate the driver base. We present a technique that enables unmodified reuse of the existing driver base, and most importantly, does so in a manner that promotes independence of the new OS endeavor from the reused drivers.

The driver independence provides an opportunity to improve system dependability. The solution fortifies the reused drivers (to the extent supported by hardware) to promote enhanced reliability and availability (with independent driver restart).

Our method for reusing unmodified drivers and improving system dependability via virtual machines achieves good performance. For networking, where packetized throughput is latency-sensitive, the throughput remains within 3–8% of the native system. The driver isolation requires extra CPU utilization, which can be re-

duced with hardware acceleration (such as direct DMA for inbound packets).

The DD/OS solution is designed for minimal engineering effort, even supporting reuse of binary drivers. The interface implementation between the new OS and reused drivers constitutes a trivial amount of code, which leverages the vast world of legacy drivers. Driver source code, by design, remains unmodified.

References

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

[2] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, et al. Utilizing Linux kernel components in K42. Technical report, IBM Watson Research, Aug. 2002.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, et al. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[5] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Eighth IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.

[6] R. Cervera, T. Cortes, and Y. Becerra. Improving application performance through swap compression. In *Usenix Annual Technical Conference*, Monterey, CA, June 1999.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.

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

[9] K. Elphinstone and S. Götz. Initial evaluation of a user-level device driver framework. In *9th Asia-Pacific Computer Systems Architecture Conference*, Beijing, China, Sept. 2004.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[11] A. Forin, D. Golub, and B. Bershad. An I/O system for Mach 3.0. In *Proc. of the Second USENIX Mach Symposium*, Monterey, CA, Nov. 1991.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, Computer Laboratory, Aug. 2004.

[13] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, Oct. 2004.

[14] A. Geffhut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, et al. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, Sept. 2000.

[15] S. Goel and D. Duchamp. Linux device driver emulation in Mach. In *USENIX Annual Technical Conference*, San Diego, CA, Jan. 1996.

[16] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6), 1974.

[17] D. B. Golub, G. G. Sotomayor, Jr., and F. L. Rawson III. An architecture for device drivers executing as user-level tasks. In *Proc. of the USENIX Mach III Symposium*, Sante Fe, NM, Apr. 1993.

[18] S. M. Hand. Self-paging in the Nemesis operating system. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[19] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symposium on Operating System Principles*, Saint-Malo, France, Oct. 1997.

[20] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08-Juli-2003, TU Dresden, Dresden, Germany, July 2003.

[21] J. Honeycutt. *Microsoft Virtual PC 2004 Technical Overview*. Microsoft, Nov. 2003.

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

[23] K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, et al. Position summary: Supporting hot-swappable components for system software. In *Eighth IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.

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

[25] J. Liedtke. On μ-kernel construction. In *Proc. of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995.

[26] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ-kernel based OS. *ACM SIGOPS Operating Systems Review*, 25(2), Apr. 1991.

[27] 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*, Rio Rico, AR, Mar. 1999.

[28] K. V. Maren. The Fluke device driver framework. Master's thesis, University of Utah, Dec. 1999.

[29] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, et al. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, U.C. Berkely Computer Science, Mar. 2002.

[30] PCI Special Interest Group. *PCI Local Bus Specification, Rev. 2.1*, June 1995.

[31] D. S. Ritchie and G. W. Neufeld. User level IPC and device management in the Raven kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium*, San Diego, CA, Sept. 1993.

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

[33] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[34] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of the 3rd Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[35] VMware. *VMware ESX Server I/O Adapter Compatibility Guide*, Jan. 2003.

[36] C. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.