TwinDrivers: Semi-Automatic Derivation of Fast and Safe Hypervisor Network Drivers from Guest OS Drivers

Aravind Menon, Simon Schubert and Willy Zwaenepoel
School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne
{aravind.menon, simon.schubert, willy.zwaenepoel}@epfl.ch

Abstract

In a virtualized environment, device drivers are often run inside a virtual machine (VM) rather than in the hypervisor, for reasons of safety and reduction in software engineering effort. Unfortunately, this approach results in poor performance for I/O-intensive devices such as network cards. The alternative approach of running device drivers directly in the hypervisor yields better performance, but results in the loss of safety guarantees for the hypervisor and incurs additional software engineering costs.

In this paper we present TwinDrivers, a framework which allows us to *semi-automatically* create *safe* and *efficient* hypervisor drivers from guest OS drivers. The hypervisor driver runs directly in the hypervisor, but its data resides completely in the driver VM address space. A *Software Virtual Memory* mechanism allows the driver to access its VM data efficiently from the hypervisor running in any guest context, and also protects the hypervisor from invalid memory accesses from the driver. An *upcall* mechanism allows the hypervisor to largely reuse the driver support infrastructure present in the VM. The TwinDriver system thus combines most of the performance benefits of hypervisor-based driver approaches with the safety and software engineering benefits of VM-based driver approaches.

Using the TwinDrivers hypervisor driver, we are able to improve the guest domain networking throughput in Xen by a factor of 2.4 for transmit workloads, and 2.1 for receive workloads, both in CPU-scaled units, and achieve close to 64-67% of native Linux throughput.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance

General Terms Performance, Measurement

1. Introduction

In a virtualized environment it is desirable, for reasons of safety and reduction in software engineering effort, to run device drivers inside a virtual machine (VM) rather than in the hypervisor. By running the drivers in a VM, a bug in the driver does not compromise the hypervisor or other VMs. Furthermore, it avoids having to (re)implement the entire driver support infrastructure in the hypervisor. Instead, one can simply re-use the driver support infrastructure already present in the guest operating system. This strategy is used in the Xen virtual machine environment [3] and in L4 [10].

The drawback of this approach is loss of performance. Since the device driver is located in a different driver VM, and thus a different address space than the guest VM, extra context switching overhead is incurred in invoking the device driver and in interrupt handling. Thus, for instance, it has been reported that network performance in Xen is a factor of 3 to 4 lower than native Linux performance [11, 12, 15].

The alternative to this approach is to run the device driver directly in the hypervisor. This approach gives better performance because it avoids context switches for calls between the hypervisor driver and the guest VM. Unfortunately, this approach requires the entire driver and its support library to be either developed anew for the hypervisor, or to be ported from an existing operating system. Both approaches incur a significant software development effort. In addition, this approach also leaves the hypervisor vulnerable to bugs in the device driver.

This paper tackles the tradeoff between performance on the one hand and safety and reduction in coding effort on the other hand. Our goal is to combine the performance benefits of the hypervisor-based driver approach with the safety and software engineering benefits of the VM-based driver approach.

We take a driver developed for a guest operating system, such as Linux, and we *semi-automatically* produce from it, by binary rewriting, a driver that *efficiently and safely* runs in the hypervisor. At runtime, two instances of the driver are run at the same time: The original one, which we call the VM instance, runs in a VM. The derived one, which we call the hypervisor instance, runs in the hypervisor. The hypervisor instance takes care of performance-critical operations of the device driver. For instance, for a network card driver, this includes transmitting and receiving packets. The VM instance takes care of the other operations such as device configuration, management, error handling, etc.

Although there are two separate instances of the driver running, there is only a single instance of the driver data, residing in the VM address space. The hypervisor instance accesses only the driver and VM data structures in the VM, and does not access any hypervisor data structures. From this simple rule derives the *safety* of the approach: The hypervisor instance cannot access, and therefore cannot corrupt the hypervisor data structures.

Although the driver data is located in the VM address space, the hypervisor instance can access this data while running in any guest VM context by using an address translation mechanism called *Software Virtual Memory* (SVM). This allows the hypervisor to invoke its driver instance while running in any guest context without switching address spaces. This is the key to achieving good *performance*.

Keeping only a single copy of the data in the VM address space also allows the hypervisor instance to invoke the driver support routines in the VM for operations on these data structures. This is done through an *upcall* mechanism from the hypervisor to the VM. The *upcall* approach avoids the implementation in the hypervisor of the entire set of driver support routines. Instead, the hypervisor only implements a small set of performance-critical support routines needed to achieve good performance. This is the key to reducing the *software engineering effort* to support the hypervisor driver instance.

We have implemented the ideas described above in the TwinDrivers system. Our implementation is targeted at the Xen hypervisor and Linux network drivers, but we believe the ideas are generally applicable. We have used our binary rewriting system to twin the Intel e1000 driver. The TwinDrivers system allows us to improve the Xen guest domain networking throughput in CPU-scaled units by a factor of 2.4 for transmit workloads, and by a factor of 2.1 for receive workloads. The resulting throughput is also within 64 to 67% of native Linux throughput.

The outline of the rest of this paper is as follows. Section 2 provides some background on the Xen I/O system. Section 3 presents the principles underlying the TwinDrivers approach. Section 4 presents in more detail the design of the TwinDrivers approach, and Section 5 presents our current implementation. Section 6 presents our performance results for network I/O. Section 7 discussed related work. Section 8 presents our conclusions.

2. Background on Xen I/O

The Xen VMM uses the so called ‘hosted’ virtual machine model in which device drivers are run in a ‘driver domain’ (dom0), which provides device I/O services for guest domains (guest VMs). Figure 1 shows a high-level picture of the network driver architecture in Xen. Guest domains are provided with a *frontend* virtual network interface, which is connected to the physical interface (NIC) driver in the driver domain through a network *bridge* and a *backend* interface. Transmit requests from the guest domain and NIC interrupts result in switches to the driver domain to invoke the device driver. More details of the Xen architecture can be found in [7].

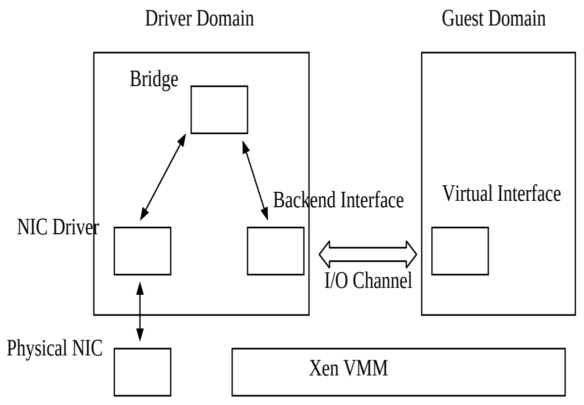

Figure 1: Xen I/O Architecture

The ‘hosted’ driver architecture incurs a significant performance overhead for guest domains, which has been studied in [15, 21, 12]. The biggest overhead is incurred due to the frequent context switches between the driver domain and guest domains for driver invocation and interrupt handling, which results in increased TLB and cache misses [12]. There are additional overheads as well, incurred because of the expensive bridging and *grant table* operations in the driver domain [15]. The overall performance impact of these overheads is a reduction in the network performance of Xen guest domains by a factor of 3-4 [11, 15].

In contrast, in a hypervisor-based driver model, the device driver executes directly in the hypervisor, and thus avoids the context switches on device invocation.

3. Principles

We discuss the design and principles underlying the TwinDrivers approach in the context of the Xen VMM, using the example of a network interface card driver. Figure 2 shows the overall architecture of the TwinDrivers approach. TwinDrivers uses two driver instances, one running in dom0 and one running in the hypervisor, but only one instance of the driver data, residing in dom0.

3.1 Two Driver Instances

We take a device driver from the Linux driver domain, and rewrite the binary to produce a driver that can execute in the hypervisor. At runtime, two instances of the driver are run at

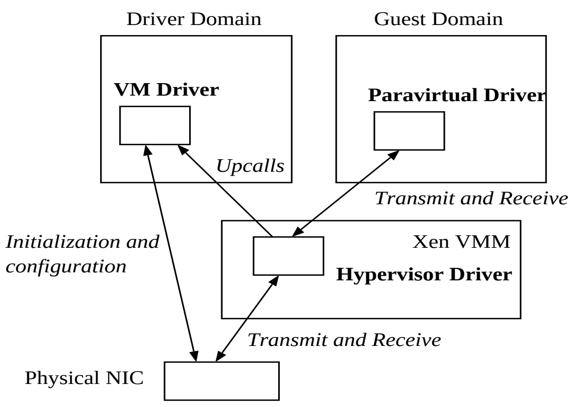

Figure 2: TwinDrivers Architecture

the same time: the original VM driver instance runs in dom0, and the derived hypervisor driver runs in the hypervisor. We first load the VM driver into the dom0 kernel where it performs the initialization of the NIC and the driver data structures. After the initialization is complete, we load the hypervisor driver into the Xen hypervisor. TwinDrivers uses this driver for performing the performance-critical send and receive operations on the NIC.

We develop a new paravirtual driver for guest domains, which interfaces with the Xen hypervisor through a hypercall interface and allows it to invoke the hypervisor driver to transmit and receive packets on its behalf. No context switch is incurred in invoking the hypervisor driver from the guest context.

The VM driver instance continues to run in dom0 to provide support for all other NIC operations which are not performance critical. These include reconfiguring NIC parameters using `ethtool`-like tools, doing periodic error checks on the NIC using timers, collecting and reporting device statistics, etc. Keeping the VM driver instance running in the driver domain for these functions allows us to restrict the hypervisor interface to the driver to just the transmit and receive functions, and avoids the need to port existing user-space tools (such as `ethtool`) to use the new hypervisor driver instead of the VM driver.

3.2 Single Instance of Driver Data Structures

In the TwinDrivers architecture, although there are two instances of the driver running, there is only a single instance of the driver data structures residing in the dom0 address space. The hypervisor driver instance accesses the shared dom0 driver data structures for all its operations.

We introduce a new mechanism called *Software Virtual Memory* (SVM) that allows the hypervisor instance to access the dom0 data structures from any guest domain address space. Software Virtual Memory is a runtime address translation and protection mechanism which is incorporated into the hypervisor driver during binary rewriting of the VM driver.

The SVM mechanism is the key to combining *efficiency* and *safety* in the hypervisor driver. By allowing the hypervisor instance to access the dom0 driver data structures from any guest domain context, expensive context switches are avoided on driver invocation, and high *performance* is achieved. By restricting all memory accesses from the hypervisor instance to the dom0 address space, *safety* is achieved and the hypervisor is protected from memory corruption bugs in the driver.

A third advantage of keeping a single copy of all data in dom0 address space is that it allows the hypervisor instance to reuse driver support routines present in the dom0 kernel. The driver support routines form a large body of code in the VM (Linux) kernel, and it requires significant engineering effort to provide identical support routines in the Xen hypervisor in order to run the dom0 drivers [10].

However, since the driver data resides in the dom0 address space, the hypervisor instance can reuse the driver support routines in the VM using an *upcall* mechanism. The *upcall* mechanism allows the hypervisor to avoid having to implement the majority of the driver support routines which are invoked only infrequently by the driver. Instead, the hypervisor implements only a small set of performance-critical support routines which are needed for good performance. The *upcall* mechanism thus allows us to reduce the *software engineering effort* needed to support the driver while retaining the performance benefits.

We now describe these mechanisms in more detail in the following sections.

4. Detailed Design

4.1 Software Virtual Memory

Software Virtual Memory (SVM) is the key mechanism that enables the hypervisor driver instance to access the driver data residing in dom0 address space. SVM uses a combination of runtime virtual address translation and page remapping to allow memory accesses to the dom0 address space from the hypervisor without a context switch, and to prevent invalid access to the hypervisor address space.

At the core of the SVM mechanism is a *Software translation table* (`stlb`) which maps from virtual memory page addresses in dom0 address space to *mapped* virtual page addresses in the hypervisor address space. The *mapped* page address in an `stlb` entry is a hypervisor page which maps to the same physical page as the corresponding dom0 page address.

To produce the hypervisor driver, every instruction which references memory locations in the original VM driver is rewritten to make use of SVM to perform the memory access (except for stack-relative memory references). Thus, at runtime, every memory access to dom0 address space from the hypervisor driver instance is first translated using the `stlb`

```
        movl    %(r_src), %r_dest
```

Figure 3: Indirect memory reference in original code

```
1.      leal    %(r_src),        %r1
2.      movl    %r1,             %r2
3.      andl    0xfffff000,      %r1
4.      movl    %r1,             %r3
5.      andl    0xfff000,        %r1
6.      shrl    $9,              %r1
7.      cmpl    stlb(%r1),       %r3
8.      jne     .L_slow_path
9.      xorl    4+stlb(%r1),     %r2
10.     movl    (%r2),           %r_dest
```

Figure 4: Rewritten code using SVM

table into a *mapped* address, and the memory access is made using the translated address. Attempts to access the hypervisor address space by the driver are detected and prevented because the stlb table does not contain valid mappings for hypervisor addresses. On such an illegal memory access by the driver, it is aborted.

Figures 3 and 4 give an example of how rewriting works, using an example (figure 3) of an indirect memory reference instruction which loads the value at the memory location in r_src into the register r_dest. Figure 4 shows how the rewritten code translates the address using the stlb table and uses the translated address to load the value. [1]

The stlb table acts as a hashtable storing translations from dom0 virtual page addresses to the *mapped* virtual page addresses in the hypervisor. In lines 1 to 6, the lower 12 bits of the dom0 page address (for a 32 bit system) are used as an index into the stlb table. In line 7, we check if the indexed stlb entry for the dom0 page is valid, i.e., if the page has been previously mapped into the hypervisor address space (and there are no hash collisions). If so, the stlb entry is used to compute the final translated address and this address is used for the memory reference (lines 9 and 10).

If the stlb translation entry for the virtual address accessed is not valid (line 8), control is transferred to a slow-path lookup routine. If the stlb lookup failed because of a hash collision, the slow-path routine looks up a hash chain and fills in the correct mapped virtual page address.

If, however, the lookup failed because the virtual address was being accessed for the first time, the slow-path routine checks the permissions of the memory access and, if the access is permitted (i.e., the memory page belongs to dom0 address space), it creates a new hypervisor mapping for the dom0 address. It allocates a new hypervisor virtual page, and maps it to the physical page corresponding to the accessed page. It then fills in the stlb table with the new translation entry. Subsequent accesses to this dom0 page are translated directly from the stlb table. [2] Entries in the stlb table are thus dynamically filled in as memory accesses to dom0 address space are made by the hypervisor driver instance. In our implementation, we use an stlb hashtable with 4096 entries, mapping up to 16MB of dom0 virtual memory.

The fast path of the SVM-based memory access replaces one memory instruction in the original code with ten instructions in the rewritten code. [3] While this may seem prohibitive at first, in practice its impact on overall performance is much smaller. Firstly, in a typical driver, only roughly 25% of the instructions reference memory, and are rewritten to use SVM (we measured this for some network drivers). Secondly, in a typical network-intensive workload, the device driver itself incurs roughly 10-15% of the total overhead. As we show in section 6, the overall performance impact of using an SVM-based device driver is quite small.

The stlb based SVM memory access is not used for stack-relative memory accesses (i.e., it is used only to translate heap memory access and not stack memory access). This is because the hypervisor driver instance uses a separate stack of its own in the hypervisor address space, and overflow on this stack is prevented by the use of guard pages.

4.2 Upcalls from the hypervisor into dom0

The hypervisor uses the upcall mechanism to reuse the driver support routines present in dom0. An upcall is a synchronous, cross-address-space function invocation and return mechanism. Upcalls are used by the hypervisor to link infrequently called support routines from the driver to the corresponding routines in the driver VM using special stub routines in the hypervisor. On a call to a stub routine by the hypervisor driver, the stub routine first saves the parameters of the call and then initiates an *upcall* into the driver domain by sending a special synchronous virtual interrupt to dom0. If the support routine is invoked while the driver is running in a guest domain context, a synchronous context switch to dom0 is done first. Additionally, before the virtual interrupt is sent to dom0, the stub routine also switches from the hypervisor stack to an ‘upcall’ stack. This is because, in the Xen hypervisor, the state of the hypervisor stack is not saved on transition to the guest domain.

An upcall handler is registered in the driver domain to receive upcall requests via synchronous virtual interrupts. It recovers the upcall parameters, sets up the stack and register parameters, and then invokes the driver support routine. On

[1] We discuss how the instruction rewriting works for more complicated x86 instructions in section 5.

[2] Actually, two consecutive dom0 pages are mapped into the hypervisor for each stlb ‘miss’. This is because the Intel instruction set permits unaligned memory accesses, so a memory access may straddle two pages.

[3] Additional scratch registers are needed for computing the address translation, which may require spilling some registers to memory, and can increase the length of the fast path. However, we avoid the cost of spilling registers most of the time by doing a register liveness analysis to determine the set of free registers available at each instruction.

return from the driver support routine, the upcall handler saves the return values of the routine and 'returns' to the stub routine via a hypercall. The stub routine eventually returns to the hypervisor driver with the support routine's return values (possibly after doing another domain switch back to the guest domain).

For the upcall mechanism to work correctly, the environment in which the driver support routine is called from the upcall handler in dom0 must be identical to the environment in which it is called from the hypervisor driver. The call environment of the routine comprises of three components: the heap, the stack and the registers. The heap environment is identical in the two invocations because there is a single driver data instance which resides in dom0 address space. The register values for the two calls are made identical by the upcall mechanism. Although the stack parameters passed are identical in the two cases, the stack address is different. This could be problematic, for instance, if the hypervisor driver passes addresses of its stack variables as parameters to the dom0 support routines. In this case, the dom0 support routine would try to dereference a hypervisor driver stack address and would cause a protection fault. One possible solution would be to use instruction emulation to trap and emulate the access from the dom0 support routine to the hypervisor driver stack, after making appropriate validity checks. In practice, since it is uncommon to pass stack variables by reference, we have not encountered the stack dereference problem for any upcall to driver support routines from network drivers. Thus, currently we have not implemented the proposed solution.

4.3 Support routines in the hypervisor

Upcalls can be expensive because they potentially involve a context switch and transition to the driver domain. To avoid the cost of an upcall on invocation of every driver support routine called by the driver, the hypervisor provides implementations of some support routines which are frequently called during the execution of performance-critical parts of the driver. For a typical driver, the set of such routines is a small fraction of the total number of support routines that are called by the driver.

For instance, table 1 lists the Linux driver support routines that are called during error-free execution of the transmit and receive routines of the Intel e1000 driver. There are only 10 such functions, compared to the 97 routines called by the e1000 driver for all its operations.

The support routines which are implemented in the hypervisor make use of the `stlb` translation table explicitly while accessing driver data in dom0 address space. For support routines that need to allocate and free memory in the dom0 heap, such as `__netdev_alloc_skb` and `dev_kfree_skb_any`, we use a preallocated pool of buffers from dom0 heap which are reserved for use by the hypervisor routines. We use a simple reference counter trick to prevent other routines in the dom0 kernel from accessing these buffers.

Routine name	*Description*
__netdev_alloc_skb	*allocate sk_buffs*
dev_kfree_skb_any	*free sk_buffs*
netif_rx	*receive network packets*
dma_map_single	*map DMA buffer*
dma_map_page	*map DMA page*
dma_unmap_single	*unmap DMA buffer*
dma_unmap_page	*unmap DMA page*
_spin_trylock	*acquire spinlock*
_spin_unlock_irqrestore	*release spinlock, restore interrupts*
eth_type_trans	*process MAC header*

Table 1: Functions called frequently from the e1000 network driver

4.4 Synchronization

Concurrent access to the shared data instance from the hypervisor and VM driver instances introduces the issue of synchronization. Fortunately, this is easily resolved. If the original driver is compiled for an SMP environment, then it already uses the correct synchronization primitives to access shared data. These synchronization operations continue to work correctly for the hypervisor driver instance since they operate on atomic synchronization variables which are also shared between the hypervisor and VM driver.

Disabling interrupts is a common synchronization mechanism used when sharing data structures between the device driver and the operating system. Since the original VM driver runs inside dom0, the dom0 kernel masks and unmasks a *virtual interrupt* flag instead of the real CPU interrupt flag, when it wants to prevent the driver interrupt handler from running. Thus, the hypervisor must respect the *virtual interrupt* flag of the dom0 kernel before invoking the interrupt handler of the hypervisor driver. This is ensured by invoking the hypervisor driver interrupt handler routine in a schedulable 'softirq' context, instead of directly in the interrupt context.

4.5 Safety of Derived Hypervisor Driver

The SVM mechanism ensures memory safety of the derived hypervisor driver. Since every heap access from the hypervisor driver is translated before the access is made, invalid accesses to the hypervisor address space, or to other domain memory, are detected and prevented by SVM.

Although the derived hypervisor driver is secure against the most common kind of driver bugs, namely memory corruption bugs, it still suffers from some safety issues that are already present in the current Xen driver domain architecture. Specifically, since the network driver has full, privileged access to the network interface, a buggy or malicious driver can set up illegal DMA transfers that allow it to read from or write to memory regions it is not allowed to access. This is a safety violation that already exists with the current Xen driver domain model, where the dom0 driver has privileged access to the NIC. A complete solution to this problem requires the use of an IOMMU that can be programmed to restrict the memory regions accessible from the network card.

There are some additional unsafe situations that are not currently handled in the TwinDrivers framework. However, these can be handled using existing mechanisms. We describe some of these below.

4.5.1 Stack Corruption

Currently, the SVM memory protection mechanism is applied only to heap accesses, and not for stack-relative accesses. This is done because the hypervisor driver does not require address translation in order to access its hypervisor stack. However, this mechanism is not sufficient to prevent stack corruption errors. For instance, a buffer overflow error in the hypervisor driver can cause the driver to return to an invalid address, which is a security violation.

The stack corruption problem can be addressed by using SVM-like mechanisms to insert checks in the hypervisor driver to ensure the safety of stack-relative memory accesses. These checks are required only for those memory accesses that cannot be statically determined to be safe. For instance, accesses to constant offsets from the stack pointer can be potentially statically verified. For the small number of variable-offset accesses from the stack pointer, additional validity checks would need to be inserted. Alternatively, the problem of control flow integrity can also be solved using techniques similar to those used in XFI [5].

4.5.2 Non-memory related errors

The TwinDrivers framework does not currently handle non-memory related errors in the hypervisor driver. For instance, if the hypervisor driver goes into a deadlock or an infinite loop, it can prevent the hypervisor from regaining control. Such resource hoarding bugs can be prevented by mechanisms similar to those used, for instance, in VINO [16]. The VINO extensible kernel makes use of timeouts to limit the duration of execution of the extension code. Similar mechanisms can be used to limit the execution time of the hypervisor driver.

Another category of bugs that is not currently handled is the use of privileged instructions, such as modifying the page tables to corrupt the system. These kinds of bugs can be detected and prevented by static inspection of the driver code during binary translation.

5. Implementation

5.1 Deriving the hypervisor driver

The hypervisor driver is created by binary rewriting of the VM driver. In the first step, we produce the assembly file of the VM driver, either by disassembling the VM driver binary, or, if the driver source is available, by directly compiling the driver into assembly. Since we work with Linux drivers, which are available in source form, we take the latter approach.

This VM driver assembler file is fed into an assembler-level rewriting tool, which generates the hypervisor assembler file as output. Conceptually, assembler-level rewriting is equivalent to binary rewriting, although working at the assembly level significantly simplifies the implementation of parsing and code generation. The hypervisor assembler file generated is eventually compiled into the hypervisor driver binary.

The rewriting tool performs a set of transformations to incorporate the SVM mechanism into the hypervisor driver. For memory reference instructions in the VM driver, the transformation applied is described in section 4.1. We now describe the transformations for other x86 instructions in the VM driver that reference memory in more complex ways.

5.1.1 String instructions

The x86 instruction set contains a number of ‘string’ instructions that can be used to perform string operations on blocks of contiguous data in memory, such as copying, string comparison, etc. Examples of such instructions include `movs`, `cmps`, `lods`, `stos`, `scas`, etc. These instructions take as operands the source and/or destination memory address, and an implicit length operand. For instance, the `rep; movs` instruction copies `ecx` bytes of data from source address `esi` to destination address `edi`.

When translating such instructions to use the SVM mechanism, it is not sufficient to simply translate the source and destination address operands. This is because the string operands of these instructions may span multiple pages, whereas the `stlb` translations for the string addresses may not necessarily map the contiguous dom0 pages containing the string to contiguous hypervisor pages.

Thus, for translating string instructions, we generate code that loops over the entire string in chunks of page length, and use the string instruction on the individual string chunks that are guaranteed to lie within a single page. Within the loop body, the regular address translation mechanism is used for the starting string source and destination addresses.

5.1.2 Indirect calls

The x86 instruction set allows routines to be ‘indirectly’ called by specifying the address of the routine as a register or memory operand. For example, the instruction ‘`call %eax`’ makes an indirect call to the routine whose address is given in the `eax` register. Since all data is shared between the hypervisor driver and the VM driver (including the values of function pointers), the address of the indirectly called routine in the hypervisor driver actually points to the routine in the VM driver.

Thus, for indirect calls, the address of the called VM driver routine is first translated to the address of the corresponding hypervisor driver routine, and then the actual call is made. Similar to the stlb table for memory addresses, an stlb_call table caches translations from VM-driver routine addresses to hypervisor-driver routine addresses. In order to translate from VM-driver routine addresses to hypervisor-driver routine addresses the first time, we need to know the correspondence between the original driver's code addresses and the translated driver's code addresses. Although this information can be generated while creating the hypervisor driver from the VM driver, overall, this approach is quite cumbersome.

We reduce the complexity of translating from the VM driver's addresses to the hypervisor driver's addresses by using the same rewritten driver for both the VM driver instance and hypervisor driver instance. For running the rewritten driver as the VM driver, the stlb table for the VM driver instance is filled with identify mappings. Thus, the VM driver instance continues to use its original data addresses and functions correctly as before, except that it runs a little slower.

Using this approach, the code addresses in the VM driver and the hypervisor driver always differ by a constant offset for all routines, and thus address translations between the two can be done in a simple manner.

5.2 Loading the hypervisor driver

The rewritten hypervisor driver is loaded into the Xen address space using a modified ELF loader.

During loading, all data references in the hypervisor instance (i.e., the driver's data symbols and the 'imported' Linux data symbols) are resolved to the corresponding symbol addresses of the driver and Linux variables in the dom0 address space. This is done with the help of the module loader in the dom0 kernel, which saves the necessary driver relocation information at the time the original driver is loaded into the dom0 kernel. This ensures that all hypervisor driver data references point only to memory locations in dom0 address space.

Hypervisor driver calls to external driver support routines are also resolved in a special way. Calls to support routines which are implemented in the hypervisor itself are resolved to the hypervisor's implementation. For other driver support routines which are not implemented by the hypervisor, the driver calls are resolved to 'stub' routines in the hypervisor. A separate stub routine is provided for each unimplemented driver support routine. The mapping between the stub routine number and the corresponding support routine in dom0 is saved by the loader. At runtime, when the stub routine is invoked, it initiates a cross-address-space call to the corresponding dom0 routine using the *upcall* mechanism described in section 4.2.

The hypervisor needs some additional information for actually invoking the transmit and interrupt handler routines in the hypervisor driver. It needs to know the driver entry points for the transmit and interrupt routines, and also some additional parameters that the driver expects to be passed on each invocation (such as a pointer to the Linux netdev structure for the transmit routine). This information is passed to the hypervisor from the dom0 process which initiates the driver loading.

5.3 Invoking the hypervisor driver

The hypervisor transmits and receives packets on behalf of the guest domains by invoking the transmit and interrupt handler routines of the hypervisor driver respectively. The guest domains interface with the hypervisor driver using a new paravirtualized network driver.

For all invocations of the hypervisor driver, all parameters passed to the driver must be valid heap addresses in dom0 address space. Thus, all packet buffers (sk_buff structures in Linux) allocated to the hypervisor driver reside in dom0 address space, and are also persistently mapped into hypervisor address space using the stlb mapping mechanism.

For transmit operations from guest domains, the hypervisor acquires a pre-allocated sk_buff in dom0 address space, copies the header of the guest packet (up to the first 96 bytes) into the sk_buff header, and chains together the rest of the guest packet using the page fragment pointers in the sk_buff (using pre-allocated page frames from dom0). It then invokes the hypervisor driver transmit routine with the dom0 sk_buff parameter.

The DMA transfers set up by the hypervisor driver work correctly because the hypervisor implementation of the DMA mapping functions, dma_map_single and dma_map_page return the correct guest machine page addresses. [4]

For receive operations, the hypervisor calls the driver interrupt routine on receiving an interrupt from the NIC. Network packets are received by the hypervisor driver into dom0 sk_buffs which are persistently mapped into the hypervisor. The hypervisor demultiplexes the received packets based on the destination MAC address, and queues the packet to the appropriate guest domain. When the guest domain is scheduled next, the hypervisor copies the packets into guest domain buffers, and raises a virtual interrupt to notify the guest domain paravirtual driver.

6. Evaluation

6.1 Experimental setup

We have implemented TwinDrivers in Xen-3.2.1 (changeset 16485) running Linux version 2.6.18.8 in dom0 and in the guest domains. We evaluate the network performance of a guest domain using TwinDrivers, comparing it with

[4] Alternatively, the hypervisor makes use of the *physical_to_machine* mapping table in the dom0 kernel to map from physical page frames of the skb in dom0 to the correct machine page frames in guest domains. This way, the DMA mapping driver functions can be even invoked using upcalls and would still work correctly.

the performance of an unoptimized Xen guest domain, Xen dom0, and native Linux. We evaluate the performance for two workloads. Our first workload uses a netperf [1] like microbenchmark to measure the transmit and receive network performance in guest domains. Our second workload consists of a web server serving concurrent HTTP requests for files from a SPECweb99 [2] like file-set.

The testbed consists of a 3.0 GHz Intel Xeon server machine equipped with five Intel Pro1000 Gigabit Ethernet cards. This machine is connected to five client machines (3.0 GHz Intel Xeon) equipped with one Intel Pro1000 Gigabit Ethernet card each.

6.2 Microbenchmark Results

The microbenchmark workload measures the maximum TCP streaming throughput achievable over a small set of TCP connections. In the experiments (both transmit and receive), the server machine is connected to each client machine using a separate TCP connection over a different NIC. The experiment measures the maximum aggregate TCP throughput (transmit or receive) the server can achieve using all five NICs.

Figures 5 and 6 show the transmit and receive performance of a Xen guest domain using TwinDrivers (“domU-twin”) and compare it with the performance achieved in an unoptimized guest domain using standard Xen networking (“domU”), the driver domain (“dom0”), and a native Linux system (“Linux”). For the domU-twin configuration, all 10 functions required for fast-path operation of the hypervisor driver (see table 1) were implemented in the hypervisor, and no upcalls were made.

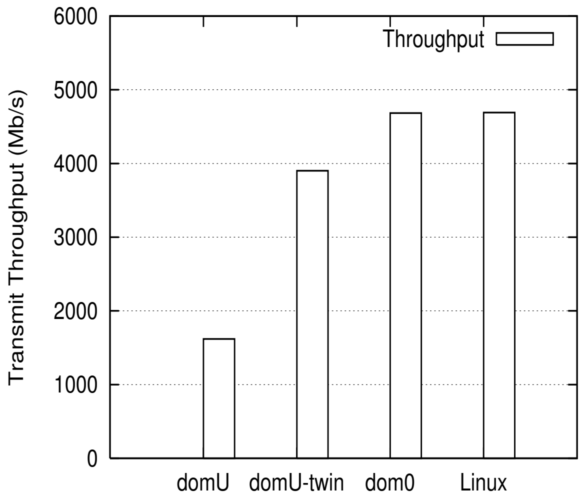

Figure 5: Transmit Performance for netperf Benchmark

For the transmit benchmark (figure 5), the native Linux system saturates all 5 NICs to achieve an aggregate throughput of 4690 Mb/s while using only 76.9% of the CPU, while Xen dom0 achieves a throughput of 4683 Mb/s with full

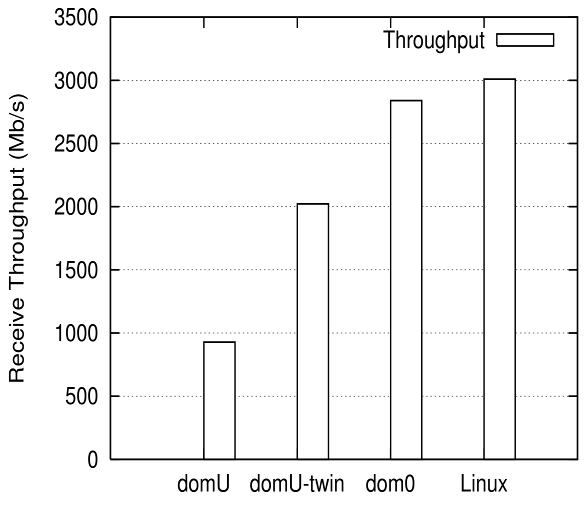

Figure 6: Receive Performance for netperf Benchmark

CPU saturation. The performance of the TwinDrivers guest domain (domU-twin) is 3902 Mb/s with full CPU saturation, which is within 83% of the dom0 performance, and is within 64% of the native Linux performance, in CPU-scaled units.

The performance of the unoptimized guest domain (domU) itself is only 1619 Mb/s at 100% CPU saturation. Thus, compared to the unoptimized guest domain, the TwinDriver guest domain achieves a performance improvement of a factor of 2.41.

For the receive benchmark (figure 6), the native Linux performance is 3010 Mb/s at full CPU saturation, and the Xen dom0 performance is 2839 Mb/s. The performance of the TwinDrivers guest domain (domU-twin) is 2022 Mb/s at full CPU saturation, which is roughly 71% of the dom0 performance, and close to 67% of the native Linux performance.

The performance of the unoptimized guest domain is only 928 Mb/s at 100% CPU saturation. Thus, the TwinDrivers guest domain improves upon the guest domain performance by a factor of 2.17.

Figure 7 shows the breakdown of packet processing overhead for the transmit workload in the four systems. This profile was obtained with the microbenchmark running only on a single Gigabit NIC. Thus, the relative numbers obtained here differ a little from the throughput results. We show the CPU overhead in terms of cycles per packet incurred in four categories: the dom0 kernel (dom0), the guest domain kernel (domU), the Xen hypervisor (Xen) and the network driver (e1000). For the native Linux case, we show the Linux kernel overhead in the dom0 kernel.

The unoptimized guest domain per-packet overhead is more than twice the overhead of the TwinDriver guest domain (21159 cycles/packet vs. 9972 cycles/packet). Most of this overhead is incurred in invoking dom0 (8394 cycles/-

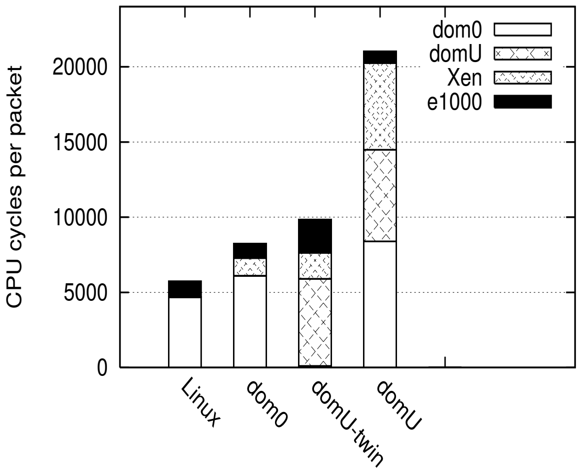

Figure 7: CPU cycles per packet for transmit workload

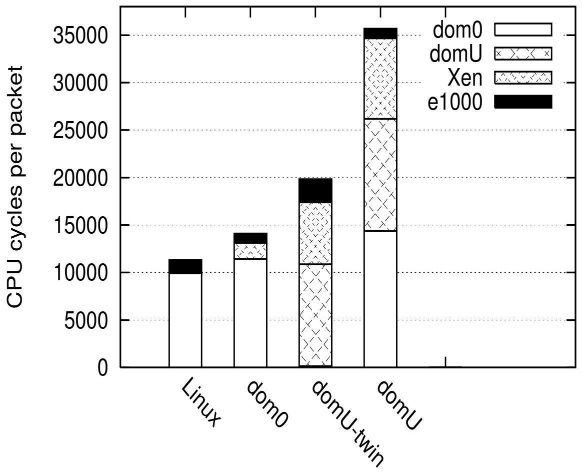

Figure 8: CPU cycles per packet for receive workload

packet) and in the additional hypervisor overhead for switching and transferring packets between the guest and driver domain [15]. The TwinDrivers guest avoids both these overheads by directly invoking the hypervisor driver.

Compared to native Linux, both the dom0 and the TwinDrivers guest incur the virtualization overhead of running on top of a hypervisor (1184 cycles/packet for dom0, and 1726 cycles/packet for domU-twin). In the TwinDrivers configuration, there is additional overhead relative to dom0 in two main areas: the overhead of running a rewritten driver instead of native driver (2218 cycles/packet vs. 960 cycles/-packet), and the additional hypervisor overhead of the hypercall interface between the paravirtual driver and the hypervisor driver. Overall, the TwinDrivers guest incurs roughly 20% higher overhead than dom0.

Figure 8 shows a similar breakdown of the overhead for the receive workload.

Here again, the unoptimized guest domain incurs almost twice the per-packet overhead as the TwinDrivers guest domain (35905 cycles/packet vs. 20089 cycles/packet), and most of this overhead is incurred in invoking dom0 (14384 cycles/packet) and in additional hypervisor overheads. By invoking the hypervisor driver directly, the TwinDrivers guest avoids most of these overheads.

Compared to dom0 and native Linux performance, the TwinDrivers per-packet overhead is quite large (20089 cycles/packet vs. 14308 and 11166 cycles/packet). Part of this can be explained as the overhead of running the rewritten driver (2445 cycles/packet vs. 972-1422 cycles/packet), but a large part of the TwinDriver receive overhead is incurred in the hypervisor itself (6514 cycles/packet). More detailed profiling shows that most of this overhead (3525 cycles/-packet) is incurred in copying the packet from the hypervisor driver to the guest domain driver.

For both the transmit and receive workloads, the overhead incurred by running a binary rewritten driver instead of a native driver is relatively small. The rewritten driver runs slower by a factor of roughly 2 to 3, but, the impact of this slowdown on the overall overhead of the guest domain is relatively low, less than 15%.

6.3 Web Server Workload

We now compare the performance of the guest domain using TwinDrivers with the original guest domain, dom0 and a native Linux system, for a web server workload. In this experiment, the server machine runs the *knot* web server, a bare-bones, lightweight web server developed as part of the Capriccio project [18]. It serves a static set of files generated from the file size distribution specified in the static content part of SPWECweb'99 [2]. Since we are only interested in the network performance, we use a file-set consisting of only a single directory. This entire file-set fits in memory and does not stress the disk I/O subsystem.

The workload for the web server is generated by running httperf [14] on a set of client machines. Requests are generated in an 'open' loop, and responses from the server are discarded if they are not received within a certain timeout.

Figure 9 compares the performance of the web server running in the guest domain using TwinDrivers ("domU-twin"), the original guest domain ("domU"), dom0 ("dom0"), and a native Linux system ("Linux"). The figure plots the aggregate throughput of responses received by all httperf clients (in Mb/s) as a function of the total connection request rate issued by the clients. In the figure, all configurations could not be tested to the same request rate because some configurations could not sustain high connection rates, and thus effectively ran at a lower connection rate even when a higher rate was requested.

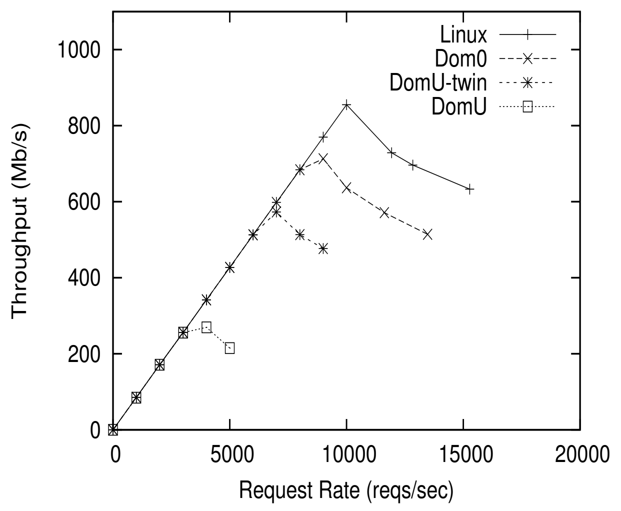

Figure 9: Web Server Workload

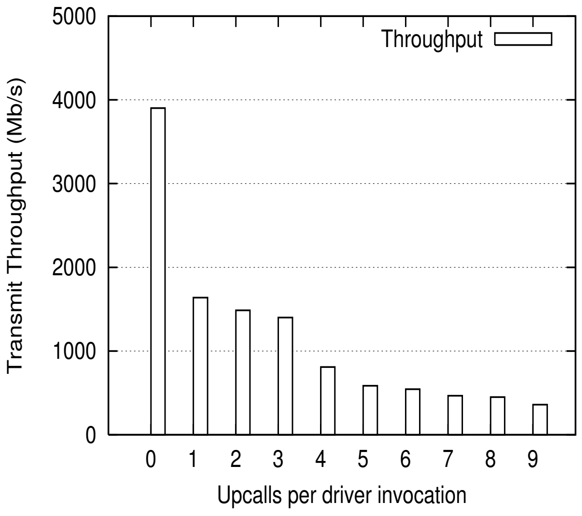

Figure 10: Transmit throughput as a function of number of upcalls

The overall trends seen in the web server workload are similar to the trends seen for the microbenchmarks. The maximum throughput achieved by the native Linux configuration is is 855 Mb/s. dom0 achieves a peak throughput of 712 Mb/s. The original Xen guest domain achieves a peak throughput of 269 Mb/s, which is only 31% of the native Linux performance. The TwinDrivers guest domain achieves a peak throughput of 572 Mb/s, which is a more than factor of 2 improvement over the unoptimized guest domain performance, and is roughly within 67% of native Linux performance.

6.4 Cost of Upcalls

To achieve good performance in the hypervisor driver, upcalls to driver support routines must be avoided during the performance critical parts of the driver. Table 1 shows that for the Intel e1000 driver, there are 10 driver support routines that are called on the fast path. Figure 10 shows how the transmit performance of a TwinDrivers guest domain drops when not all the necessary upcalls are implemented in the hypervisor.

The X axis shows the number of performance-critical support routines for which the hypervisor has to make an upcall. When no upcalls are made (first bar), transmit performance is 3902 Mb/s. As soon as the hypervisor has to make even one upcall per driver invocation, the performance drops to 1638 Mb/s (second bar). The performance drops progressively as more and more upcalls are needed, until finally it drops to 359 Mb/s when all but the network receive function (`netif_rx` in Linux) are implemented as upcalls.

6.5 Engineering Effort

The 10 driver support routines listed in table 1 were implemented in the Xen hypervisor. The entire implementation took 851 lines of commented C code and header files. This is a very small development effort compared to the effort that would be needed to support the entire driver support interface.

7. Related Work

Device drivers have received an enormous amount of interest from the research community, and space constraints limit us to only a sampling of the most related work.

As mentioned in the introduction, our work builds on the notion of running device drivers in a virtual machine [7, 10], but we go beyond that work in allowing safe execution of the performance-critical parts of the driver in the hypervisor. The idea of executing device drivers in user-level processes [9] is similar, and we project that it can benefit from our techniques as well.

The VMware ESX server [20] runs selected drivers in the hypervisor by porting the drivers and their support routines to the VMM. Not only does this involve significant development effort, it also leaves the hypervisor vulnerable to bugs in the driver. Our work ensures that the rewritten hypervisor driver executes safely in the hypervisor. Additionally, since the number of driver support routines needed to run the error-free performance-critical path in the driver is very small, the software development costs of our approach are significantly smaller.

Reusing drivers developed for one environment in a new environment is a difficult task. In the Flux OSkit [6], driver sources are ported from the original OS into a new OS by reimplementing the entire driver-kernel API support library. The kernel driver support library is a large and poorly documented body of code. Reimplementing this library requires a deep understanding of the internals of the original OS, and can be a a source of subtle bugs. The issues arising from the semantic differences between the old and new OS environments are discussed in detail in [10].

Numerous attempts have been made to reduce the vulnerability of the kernel to device driver crashes, without going all the way to running the device driver in a virtual machine [17, 19]. These approaches typically constrain memory accesses by the drivers to prevent wild writes that corrupt the kernel data structures, either by erecting address space barriers or by checking memory accesses. In our approach, we avoid any vulnerability of this nature as a by-product of leaving all the driver data structures in the VM address space and not allowing the driver any access to the hypervisor data structures.

A number of research efforts have looked at mechanisms to safely extend operating system functionality with third-party extensions [4, 16]. The SPIN extensible kernel [4] guarantees safe execution of extension code by requiring that the kernel and all extensions be written in a type-safe language (Modula-3). In contrast, the TwinDrivers approach does not requires extensions to be written in any particular language, and works with existing compiled driver binaries. The VINO extensible kernel [16] uses software fault isolation [19] as its safety mechanism. It does not require a translation mechanism such as SVM, because the extension executes in the same address space as the kernel. In contrast, TwinDrivers uses SVM to implement both a protection and a translation mechanism, and this is required because the device driver data is located in an address space that is different from the hypervisor address space.

We borrow from the Microdrivers project [8] the idea of running performance-critical parts of the driver in the kernel/hypervisor and other parts in user-space processes or in a virtual machine. Many differences, however, exist between the two approaches. First, our hypervisor instance cannot corrupt the hypervisor data structures, while the part of the Microdrivers that runs in the kernel has the potential of crashing the kernel. Methods like those used in SFI [19] or Nooks [17] have to be used to reduce this vulnerability, potentially leading to extra performance overhead. Second, the Microdrivers approach requires manual annotations for all kernel and driver data structures that can be shared between the user-space and kernel-space driver. These annotations are necessary because Microdrivers use explicit data marshaling to keep the (separate) data structures of the kernel and user driver consistent with each other. In contrast, we use a single copy of all data structures mapped at different virtual addresses in the hypervisor and the VM, providing us trivially with consistency and obviating the need for marshaling and annotations. Thus, no driver-specific knowledge or engineering effort is required in out approach; our framework works with unmodified binary drivers. Third, unlike Microdrivers we do not *split* the driver. Both instances of our driver are complete, and we can choose what instance to use for what aspects of the functionality of the driver. This allows us, for instance, to leave all of the support functionality for the error handling code in the transmit and receive parts of the network driver out of the hypervisor.

A number of recent efforts have focused on ways to improve networking performance in virtual machines, using either software [11, 13, 15], or hardware [21] mechanisms. The software techniques proposed include using packet aggregation techniques [11, 13], interrupt coalescing [15], etc, to reduce per-packet processing overheads in the Xen network I/O architecture. We believe these techniques are complementary to our approach, and can be used in the TwinDrivers architecture to yield additional benefits. Hardware techniques [21] involve using virtualization-aware network interfaces which can be directly accessed from guest domains, yielding better performance and scalability. While the hardware approach offers improved performance, it does so at the cost of tying down the guest VM to a specific NIC, and does not address the safety issues associated with running drivers in the hypervisor.

8. Conclusions

We presented TwinDrivers, a framework which allows us to create safe and efficient hypervisor drivers from guest OS drivers. The derived drivers run directly in the hypervisor and execute the performance-critical operations of the device on behalf of guest domains, such as transmitting and receiving packets for network cards.

TwinDrivers uses binary rewriting to ensure memory safety and efficiency in the hypervisor driver. The Software Virtual Memory mechanism allows the hypervisor driver to efficiently access its data in the VM address space, while protecting the hypervisor address space from memory access from the driver. The upcall mechanism allows the hypervisor to implement only a small set of performance-critical support routines to run the driver, reducing the software engineering costs.

We used TwinDrivers in the Xen virtual machine environment to create a hypervisor driver which can be directly invoked by Xen guest domains. Using the hypervisor driver improves the Xen guest domain networking performance by a factor of 2.4 for transmit workloads, and by a factor of 2.1 for receive workloads. The resulting transmit performance is within 64% of native Linux performance, and the receive performance is within 67% of Linux performance.

References

[1] The netperf benchmark. `http://www.netperf.org/netperf/NetperfPage.html`.

[2] Specweb'99 benchmark. `http://spec.org/web99`.

[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 *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[4] Brian N. Bershad, Stefan Savage, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN operating System. In *Symposium on Operating System Principles (SOSP)*, 1995.

[5] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *16th ACM Symposium on Operating System Principles (SOSP'97)*, Saint-Malo, France, October 1997.

[7] 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 (OASIS)*, Oct 2004.

[8] Vinod Ganapathy, Matthew Renzelmann, Arini Balakrishnan, Michael Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *13th Internantional Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, Seattle, WA, March 2008.

[9] Jorrit N. Herder, Herbert Boss, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *ACM SIGOPS Operating System Review*, 40(3):80–89, July 2006.

[10] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gotz. Unmodified Driver Reuse and Improved System Dependability via Virtual Machines. In *Operating Systems Design and Implementation (OSDI'04)*, San Francisco, CA, December 2004.

[11] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In *USENIX Annual Technical Conference*, Boston, MA, June 2006.

[12] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, Chicago, USA, June 2005.

[13] Aravind Menon and Willy Zwaenepoel. Optimizing TCP Receive Performance. In *USENIX Annual Technical Conference*, Boston, MA, June 2008.

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

[15] Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Ian Pratt. Bridging the gap between hardware and software techniques for i/o virtualization. In *USENIX Annual Technical Conference*, 2008.

[16] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Operating Systems Design and Implementation (OSDI)*, 1996.

[17] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *19th ACM Symposium on Operating System Principles (SOSP'03)*, Bolton Landing, NY, October 2003.

[18] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[19] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP'93*, 1993.

[20] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI'02*, 2002.

[21] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *HPCA*, 2007.