Bakkalaureatsarbeit

DDEKit Approach for Linux User Space Drivers

Hannes Weisbach

28th February 2011

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Björn Döbel

Danksagung

Ich danke meiner Familie für die aufgebrachte Geduld und fortdauernde Unterstützung.

Meinen Freunden danke ich für anregende Diskussionen, unermüdliches, kurzfristiges Korrekturlesen und für gelegentlich kostenloses Essen, besonders Nancy, Thomas, David und Anja.

Für unermüdliche Zerstreuung, Prokrastination, Tipps, interessante Gespräche und den ein oder anderen Kaffee danke ich meinen Kommilitonen im Studentenlabor.

Bei Professor Härtig möchte ich mich für die Möglichkeit bedanken meine Abschlussarbeit am Betriebssysteme Lehrstuhl zu verfassen. Ebenso bedanke ich mich bei allen Mitarbeitern am Lehrstuhl, die zum Entstehen dieser Arbeit beigetragen haben; insbesondere meinem Betreuer, Björn, danke ich für die unentwegte Motivation.

I almost wish I hadn't
gone down that rabbit-hole
– and yet – and yet –
it's rather curious, you know,
this sort of life!
-Alice

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 28. Februar 2011

Hannes Weisbach

Aufgabenstellung

Die Ausführung von Gerätetreibern als Nutzerapplikation führt zu einer verbesserten Robustheit des Systems durch Isolation der kritischen Komponenten. Das DDEKit stellt eine Abstraktion bereit, auf der Gerätetreiber implementiert und betriebssystemspezifische Treiber-Frameworks implementiert werden können, welche die Verwendung existierender Treiber in einem neuen Betriebssystem ermöglichen.

Ziel der Aufgabe ist es, den DDEKit-Ansatz auf die in Linux verfügbaren Schnittstellen abzubilden, um auf diese Weise Linux-Kern-Treiber als Nutzer-Anwendung ausführen zu können. Hierzu sollen geeignete Interfaces (bspw. Linux/UIO) auf ihre Tauglichkeit untersucht werden und die Umsetzung des DDE-Ansatzes für eine ausreichend komplexe Geräte-Klasse (z.B. Netzwerk- oder Blockgeräte) demonstriert werden.

Contents

List of Figures

List of Tables

1 Introduction

Device drivers, as integral part of every operating system, have been identified as a major source of errors in modern day operating systems [SBL03, CYC+01, SMLE02]. Because device drivers are executed in kernel mode, a bug can compromise other parts of the system or cause catastrophic system failure.

As a consequence, various mechanisms were developed to execute device drivers safely [GRB+08, BWZ10, LCFD+05, SMLE02]. They all employ virtual memory protection and lowering the privilege level of the driver or at least parts of the driver [GRB+08]. Running device drivers in user space does not only provide isolation, but also reduces development effort because sophisticated development tools for debugging and performance optimisation are at hand. Type safe languages can be used to prevent type errors in device drivers [BSP+95, Rit97]. If an error occurs and the user space driver crashes, it can be restarted easily, because it is an ordinary user space process.

However, user space drivers need access to the hardware they are controlling. Therefore dedicated interfaces are introduced to allow the user space driver to interact with the hardware. Such interfaces are usually accompanied by performance degradation or increased CPU utilisation [SMLE02, LU04].

Device drivers also comprise much of the code of operating systems [CYC+01]. Thus, operating system projects face a considerable effort in implementing and debugging device drivers. Alternatively, device drivers can be re-used from existing operating systems. The DDEKit/DDE [Hel01] approach was developed to permit the reuse of drivers from commodity operating systems, such as Linux. DDEKit also offers portability by decoupling the functionality of a user space driver from the operating system.

In this thesis, I present the implementation of a DDEKit for the Linux operating system. DDEKit-Linux enables the re-use of device drivers in Linux, although an extensive number of drivers are available for Linux. DDEKit-Linux increases the stability of device drivers by running them as Linux user space processes, thus isolating the device driver from the rest of the system. Additionally, DDEKit-Linux enforces resource restrictions. DDEKit-Linux can also be used to optimise or analyse device drivers in user space. An optimised or debugged version of the device driver can then be migrated back into kernel space, because no modifications to the device driver are necessary.

The rest of this document is structured as follows. The next chapter presents the necessary basic knowledge to understand this document. I introduce DDE, present selected aspects of the PCI bus as well as the Linux UIO framework. A literature review concludes the second chapter.

In Chapter 3, I present the design and implementation of DDEKit-Linux. I discuss problems occurred and possible solutions in detail.

I examine various performance aspects of DDEKit-Linux in Chapter 4 and conclude my work in Chapter 5 with a summary of future improvements.

2 Background and Related Work

In this chapter, I provide some basic information related to the understanding of my work. At first, I introduce the Device Driver Environment (DDE), followed by a brief overview of PCI and the Linux UIO driver framework. An overview over related work will conclude this chapter.

2.1 Device Driver Environment

When running legacy device drivers in user space, the challenge is to adapt the execution environment or application programming interface (API) from the host operating system to the execution environment expected by the device driver. The necessary adaption layer was first implemented for the Fluke micro kernel [FFH+96] and called "Device Driver Framework" [Mar99]. For DROPS, the **D**resden **R**ealtime **Op**erating **S**ystem (DROPS) [DRO], the "Device Driver Environment" (DDE) [Hel01] adapts the DROPS-interface to Linux device drivers.

A multitude of research, special purpose, and commodity operating systems is available, each of which can be a host or guest for a DDE. Each combination of host and guest operating system requires its own DDE-implementation, resulting in a large number of DDEs. To achieve higher portability by decoupling the DDE from its host operating system, DDEKit [Fri06] was introduced. While the DDE implements the API expected by the device driver, DDEKit abstracts primitives from the operating system and makes them available to the DDE (Fig. 2.1). The DDE uses only the DDE–DDEKit-interface and thus is independent from the host operating system.

Instead of implementing a DDE for each host–guest-combination, a DDEKit is implemented for every host [Gen, HUR] and a DDE for every guest operating system [Hel01, Fri06].

Figure 2.1: The DDE provides the API expected by the device driver. DDEKit decouples the DDE from the host OS.

2.2 PCI Local Bus

The Peripheral Component Interconnect Local Bus (PCI Bus or PCI, now Conventional PCI) is a bus in computer systems. Introduced in 1992, PCI eventually became the standard bus for attaching hardware in computer systems.

2.2.1 PCI Bus structure

The PCI bus interconnects PCI devices and independent PCI buses are connected by *PCI bridge*s, thus creating a hierarchical structure (Fig. 2.2). Three types of bridges exist: *PCI-to-PCI bridge*s, *host bus bridge*s, and *expansion bus bridge*s. The hierarchical structure imposes an addressing scheme, consisting of the *bus* number, the *device* number, and the *function* number.

According to the PCI specification, each device must implement at least one function and may implement up to eight independent functions. PCI devices implementing two to eight functions are called *multi-function* devices, as opposed to *single-function* devices.

Each function is uniquely identified by the combination of its bus, device, and function number, also known as *BDF-address*.

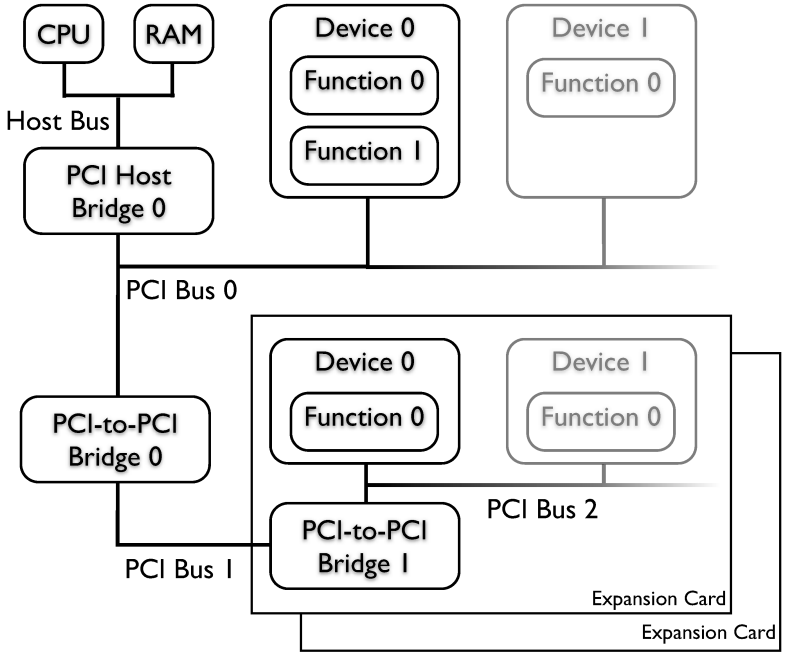


Figure 2.2: PCI Bus structure

2.2.2 PCI Address spaces

The PCI specification defines three distinct address spaces. The *Memory* and *I/O Address space*s are implementation-specific, whereas the *PCI Configuration space*'s struc-

Device ID		Vendor ID		0x00
Status		Command		0x04
Class Code			Rev. ID	0x08
BIST	Header Type	Latency Timer	Cache Line Size	0x0c
Base Address Register 0				0x10
Base Address Register 1				0x14
Base Address Register 2				0x18
Base Address Register 3				0x1c
Base Address Register 4				0x20
Base Address Register 5				0x24
Cardbus CIS Pointer				0x28
Subsystem ID		Subsystem Vendor ID		0x2c
Expansion ROM Base Address				0x30

Figure 2.3: Type 0 (non-bridge device) configuration space header layout.

ture is imposed by the PCI specification. Each function of a PCI device has its own configuration space of 256 bytes (Fig. 2.3). At system boot, configuration software scans for devices on a bus by reading each slot's *Vendor ID*. If a device, and thus a function, is present a valid *Vendor ID* will be returned. Otherwise, a PCI bridge will report an invalid *Vendor ID* (`0xffff`) to indicate a non-existent device.

The *Configuration space* contains six *Base Address register*s (BARs). Each PCI device can request up to six address ranges in *Memory* or *I/O Space* using its BARs. Configuration software determines the type and size of an address range by writing all-ones to a BAR and reading the value back. A device returns zero in all unused address bits, thereby virtually[1] specifying the size of the required address space. To request 4 kiB of memory address space, the device would return the top 20 bits as ones, the lower 12 as zero[2]. After constructing an address map, configuration software assigns the locations of the requested address ranges to each device by writing the *base address*es in the respective BARs.

Two registers from the configuration space are of particular importance: the *Command Register* and the *Status Register*. Both are 16 bits wide. The Command register is used to control the behaviour of a device on the PCI bus. Writing `0x00` to the Command register logically disconnects the device from the PCI bus. The Status register holds information regarding PCI bus related events.

[1] Size calculation requires masking the type information, inverting the value, and adding 1.
[2] The lowest 4 bits carry additional information, unrelated to the requested size.

2.2.3 Interrupts

Interrupt occurrences can generally be signalled in two ways: level-triggered and edge-triggered. Level-triggered interrupts carry the information in the state of a signal, whereas the change of a signal is used for edge-triggered interrupts.

Conventional PCI uses level-triggered interrupts, because it simplifies interrupt sharing and increases the robustness. Edge-triggered interrupts are not as robust as level-triggered interrupts because state transitions are easier to miss, whereas the state of a level-triggered interrupt persists. A shared interrupt line remains asserted, until the interrupt is serviced, thus simplifying shared interrupt handling.

Conventional PCI includes four interrupt lines. Every interrupt can be shared between different devices. If an interrupt occurs, the operating system calls every interrupt handler registered for that particular interrupt line until the interrupt is served and the line is released.

From PCI 2.2 (1998) onwards, message signalled interrupts (MSI) were supported. PCI Express (PCIe), introduced in 2003, uses message signalled interrupts exclusively. MSIs are edge-triggered, because the interrupt is delivered by writing to a specific memory location. Instead of out-of-band signalling on dedicated interrupt lines, MSIs are signalled in-band, as bus transactions. Thus, MSIs avoid synchronisation issues between bus transactions and interrupts. Moreover, MSIs alleviate the scarcity of interrupt lines.

2.2.4 Changes introduced with Conventional PCI Revision 2.3

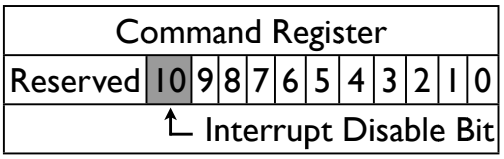
In 2002 the Conventional PCI Specification Revision 2.3 [PCI02] was released. Changes were made to the Command and Status register, as shown in Figure 2.4. Important to note is the introduction of the *Interrupt Disable* Bit in the Command register and the *Interrupt Status* Bit in the Status register.

The *Interrupt Status* Bit reflects the interrupt status of the device whereas the *Interrupt Disable* Bit controls the assertion of the interrupt line. In combination the two bits allow generic interrupt handling and form the basic building block for a device-independent driver based on Linux UIO (Section 2.3).

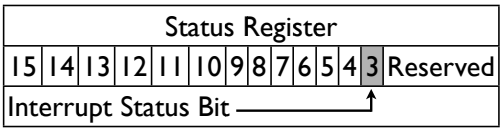
2.2.5 DMA and IOMMU

Direct memory access (DMA) is a feature in computer systems that enables hardware to access the system memory independent of the processor. Without DMA, data has to be transferred using programmed I/O or *load* and *store* instructions. When using DMA, the processor is free to perform work during transfer, in contrast to being fully occupied otherwise.

All transactions on a PCI bus are initiated by a bus master. The destination device of a transaction is called the *target*. Exactly one device may be bus master at any given time; the PCI bus controller arbitrates simultaneous bus master requests. If a device accepts a transaction from a bus master, it *claims* the transaction.



(a) The *Interrupt Disable Bit* in the Command register.



(b) The *Interrupt Status Bit* in the Status register.

Figure 2.4: The new bits, introduced by the PCI Specification Revision 2.3 [PCI02]

The PCI bus implements DMA by allowing a device to become *bus master*. Once the component become bus master, it issues read and write commands on the PCI bus. The PCI controller claims these transactions and translates them to memory accesses.

An *input/output memory management unit* (IOMMU) [AMD, Int] translates memory addresses from an I/O bus to physical addresses in system memory. The operation is similar to a conventional MMU translating between virtual and physical memory addresses (Fig 2.5). The processor usually uses virtual addresses which are translated by the MMU to physical addresses prior to memory access. A device uses *bus* or *DMA addresses* to access the system memory. In the absence of an IOMMU, bus addresses correspond directly to physical addresses. If an IOMMU is available, bus addresses have to be translated to physical addresses prior to a DMA-transfer.

The main advantage of IOMMUs is memory protection. A device is no longer able to read or write the system memory arbitrarily, thereby corrupting the computer system.

Moreover, an IOMMU simplifies the design of user space drivers: buffers intended for DMA-transfer do not need to be physically contiguous anymore. The IOMMU is able to map fragmented, physical addresses to a coherent area in the bus address space.

2.2.6 Representation of the PCI bus in Linux user space by the sysfs virtual file system

Sysfs [Moc05] is an in-memory file system provided by the Linux 2.6 kernel. *Sysfs* exposes device information and attributes to user space. It provides an information channel from kernel to user space, allowing for device configuration.

The PCI-subsystem is represented in the `bus`–directory of *sysfs* by the `pci`–directory. In the `pci`–directory, the `devices`–subdirectory contains a flat listing of all discovered PCI devices. Each device is represented by a directory, containing various attributes of a device. These attributes not only provide device configuration information such as I/O ports and memory mapped I/O regions, but also offer access to said resources.

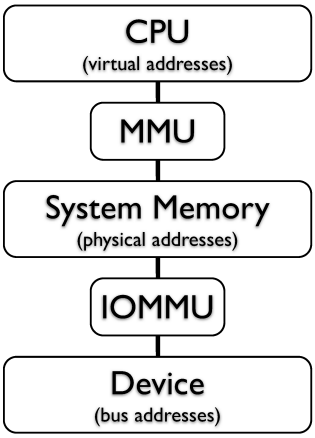


Figure 2.5: Isolation of the system memory by the MMU and IOMMU from the CPU and devices, respectively.

Like most attributes in *sysfs*, the `resource`-attribute is an ASCII encoded file. It contains a line-based representation of the I/O regions used by the device. Consider Figure 2.6 for an example: every line defines an address range in *Memory* or *I/O Address space*. A definition consists of the first address, the last address, and properties. The address space of the mapping is encoded as flags. Along with every non-zero line in

```
cat /sys/bus/pci/devices/0000:01:00.0/resource
0x0000000090200000  0x000000009021ffff  0x0000000000020200
0x0000000090100000  0x000000009021fffff  0x0000000000020200
0x0000000000002000  0x000000000000201f  0x0000000000020101
0x0000000000000000  0x0000000000000000  0x0000000000000000
```

Figure 2.6: PCI resources for an exemplary device (Intel 82573E Gigabit Ethernet Controller). The first two lines designate memory mappable I/O regions, whereas the 3rd line informs about the I/O ports used. Trailing zero-lines are truncated for brevity.

the `resource`-file, a `resourceN`-file exists, where `N` is the number of the line in the `resource`-file, starting with `0`. This `resourceN`-file offers user space access to the I/O resource represented by that file.

However, the access method differs depending on the resource type. After opening a file representing memory mapped I/O, the `mmap` system call is used to map the memory region in the address space of the calling process. Upon successful return, the I/O memory region can be accessed at the virtual address returned by the `mmap` system call.

I/O space resources cannot be mapped into an address space, because special instructions are used to access I/O ports on the x86 architecture. Instead, after opening the I/O space resource file, it can be read from and written to, using the desired I/O port

number as (file-)offset. For convenience the `pread` and `pwrite` system calls can be used. *Sysfs* conducts thorough range checks on the supplied I/O port argument before an I/O port access is executed.

Sysfs provides access to the PCI configuration space via the `config`-attribute in a device directory. *Libpci* [Mar] is a portable library providing access to the PCI configuration space on a variety of operating systems.

2.3 Linux UIO Framework

The Linux user space I/O (UIO) framework [UIO07] is primarily intended as a driver framework for devices which do not need a fully-fledged kernel module: the device may be a prototype or it does not need any of the resources the Linux kernel provides. Instead, UIO permits the use of regular user space development tools and libraries for a device driver.

The UIO framework is a skeleton driver providing facilities to handle interrupts and use PCI resources from user space. The user space interfaces exclusively with the UIO-core.

An additional driver module is needed to interface with the device. The UIO driver modules implement device-setup, a device-specific interrupt handler, and inform the UIO core of PCI resources used by the device (Fig 2.7).

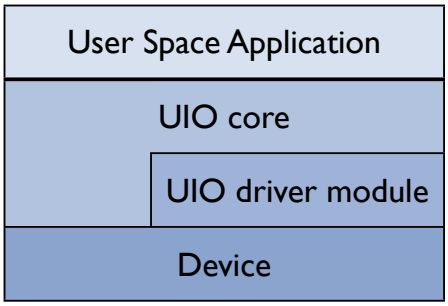


Figure 2.7: Linux UIO architecture

2.3.1 UIO interrupt handling

UIO requires an interrupt handler to provide slightly different functionality than an interrupt handler of an in-kernel driver. While a normal interrupt handler will directly service the device or setup deferred servicing of the device, a UIO interrupt handler usually only needs to acknowledge the interrupt. If shared interrupts are used, querying the device is necessary to determine whether it is the interrupt source. Instead of acknowledging an interrupt, it is also possible to mask the interrupt until it is handled in user space. Acknowledging or masking is required to avoid interrupt storms caused by level-triggered interrupts. Furthermore no new interrupts should be generated until a previous interrupt is handled by the user space driver.

By registering the interrupt handler from the supplementary module directly, the UIO core would have no information about interrupts generated by the device. To mitigate this, the UIO framework installs a wrapper interrupt handler, which in turn calls the device-specific interrupt handler. The device-specific interrupt handler signals the UIO core, whether the device was the interrupt source or not, using the return values `IRQ_HANDLED` or `IRQ_NONE`. The UIO core uses a counter to keep track of the number of occurred interrupt events. The event is subsequently signalled to user space, where the interrupt is processed. If interrupt masking is used, the user space is required to unmask the interrupt, before waiting for the next interrupt event. Because the user space interrupt handler cannot determine whether the kernel space interrupt handler uses acknowledging or masking, both interrupt handlers have to be designed to work together.

The user space application waits for an interrupt by reading a device file, which is created when a device is bound to the UIO driver. Using `udev`, the first UIO device is called `/dev/uio0`; each additional device will be assigned another number incrementally. Opening and reading this file returns an integer value representing the number of interrupts triggered by the device so far. If no interrupt occurred since the last read, the operation will block until an interrupt is received, though non-blocking operation is also supported. Returning the number of occurred interrupts allows for the detection of missed interrupts, by comparing the event counts of two subsequent read operations. An interrupt was missed if two consecutive event counts differ by more than one. Upon return from the (blocked) read-operation interrupt handling can proceed in user space.

2.3.2 PCI resources

Each PCI device may use up to six I/O resources. A resource may be mapped in *Memory* or *I/O address space*. The UIO framework provides user space access to *memory mapped* resources of a device. The UIO driver passes `struct uio_mem` data structures to the UIO core. For each I/O resource, such a data structure has to be supplied at compile-time. A user space driver can access *memory mapped* resources by calling `mmap`, using the file descriptor of a `/dev/uio` device file. If a device has multiple resources, the `offset` argument of `mmap` is used to specify a particular one. To map the n^{th} resource, n is multiplied by the architecture's page size and the result is passed as the `offset` parameter.

Although `struct uio_mem` data structure can represent resources in *I/O address space*, UIO does not offer any possibility to access this resource type. The representation of *I/O address*es by UIO is purely informational. UIO exports all information of I/O resources to user space via *sysfs*, independent of the PCI bus representation of *sysfs*.

The inconvenient interface to PCI resources is one reason DDEKit-Linux accesses PCI resources using the *sysfs*-facilities, described in Section 2.2.6.

2.3.3 The `uio_pci_generic`-module

The changes introduced with PCI 2.3, described in Section 2.2.4, allow a more generic interrupt handling in UIO. Using these new features, the `uio_pci_generic`-module can serve as in-kernel UIO module for every PCI 2.3 and later compliant device, effectively eliminating the need for a custom, in-kernel module. The *Interrupt Status Bit*, located in the Status register, can be queried to determine if a device generated the interrupt. Although the interrupt cannot be acknowledged, the *Interrupt Disable Bit*, located in the Command register, can be used to mask further interrupts from a device. After handling the interrupt in user space, the *Interrupt Disable Bit* can be reset, allowing the device to signal further interrupts. As a result, no additional kernel code needs to be written, thus moving driver development entirely to user space.

A device needs a suitable driver to work. The *Device* and *Vendor ID* fields from the *PCI configuration space* distinguish PCI devices. A list of *Device* and *Vendor ID*s is statically linked into a driver's code. Linux compares the IDs of a device to the IDs in the driver's list to find a suitable driver for the device.

In case PCI IDs are unknown at compile-time of the driver, Linux offers a facility to add new IDs dynamically to the driver. *Sysfs* exposes an interface to add new ID pairs from user space. Writing an ID pair to the file `/sys/bus/pci/drivers/driver/new_id` adds the pair to the dynamic ID list of that driver.

The `uio_pci_generic`-module's list of IDs is empty, because it is a generic driver. The *Device* and *Vendor ID* of the device, which needs to be bound to the `uio_pci_generic`-driver, have to be supplied to the driver, as described above.

PCI resources used by a device cannot be handled easily with a generic driver such as `uio_pci_generic`. Each device uses different resources and the resources are not known in advance – if at all. Hence, `uio_pci_generic` does not use the facilities provided by UIO to handle *memory mapped* and *I/O space* resources. As a consequence, DDEKit-Linux uses other means to access PCI resources, as I describe in Section 3.2.

2.4 Related Work

Device drivers have been identified as source of a considerable number of errors in operating systems [CYC+01, SBL03]. Different approaches have been explored to reduce or eliminate the effect of failing device drivers on operating systems. This includes running unmodified drivers [Kan10, BWZ10] as well as modified or newly written drivers [GRB+08, LCFD+05, EG04] in user space. Unmodified drivers have the advantage of low maintenance at the cost of computational overhead for emulating the kernel environment in user space. Microdrivers [GRB+08] divide the code into a high-speed data path, running in the kernel and (slower) management routines running in user space. This combines the advantages of fault isolation in management and error handling code with the achievable high performance of in-kernel data handling. DDEKit-Linux requires no modification of the driver's source code, thus offering low maintenance effort but also suffering a performance penalty.

Running unmodified Linux device drivers in a virtualised environment diminishes performance considerably [LU04], but provides stronger isolation to the extent of allowing

potentially malicious drivers to run, as shown by the SUD framework [BWZ10]. Relying on hardware support from an IOMMU and a PCI Express bus, the SUD framework runs unmodified Linux device drivers in a User-mode Linux process [UML]. The IOMMU protects the system from arbitrary DMA transactions, issued by a malicious driver, whereas the transaction filtering capabilities of the PCI-Express bus are used to isolate the devices from one another. SUD and DDEKit-Linux show great similarity in certain mechanisms used, for example access to PCI devices. SUD, however, offers stronger isolation and protection guarantees, which DDEKit-Linux does not support, because it was not the focus of development. In Section 5 I discuss the adaption of the isolation properties of SUD to DDEKit-Linux.

The NetBSD Runnable Userspace Meta Programs (rump) [Kan10] approach runs unmodified kernel code in user space. The kernel code is split into functional entities, called *components*, which allows a fine-grained control over the used code. Up until now, only pseudo-devices (like the Berkley Packet Filter or RAIDframe) and USB devices are supported. Rump does not yet offer the possibility to control PCI devices from user space, which DDEKit-Linux does. In contrast to DDEKit-Linux, rump exports a driver interface, allowing every application easy use of the rump-driver's services. DDEKit-Linux uses co-location i.e., it is linked directly in the application using the user space driver. A network card with DDEKit-Linux as user space driver cannot be used easily by multiple applications. However, one could envision the user space application to be a proxy to a kernel space interface, similar to the rump approach. In the kernel, device driver stubs would forward requests via the proxy-interface to the user space driver and vice versa.

A user space device driver framework based on Linux is presented in [LCFD+05]. It was shown, that user space device drivers could achieve performance similar to that of conventional, in-kernel drivers in terms of throughput. However, a small overhead in CPU utilisation is incurred. The proposed interface does not allow the reuse of already available device drivers, which is a key feature of DDEKit. Also a new system call was introduced to implement the required interfaces. This hinders spreading of this particular framework, because a new system call may not be adapted to mainstream Linux. Therefore, it is required to patch and compile kernel sources. DDEKit-Linux implements its user space – kernel space interface in modules. There is no need for patching or compiling a kernel, moreover DDEKit-Linux can be used on Linux distributions like Ubuntu, which do not necessarily provide a kernel tree at all, because modules can be compiled out-of-tree.

3 Design & Implementation

In this section, I discuss the problems and difficulties that arise when running device drivers in Linux user space. I present possible solutions and the chosen implementation for DDEKit-Linux.

Almost all of the Linux kernel code can be run in user space without modification, because it does not contain privileged instructions. This also holds true for device drivers. Nevertheless some mechanisms and resources are only available in the kernel. If drivers are run in user space, access to those resources has to be provided.

To use PCI devices from user space it does not suffice to provide means of interrupt handling. Access to PCI configuration space as well as access to memory mapped I/O regions and I/O ports is necessary. I describe how these resources are accessed by DDEKit as well.

3.1 IRQs

In Linux, a device driver requests notification of a given (via a device structure) interrupt by registering a callback function, called the interrupt handler, using `request_irq()`. The interrupt handler clears the interrupt and schedules deferred interrupt handling, if necessary [CRKH05, Ch. 10].

With the interrupt handler running in user space, two problems present themselves: first, a new, in-kernel interrupt handler is required. Secondly, interrupts need to be relayed to user space.

The new interrupt handler has to work with different devices without modification. Having a device-specific interrupt handler in the kernel would defeat the purpose of user space drivers. Therefore, the kernel space interrupt handler should not use detailed knowledge about the device's internal operation to clear the pending interrupt. The `uio_pci_generic` driver satisfies this requirement for devices compliant with the PCI Specification Rev. 2.3.

Multiple possibilities exist to signal the user space the occurrence of an interrupt event. These include a `signalfd` [SFD], if interrupts are relayed utilising signals, or `eventfd`'s [EFD], if event queues are used. A third option, used by Linux UIO, is to read a special file. I provide a detailed explanation of this method in Section 2.3.1.

As DDEKit-Linux was ported from DDEKit-L4, DDEKit-Linux inherited the basic interrupt handling scheme. Attachment to an interrupt results in the creation of a thread, dedicated to handling the requested interrupt. The interrupt thread itself did not need much modification[1], because all platform-specific operations are encapsulated

[1] I moved resource de-allocation into a *pthread* cleanup-handler, instead of waiting for the interrupt thread to terminate and explicitly free all resources.

in functions, called by the interrupt thread. These functions include attachment and detachment of the interrupt as well as waiting for an interrupt. Using UIO to deliver interrupts to user space, interrupt attachment, detachment and waiting are simply a matter of opening, closing and reading a file descriptor of a `/dev/uio`-device.

Attaching an interrupt handler to a device using `uio_pci_generic`

In the context of DDEKit-Linux's interrupt handling, a notion of a device does not exist; only the interrupt number is known. To attach to a device interrupt with `uio_pci_generic`, not an interrupt number but device information is required. Moreover, an interrupt number cannot be used to identify a device, because it might be a shared interrupt. DDEKit-Linux circumvents this restriction by maintaining a virtual PCI bus with only one device on it. The DDE on top of DDEKit-Linux will only see the virtual bus. Because only one device is on the bus, DDEKit-Linux knows to which device the interrupt number belongs. The device, which will be on the virtual bus, can be configured by passing a BDF-address to DDEKit-Linux.

Another solution would be to assign unique, 'virtual' interrupt numbers when the virtual bus is created. Because the interrupt number of the device is determined by the DDE from the *configuration space*, accesses to the *configuration space* have to be emulated. If the *Interrupt Line*-field in the *configuration space* is accessed, the value has to be substituted with the virtual interrupt number. DDEKit-Linux would then map the unique interrupt number to a device and attach an interrupt handler to it.

A device is bound to UIO, or more precisely to `uio_pci_generic`, by writing the *Vendor* and *Device ID* to the `new_id` file in *sysfs* (Section 2.3.3). As a result, a new UIO device is created. To distinguish multiple devices controlled by UIO, the minor number of each device is appended to the corresponding device-node. The first device will have the device-node `/dev/uio0`. To attach to the interrupts of the correct UIO-device, the minor number of the newly created device needs to be determined. If a device is bound to a UIO driver, a directory will be created in the *sysfs*-directory corresponding to the device. This directory is called `uio` and will contain another directory, named after the UIO device. DDEKit-Linux uses this procedure to establish a relationship between a (physical) device and the UIO pseudo-device. Because the minor number of the UIO-device corresponding to the requested interrupt number is now known, the device file of the UIO-device is opened.

Waiting for interrupts with `uio_pci_generic`

As mentioned in Section 2.2.4, the user space needs to reset the *Interrupt Disable* Bit before waiting for an interrupt. In DDEKit-Linux's interrupt subsystem only the interrupt number and the UIO minor number are known. Neither a (physical) BDF-address nor one of DDEKit-Linux's virtual BDF-addresses is available to designate a device for a *configuration space* access. Consequently neither DDEKit-Linux's own `ddekit_pci_read` and `ddekit_pci_write` functions nor *libpci*'s access functions can be used. Instead, the

`config`-file in the *sysfs*-directory corresponding to the UIO device [2] is used. The minor number of the UIO-device determined earlier is re-used to locate the correct entry.

Before waiting for an interrupt, the *Interrupt Disable Bit* has to be turned off to disable interrupt masking. Reading 4 bytes from the UIO-device file-descriptor afterwards returns the interrupt count so far – indicating an interrupt occurred since the last read – or blocks until an interrupt event occurs. If an interrupt event occurs, the interrupt handler, registered as callback function, is called by DDEKit-Linux.

Detaching from UIO interrupts

When detaching from interrupts, it does not suffice to close the file descriptor to the UIO-device node. The device bound to `uio_pci_generic` has to be released. A device is released by writing its physical BDF-address to a special *sysfs*-file, similar to binding a device. The file is called `unbind` and is located in the driver's *sysfs*-directory.

3.2 PCI support

Although UIO itself provides rudimentary support to access PCI resources, a generic device driver like `uio_pci_generic` has no information about device-specific resources. Hence, other features of the Linux kernel are used to access those resource from user space.

3.2.1 PCI configuration space access

On the x86 platform the PCI configuration space can be accessed by reading from and writing to two I/O ports. In Linux user space, access to I/O ports is restricted. To use `inb` and `outb` instructions, permission has to be requested using the `ioperm()` system call.

However, granting a user space process access to those I/O ports enables the process to access the PCI configuration space of every PCI device in the system. Doing so violates the *principle of least privilege* [Jer75] and alternative ways of accessing the PCI configuration space should be favoured. DDEKit-Linux uses `pci_[read,write]_[byte,word,long]()` function calls provided by libpci to facilitate configuration space access.

3.2.2 I/O Address Space access

As already mentioned, port I/O is available via the `inb` and `outb` instructions, if the necessary permissions have been granted. Failing to acquire appropriate permission will cause a segmentation fault, if `inb` or `outb` instructions are executed.

To prevent a driver from requesting I/O ports not belonging to 'its' device, requested ports should be checked against the I/O ports used by a device and access should be denied, if necessary. The I/O ports used by a device are assigned by configuration software during device enumeration. On the x86-architecture the BIOS assumes the

[2] The full directory name is `/sys/class/uio/uio`*n*`/device/`.

position of configuration software. However, BIOSes may be faulty and cause problems [MHW00]. The operating system has to work around those problems and possibly reconfigure PCI devices. In the Linux kernel, all device-configurations are known and exported to user space via *sysfs*. If the ports requested by the driver match those listed by *sysfs*, DDEKit-Linux invokes the `ioperm()` system call.

On successful return, the driver may use `inb` and `outb` instructions to communicate with the device.

3.2.3 Memory mapped I/O

When using memory mapped I/O, the memory address space is also used for device I/O. Accessing a device is performed using the same instructions as for memory access. Memory mapped I/O regions are configured during the bus enumeration process, similar to I/O port regions. Once configured, the memory mapped I/O regions need to be mapped in the virtual address space (kernel space as well as user space), to be used.

The UIO framework provides a mechanism to map I/O memory to user space. For this mechanism to work, the PCI BARs requiring remapping have to be known at compile-time of the UIO driver module. This information is not always at hand. Moreover, if prior knowledge of the device configuration is required, the driver may not be considered unmodified. Therefore, the mechanism implemented in the UIO framework is not used by DDEKit-Linux.

Instead, memory mapped I/O access is implemented using the *sysfs* virtual file system. To use a memory mapped I/O region from user space, it is sufficient to open the corresponding *sysfs* resource-file and use the `mmap` system call to map the region in the virtual address space of the user space process.

3.3 DMA

As explained in Section 2.2.5, DMA is used for data transfers to achieve better performance and to relieve the CPU. For DMA to work, the memory from or to which data is transferred is required to be physically contiguous and the physical address of the memory area has to be known.

3.3.1 Address translation in DDEKit-L4

I will consider memory management in DDEKit-L4, before describing the handling of DMA-operations in DDEKit-Linux in detail. Memory allocation is handled by a dataspace manager, which provides an option to make an allocation physically contiguous. The dataspace manager also provides the physical address of the allocated memory. Both requirements for DMA are therefore fulfilled by DDEKit-L4.

After allocating memory from a dataspace manager, the virtual address, the physical address and the size of the memory area allocated are cached in DDEKit-L4's own page table facility. Upon request the cache is searched to find a physical address for a given virtual address or vice versa. In dde-linux-2.6 [Men03], all address translations from

virtual to physical addresses are used to facilitate DMA-transactions. Thus, physical addresses are not required as such, but in the context of a DMA transaction.

3.3.2 Address translation in DDEKit-Linux

DDEKit-Linux uses the standard libC `malloc()` implementation to allocate memory. This results in the violation of both prerequisites for DMA: the physical address of the allocated memory is not known in user space and the memory may not be physically contiguous. Linux implements virtual memory management in the kernel. In contrast to L4, no means are provided by Linux to translate a virtual address to a physical address and pass it to a user space applications.

Instead of using `malloc`, all memory can be allocated in the Linux kernel and be mapped into DDEKit-Linux's address space. The Linux DMA-API [DMA] offers the possibility to allocate memory suitable for DMA. However, this approach still has a drawback: memory allocated in kernel space cannot be read or written by the `ptrace` system call, therefore prohibiting the use of debuggers, like *gdb*.

3.3.3 DMA-operation in DDEKit-Linux

Comparing the outlined address translation scheme of DDEKit-Linux with the one used in DDEKit-L4, it becomes evident that address translation does not work with DDEKit-Linux as is. Hence the DDE-DDEKit-API was extended to support DMA-operations.

Four functions comprise the DDEKit-DMA-API: first, `ddekit_dma_map_single` takes a virtual address, a size and a direction argument and returns an address on which a DMA-operation can be performed. Second, `ddekit_dma_unmap_single` reverses all actions performed `ddekit_dma_map_single`. The function takes a DMA address, a size and a direction argument to specify which DMA-buffer gets unmapped. Third and fourth, `ddekit_dma_malloc` and `ddekit_dma_free` allocate and free consistent (coherent) DMA-able memory. "Mapping" refers to all actions necessary to set up a DMA transaction, "unmapping" reverses those actions.

Allocating DMA-suitable memory in the Linux kernel

I implemented a pseudo-device to allocate DMA-able memory in the Linux kernel using the Linux DMA-API. The DMA-able memory is mapped into the address space of a user space process. The pseudo-device stores the virtual and physical addresses of all allocated and mapped memory areas.

The pseudo-device has a device node called `dma_mem`. After opening the device file, memory is allocated by calling `mmap`. The required size of the DMA-able memory is passed as the `length` argument. The protection arguments have to be `PROT_READ` and `PROT_WRITE`, so the memory can be read from and written to. To avoid a copy-on-write mapping, the `flags` argument has to contain the `MAP_SHARED` flag. When using a shared mapping with write access, the file descriptor has to be opened for reading and writing `O_RDWR`.

The pseudo-device allocates memory of the requested size using `dma_alloc_coherent`. The function returns the address of the allocated memory in the kernel and a bus address

for DMA. The memory area is remapped in the calling process's virtual memory. Before returning, all information is stored by the pseudo-device. The `mmap` system call returns the address of the DMA-memory in the process's address space.

The bus address, which can be passed to a device to perform a DMA operation, is transferred to the user space by reading the device node. The `offset`-parameter of the `pread` system call specifies the virtual address, to which a bus address is required. The pseudo-device knows about all mappings and is therefore able to return the DMA-address in the reading buffer. If a translation for an invalid virtual address is requested, nothing is returned.

The `unmap` system call removes the mapping of the DMA-buffer to user space, but does not free the allocated memory. The virtual address and size of the DMA-buffer are passed to `unmap`. To free the DMA-memory, the virtual address of the buffer is written to the pseudo-device. If a matching entry is found, the memory is freed.

When the file descriptor of the pseudo-device is closed, all remaining memory is freed.

DMA-transaction with in-kernel allocated memory

A DMA-transaction can have one of two directions: from memory to a device or vice versa. DMA buffer handling depends on the direction of the transfer. DDEKit-Linux handles a mapping request by allocating DMA-able memory from the kernel via the `dma_mem` pseudo-device. The bus address of the allocated memory is read from the pseudo-device and cached internally by DDEKit-Linux. If the direction of transfer is from memory to device, the contents of the user space buffer is copied into the DMA-able memory. DDEKit-Linux returns the bus address of the DMA-able memory as result of the mapping operation.

After the DMA-transaction is finished, the buffer is unmapped. If the direction of the transfer was from device to memory, the contents of the DMA-buffer is copied into the user space buffer. The DMA-buffer is then unmapped from the address space of the user space process and freed in the kernel.

Because copying is involved, I call this method "copy DMA".

Performing address translation and page locking

To perform DMA on memory allocated in user space, the memory has to be *locked*. As operation of the memory management subsystem, page locking can only be performed in kernel space. Locking is performed with page granularity. When a page is locked, it can neither be replaced nor swapped out. A locked page can therefore be mapped for DMA-operation. If a user space buffer spans multiple pages, a DMA-transfer is only possible, if the locked pages are contiguous in physical memory. A memory area spans multiple pages if it is bigger than one page or if it is positioned with a offset inside a page.

I implemented a second pseudo-device to lock and map user space buffers for DMA. This pseudo-device is named `dma_map`. The virtual address and size of the memory are written to the pseudo-device, along with the intended operation (mapping or unmapping). The pages backing the memory are identified by `get_user_pages_fast`. Pages

currently swapped to disc are also transferred back into RAM. If more than one page is the result, the pages are unlocked by `page_cache_release` and an error is returned. Otherwise, the page is mapped for a DMA-transfer by `dma_map_page`. The virtual address, bus address, size and involved pages are stored by the pseudo-device internally. A successful mapping is indicated by the pseudo-device with a return value of 0.

If the mapping operation has succeeded, the bus address is requested by the user space driver via the `read` system call on the `dma_map` device node. The file offset indicates the virtual address, to which the bus address is required. For this purpose the `lseek` or `pread` system call can be used. In user space the returned physical address is used to set up the DMA-transfer.

Unlocking pages is required, because locked pages would eventually occupy the entire system memory, rendering the computer system unusable. To unlock a DMA-buffer, the bus address, size, and a constant, specifying the unlock operation are written to the pseudo-device. The device searches for the bus address in the stored information and unlocks the page with `page_cache_release`.

If the file descriptor to the pseudo-device is closed, all pages still locked will be unlocked. This ensures resource de-allocation, even if the user space driver crashes.

DMA-transactions with address translation

Performing DMA with DDEKit-Linux using the pseudo-device performing locking and mapping (`dma_map`) is straightforward. The mapping request consisting of virtual address and size is written to the pseudo-device. Subsequently, the bus address is read from the pseudo-device and can be used for DMA.

A problem arises, if the buffer is larger than a page or crosses a page boundary. In case the buffer cannot be mapped, DDEKit-Linux uses the first pseudo-device (`dma_mem`) to allocate memory from the kernel and sets up the DMA-transaction as described earlier. In contrast to copy DMA (Section 3.3.3), I call this method "zero-copy DMA", because ideally no data will be copied.

The method DDEKit-Linux uses to set up DMA-transfers can be configured by a preprocessor macro. DDEKit-Linux hides all interfacing with the pseudo-devices.

Suppressing DMA-operations

A problem arises when a user space driver performing DMA quits or crashes. On program exit every resource is freed. This also holds true for DMA resources obtained from one of the pseudo-devices `dma_map` or `dma_mem`. If a PCI device is not properly shut down, it may continue to perform DMA-transactions. The memory, of which the device holds an address, may be re-used by the system for other purposes in the mean time. DMA-transactions from device to memory pose a security risk. Transferring data to this memory will corrupt the system and eventually cause a system crash. For this reason, DDEKit-Linux installs an exit-handler which revokes the *bus master* capability from the device controlled by DDEKit-Linux. If a device cannot become bus master, it cannot perform any DMA-operation.

3.4 Adapting dde-linux-2.6 to the new DMA-API

As mentioned in Section 3.3.2, DMA is not possible from Linux user space at all or at least not without seriously limiting DDEKit's usefulness in terms of debugging. Thus, I expanded the DDEKit-DDE-interface to allow DMA-transactions. The introduced change in the DDEKit-DDE-interface implies changes to dde-linux-2.6. Linux already has an API to manage DMA-transactions; the implementation in dde-linux-2.6 does not do any work, except parameter checks.

The Linux DMA-API consists of a set of functions implementing allocation of DMA-memory as well as mapping already-allocated memory for DMA. Functions facilitating de-allocation and unmapping are part of the Linux DMA-API as well. A struct `dma_mapping_ops` holds all functions pertaining to the DMA-API. An instance of `dma_mapping_ops` exists for each IOMMU-type. Each instance holds functions which set up DMA transaction IOMMU-specific. To map memory allocated in user space for DMA, the implemented function set can be viewed as "soft-IOMMU", mapping user space virtual addresses to bus addresses.

Not all functions constituting the Linux DMA-API are currently implemented; I will explain briefly the implemented functions. `Dma_alloc_coherent` is used to allocate persistent, DMA-able memory. In dde-linux-2.6 the function calls `ddekit_dma_malloc`, which in turn allocates DMA-able memory from the `dma_mem` pseudo-device. The allocated memory is remapped to user space, as explained in Section 3.3.3. The bus address of the newly allocated memory is then requested from DDEKit-Linux.

To free persistent DMA-memory, `dma_free_coherent` calls `ddekit_dma_free`. The specified memory is freed by the `dma_mem` pseudo-device. DDEKit-Linux hides all interface details from the DDE.

The `map_single` and `unmap_single` members of the `dma_mapping_ops` call directly `ddekit_dma_map_single` and `ddekit_dma_unmap_single` (Section 3.3.3), respectively.

As I stated earlier, all functions in dde-linux-2.6 belonging to the DMA-API are contained within a struct. For each implementation of an IOMMU as well as for no IOMMU such a struct exists. With this construction a DMA-mapping can be established, regardless if any or which IOMMU is used. As I already suggested, the address translation of buffers allocated in user space to DMA bus addresses can be viewed as a "soft-IOMMU". Thus, I introduced a new instance of `dma_map_ops`: `soft_iommu`. Aside from `soft_iommu_alloc_coherent` and `soft_iommu_free_coherent`, the new instance has the members `soft_iommu_map_single` and `soft_iommu_unmap_single`. Scatter-gather DMA-transfers are not implemented to date.

A cautionary note may be required to not confuse DDEKit-Linux's "soft-IOMMU" with an actual IOMMU. Using DDEKit-Linux, a Linux-host may or may not have an IOMMU. The host's IOMMU is not programmed by DDEKit-Linux directly, but by on of the DMA pseudo-devices, when memory is allocated or mapped for DMA. After translating the virtual address into a physical address, the physical address is translated in a bus address by the Linux DMA-API. In the same step the physical IOMMU, if present, is programmed.

3.5 Threading, Synchronisation and Timers

The DDEKit-interface includes primitives for threading, synchronisation and timers. DDEKit supports thread creation, suspension of a thread's execution and thread local storage. Since DDEKit-L4 implemented threading using the *pthreads*-library, I did not need to modify the source code.

DDEKit's synchronisation primitives comprise condition variables, semaphores and mutexes. Condition variables and mutexes are part of the *pthreads*-library as well and are used by the DDEKit-L4 implementation. Thus, no effort for porting these parts were required. Semaphores, on the other hand, are not part of *pthreads*-library and are now implemented using POSIX semaphores (Linux's `sem_t`).

Timers in DDEKit are one-shot timers. All timer events are kept in a sorted list. The period to the first timer event in the list is bridged by a timed wait on a semaphore. If a new event is enlisted it might be scheduled before the current head of the list. Therefore, the semaphore is posted to wake the timer thread up. If the time for the first timer is up, it is executed. Otherwise, the amount of time until the next timer event is due is re-calculated.

4 Evaluation

I evaluated DDEKit-Linux with regards to security, performance and implementation effort. Performance aspects are throughput and latency of a PCI network card. I measured implementation effort by determining lines of code.

4.1 Security

Although DDEKit-Linux is not designed to allow secure execution of malicious drivers, a certain fault-resistance is needed so a bug will not compromise the machine. The resources protected by DDEKit-Linux are I/O ports and memory.

As user space process a driver running on top of DDEKit-Linux can only access memory resources allocated to it. Thus it cannot compromise other processes, other device drivers, or the kernel.

A driver should only access I/O ports used by the device it is controlling. DDEKit-Linux enforces this I/O port access restriction by checking the requested I/O port range against the I/O port range mapped by the device. If the requested I/O port range is a subset of the I/O port range used by the device, access is granted to the user space process running the driver via the `ioperm()` system call. Subverting the range check and accessing an I/O port directly will result in the delivery of a segmentation violation in conjunction with process termination. The drawback of this method is the need for root privileges or at least the `CAP_SYS_RAWIO` capability. Accessing I/O ports using *sysfs* resource files does not have these restriction but requires changes to dde-linux-2.6. The required changes involve the substitution of in-line assembly `inb` and `outb` instructions with callbacks to DDEKit.

In DDEKit-Linux, DMA is not restricted in any way. The driver is trusted to supply the device with correct DMA-addresses. In [BWZ10] a scheme is described to perform DMA safely from user space. It relies on an IOMMU and PCI express bridges to protect the system from erroneous or malicious DMA.

4.2 Performance

To evaluate DDEKit-Linux's performance, a user space driver for a network interface card (NIC) was run with dde-linux-2.6. I measured latency with and without a TCP/IP-stack on top of dde-linux-2.6. I performed throughput measurements with (TCP and UDP) and without (UDP) a protocol stack.

4.2.1 Test Setup

All tests were performed on an 2.66 GHz Intel Celeron based computer with 2 GiB system memory. The on-board network card, an Intel 82573E GBit Ethernet Controller, is attached via PCIe. A Macbook Pro, featuring a Core 2 Duo CPU, 4 GiB system memory, and a Broadcom 5764 Ethernet Controller, was connected to the test machine. The machines were connected directly, without a hub or a switch.

I used the *Iperf*-tool [Ipe] to measure the data rate. The test-computer served as *Iperf*-server i.e., data sink; the Macbook Pro was configured as *Iperf*-client, generating traffic. Both, UDP and TCP measurements were performed with the lwIP TCP/IP stack [LWI] on top of dde-linux-2.6. UDP tests with stack were performed uni-directionally and bi-directionally. For a uni-directional connection the test application just sinks the data, whereas the data is echoed back for bi-directional testing. For comparison the same tests were performed with the `e1000e`-module loaded in the kernel and the standard, in-kernel Linux TCP/IP stack. To determine the influence of the lwIP-stack on the results, I repeated the UDP measurement without a stack. To measure UDP receive performance without a stack, I recorded the size of the received packets and discarded them. All data rates were obtained by averaging over a 10 second window.

To perform latency measurements the *ping*-tool was used with the same hardware setup. I executed the *ping*-command on the Macbook Pro, letting the lwIP-stack generate the ICMP echo replies. The round trip delay time was also determined with the *ping*-tool and a small handler running on top of dde-linux-2.6, generating replies.

I also measured the delay time, introduced by user space interrupt handling, using the `rdtsc`-instruction. The time between when the UIO interrupt handler was called and the `read`- system call on the `/dev/uio`-device returned, is considered the introduced delay.

4.2.2 Latency

All latency results are the mean value of 2^{14} measurements. I summarise the results in Table 4.1. At first, I evaluated DDEKit-Linux with the zero-copy DMA facility. DDEKit-Linux exhibited a low latency of 0.58 ms ($\sigma^2 = 0.65$) in comparison with 0.34 ms ($\sigma^2 = 0.03$) for an in-kernel driver and stack. An IPv4 based ICMP packet is very small; it is only 8 bytes big. Accounting for the ethernet header, IPv4 header, and the standard 56 bytes payload the packet has a total size of 98 bytes. To evaluate the different methods for DMA, I repeated the test with 1472 bytes of payload, resulting in a packet size of 1500 bytes, the maximum for ethernet. The latency increased to 2.17 ms ($\sigma^2 = 4.8$) in contrast to just 0.42 ms ($\sigma^2 = 0.02$) for the in-kernel driver and IP-stack. Although the result was expected to worsen, because more data needed to be copied between the user space buffer and the DMA-buffer, the increase in variance was not. DDEKit-Linux performed best without lwIP-stack, which was expected. Without stack, the latency was 1.4 ms ($\sigma^2 = 3.6$) for a 1500 byte packet using the zero-copy DMA method and declined to 0.53 ms ($\sigma^2 = 0.04$) using copy DMA. For 56 bytes payload I measured 0.42 ms ($\sigma^2 = 0.06$) for zero-copy DMA and 0.43 ms ($\sigma^2 = 0.01$) for copy DMA, respectively.

Type	56 bytes Payload				1472 bytes Payload			
	zero-copy DMA		copy DMA		zero-copy DMA		copy DMA	
Linux Kernel	0.341 (0.030)				0.419 (0.022)			
LWIP	0.58	(0.656)	0.55	(0.021)	2.17	(4.807)	0.657	(0.11)
LWIP improved	0.508	(0.459)	0.507	(0.027)	2.084	(4.614)	0.625	(0.103)
Without stack	0.415	(0.06)	0.428	(0.014)	1.488	(3.682)	0.534	(0.039)

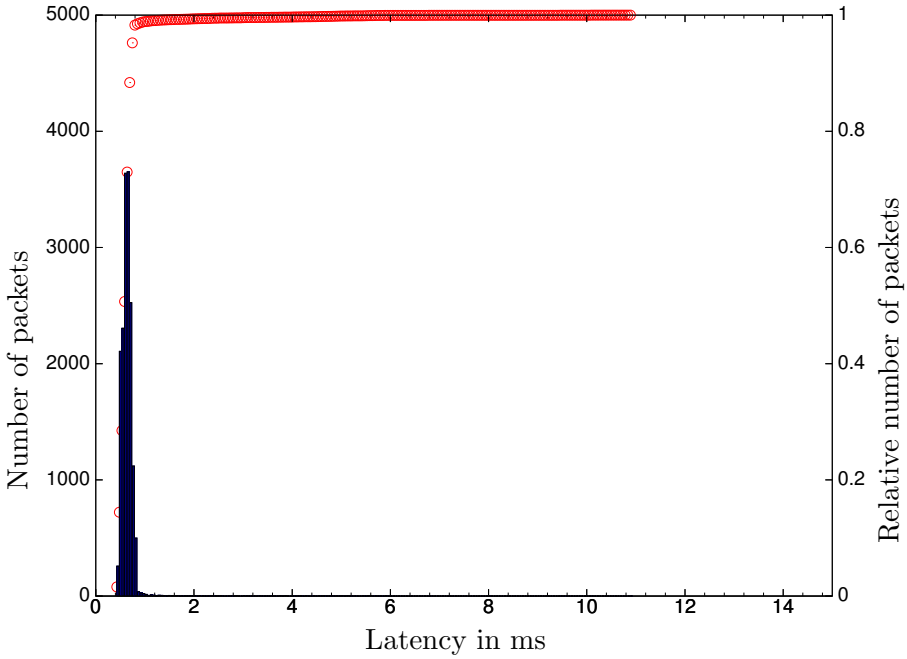
Table 4.1: Summarised latency data in ms (variance in parenthesis) under various conditions, averaged over 2^{14} runs.

Subsequently, DDEKit-Linux was modified to use the copy-method for DMA. The change resulted in a slightly better round trip time of 0.55 ms ($\sigma^2 = 0.02$) for the 56 bytes payload and 0.65 ms ($\sigma^2 = 0.11$) for the 1472 bytes payload. As a conclusion, the driver using DDEKit-Linux and LWIP needed about 61 % more (56 % more latency for 1472 bytes payload) than the same driver in the Linux kernel using the Linux in-kernel TCP/IP-stack.

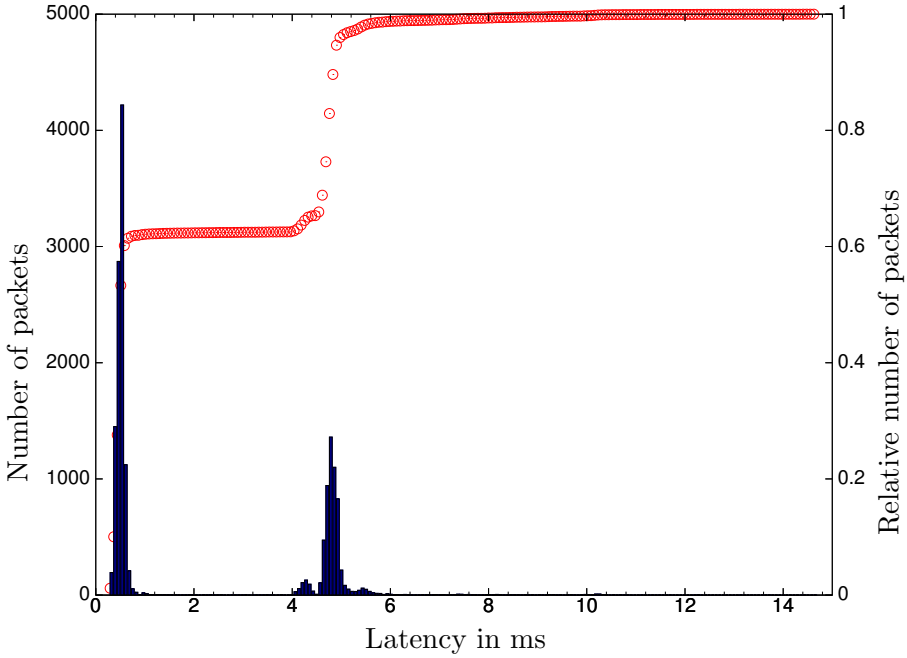
I used the *callgrind*-tool from the *valgrind*-tool-suite [NS07] to identify performance bottlenecks. The interface of the `dma_map` pseudo-device (performing address translation) incurred the most significant performance penalty. To allow easy usage of the interface all parameters are passed ASCII-encoded. The functions `snprintf` and `sscanf` were used to create and process ASCII-encoded strings. The processing time of those two functions accumulated to five percent of the total processing time. After altering the interface to binary coded parameters, the round trip times were 0.48 ms ($\sigma = 0.02$) and 0.6 ms ($\sigma = 0.02$) for the 56 bytes payload and the 1472 bytes payload respectively. The user space driver now performed 49 % slower for both payload sizes. Without stack DDEKit-Linux was 22 % slower for 56 bytes payload and 27 % for 1472 bytes payload.

I further analysed the latency data, to explain the bad performance of the zero-copy DMA method. Figure 4.1 shows histograms of the measurements with 1472 bytes payload. Both methods handle the bulk of packets in under 1 ms. However, zero-copy DMA shows a second peak around 5 ms (4.1(b)). The second peak is caused by memory buffers which crossed a page boundary and therefore were not mapped for DMA. Hence, copy DMA was used as fallback method to transfer those packets. 37.6 % of the packets in the data set for zero-copy DMA had a longer round trip time than 2.5 ms. I assume for those packets failed the mapping of the buffer. This fits the observation, that 20 % to 50 % of packets are not mapped for zero-copy DMA. The number of packets, which can be mapped declines with increasing packet size.

Figure 4.2 also underlines the assumption that the poor latency with zero-copy DMA is caused by failed mapping attempts. The lower graph shows the latency for every packet in a trial run with 1472 bytes payload in a zero-copy DMA configuration. The second graph shows the relative number of buffers which could not be mapped for DMA i.e., had to be copied. As stated earlier, the lower bound of buffers, which cannot be mapped is about 20 %. At the same time, I measured a low latency.



(a) Distribution of packet latencies (left axis) and cumulative number of packets (right axis) for copy DMA with 1472 bytes payload.



(b) Distribution of packet latencies (left axis) and cumulative number of packets (right axis) for zero-copy DMA with 1472 bytes payload.

Figure 4.1: Latency analysis for both DMA-methods with 1472 bytes payload.

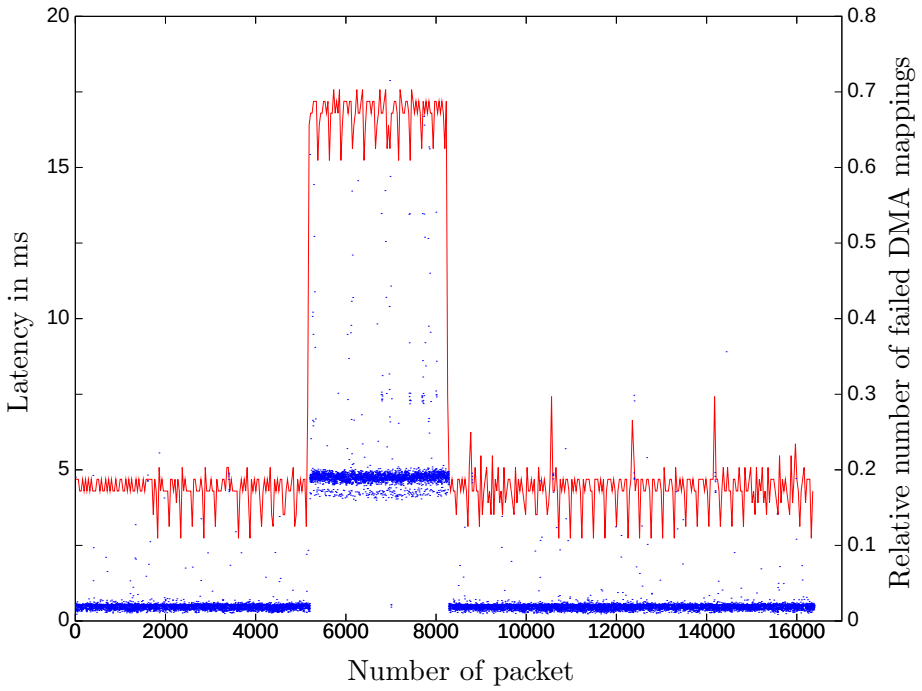


Figure 4.2: The diagram shows the latency for all packets in the trial run (left scale). Overlaid is a graph of the relative amount of buffer, which could not be mapped for DMA (right scale). An offset was added to the second graph for better readability.

When latency increased to about 5 ms, about 70 % of mappings failed. I could not discern whether the high latency was a result of the failed mappings or if both effects were the result of another common cause.

Interestingly, for all measurements involving the LwIP stack, the high latency packets were evenly distributed over the entire measurement period (Fig. 4.3). The cause for the 'clustering' of mapping errors is unknown.

4.2.3 Interrupt delay introduced by the kernel space – user space boundary

I measured the delay from the kernel space interrupt handler until the user space is informed of the interrupt event. I used the time stamp counter to perform the measurements. The first measurement was taken, before the UIO interrupt handler woke up the wait queue. In user space, the second measurement was taken right after the interrupt thread returned from the `read` system call of the `/dev/uio`-device. The results are summarised in Table 4.2. All presented data is the average over 3,000 runs. I repeated the measurement under different conditions. At first, I measured the delay, when no data was transferred. The interrupt source was the watchdog timer of the NIC, resulting in a delay of 20.8 μs. Subsequently, I transferred data using the *ping* utility

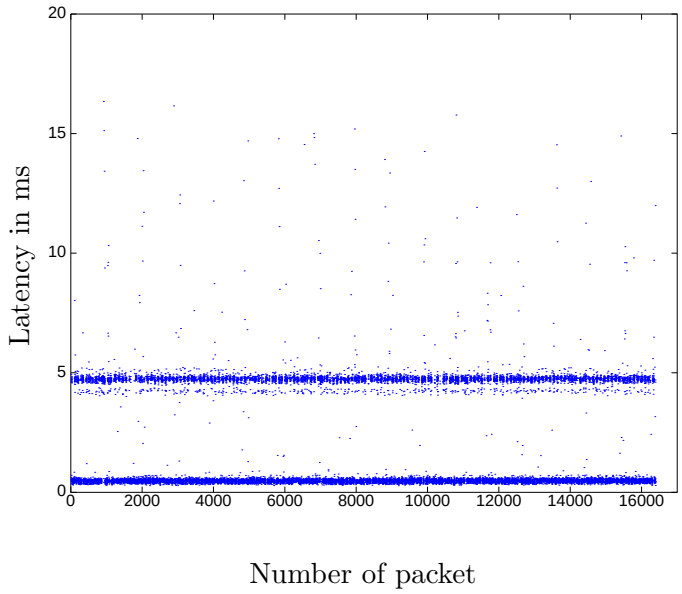


Figure 4.3: Latency data for the improved DMA-interface with lwIP-stack, 1472 bytes payload and zero-copy DMA.

Measurement condition	Ticks	μs	%
watchdog interrupts	55,514	20.8	100%
idle	56,320	21.2	+1.9%
minimal priority process	61,845	23.2	+11.5%
normal priority process	68,438	25.7	+23.5%
maximum priority process	66,569	25.0	+20.2%

Table 4.2: Interrupt delay under various conditions, averaged over 3,000 runs.

and measured the delay again (21.1 μs). Additionally, I increased CPU usage by running another process with different priorities (23.2 μs, 25.7 μs and 25.0 μs). As expected, the delay increased when data was processed or the CPU was otherwise fully utilised.

4.2.4 Throughput

Initially, DDEKit-Linux performed really badly, maintaining a data rate of 5.6 MBit/s with TCP (Fig. 4.4) and 5.9 MBit/s with UDP (2.7 MBit/s bidirectional) (Fig. 4.5). The reason was the zero-copy DMA method. Whenever possible, a user space buffer was locked into memory and its bus address was used for DMA. Although this is a zero-copy solution, performance is degraded by the frequent kernel entries and exits. To lock and map a user space buffer a `read` and a `write` system call have to be executed. Additionally overhead was added by the required locking, not only in DDEKit-Linux, but

also in the `dma_map` pseudo-device. DDEKit-Linux locks data structures caching buffer information in user space. The mapping pseudo-device needs to acquire a semaphore before modifying memory management structures in the kernel.

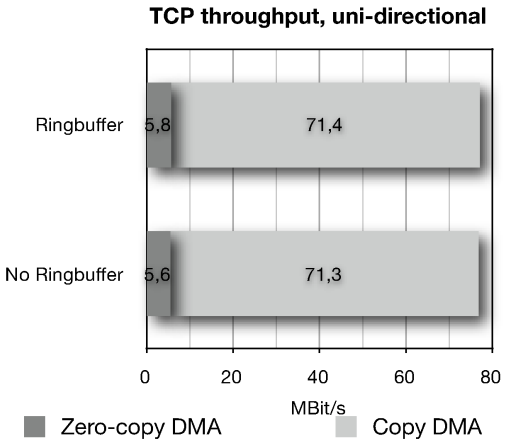


Figure 4.4: TCP throughput for both DMA-methods offered by DDEKit-Linux. The performance of the in-kernel `e1000e`-module was 504 MBit/s.

To improve performance, DMA setup in DDEKit-Linux was changed to use only the `dma_mem` pseudo-device. UDP throughput increased to 78.4 Mbit/s (74.9 Mbit/s bi-directional). TCP performance improved to 71.3 Mbit/s.

I introduced a ring buffer to decouple packet reception and processing in the stack. As a result UDP performance increased slightly to 92.6 MBit/s (78.5 Mbit/s bi-directional). Re-evaluation of the zero-copy DMA setup with a ring buffer showed little or no increase in throughput. Please note that this optimisation is not part of DDEKit or dde-linux-2.6, but the user space application.

After removing the stack completely, the driver was able to receive packets with a data rate of 221.2 MBit/s (UDP) on a 1 GBit/s link.

The in-kernel `e1000e`-module achieved a data rate of 504 Mbit/s (TCP) and 490 MBit/s (UDP).

I further examined the setup without TCP/IP stack, using the *callgrind*-tool. *Callgrind* revealed DDEKit-Linux's page table facility as performance bottleneck. The page table facility is tightly interwoven with DDEKit's DMA handling. If address translation is not possible, DMA-able memory is allocated. Along with the allocation, information of the memory is stored in the page table facility. To perform DMA, the bus address has to be extracted from the stored information. The page table facility is implemented as a linked list, which must be searched for the required entry. The traversal of the list limits DDEKit-Linux's performance.

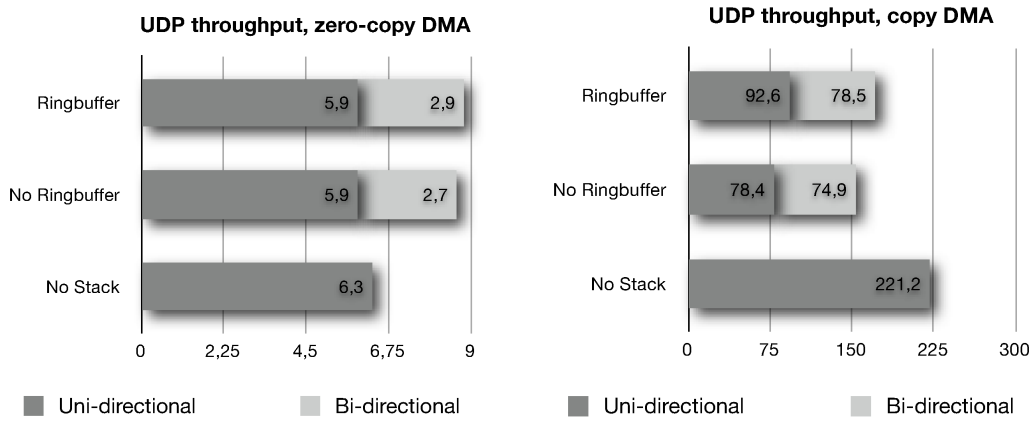


Figure 4.5: UDP throughput for both DMA-methods. The in-kernel `e1000e`-driver achieved 490 MBit/s.

4.3 Effort

I counted source lines of code to estimate the effort required to implement DDEKit-Linux. All numbers were obtained with SLOCCount [SLO] and are subsumed in Table 4.3. DDEKit-Linux comprises about 2,736 lines of C source code running in user space. Additionally two Linux kernel modules are required to allocate DMA-able memory and map it to user space (301 lines) and translation of virtual address to bus addresses (380 lines).

The Linux UIO framework adds 725 lines and the `uio_pci_generic`-module 139 lines, respectively. The total amount of Linux kernel code is 1,545 lines.

Mostly, DDEKit required only minor changes, resulting from the use of the cross-platform *pthreads*-library. I added 5 source files related to handle PCI devices and resources. The DMA-capabilities of DDEKit-Linux account for many changes and comprise much of the source code. DDEKit-L4 spanned about 2,000 lines of code, from which I removed about 1,200 and added about 2,000 lines to implement DDEKit-Linux.

Component	**SLOC**
DDEKit-Linux (user space)	2,736
`dma_mem`-module	301
`dma_map`-module	380
Linux UIO module	725
`uio_pci_generic`-module	139
Total	4,281

Table 4.3: Lines of C-code for the individual sub-modules of DDEKit-Linux.

5 Conclusions and future work

In this thesis I introduced DDEKit-Linux. DDEKit-Linux is an abstraction layer supporting DDEs on a Linux host. DDEKit-Linux was evaluated using dde-linux-2.6 to run a network card driver in Linux user space. The commonly used user space tools *gdb* and the *valgrind*-suite were used to develop, debug, and optimise DDEKit-Linux as well as sample applications, thereby validating the claimed advantages of user space drivers. Although the performance of DDEKit-Linux lagged behind expectations, I pointed out several possibilities to improve DDEKit-Linux's performance.

Future work

Security

DDEKit-Linux already provides strong protection, for example memory protection as a result of running as a user space process. It is possible to equip DDEKit-Linux with mechanisms ensuring even stronger protection. Interrupts can be rate-limited to avoid (accidental or intentional) interrupt storms. MSI-X offers generic interrupt masking, similar to PCI 2.3. IOMMUs can be used to prevent arbitrary DMA transaction into system memory, whereas PCIe transaction filtering isolates PCIe devices on the same bus. The PCI configuration space access can be emulated to prohibit malicious device re-configuration.

Usability

Since Conventional PCI is gradually displaced by PCIe, DDEKit-Linux, and especially UIO, should support MSI(-X) interrupts. MSI(-X) interrupts are edge-triggered; therefore interrupt masking is not strictly required for MSI(-X) to work with UIO. However, the interrupt masking capability is optional in MSI, but not in MSI-X. Masking interrupts may be used to suppress interrupt storms. DDEKit-Linux's implementation of interrupt handlers with one thread per interrupt is well-suited to handle multiple MSI(-X) interrupt vectors.

DDEKit-Linux might also benefit from a "device-proxy" as presented in [Kan10]: A in-kernel device proxy forwards application requests to a user space device driver and the device driver from the hardware to the kernel (or user space). Although performance can be expected to diminish considerably, the advantage is the seamless integration of user space drivers in Linux. Application might use device nodes or the Linux socket API directly. The current implementation requires changes to the source code or that an application is linked into DDEKit-Linux. Additionally, multiple applications might

share the same network card. Similarly, devices which are accessed through device nodes in the `/dev`-directory could be used without change to applications.

Performance

As already mentioned in Section 4, the DMA-facility seems to offer the best prospects of performance improvement. Both DMA pseudo-device interfaces should be revised to use `ioctl`s to minimise system call overhead. An interface using the `ioctl` system call instead of `read` and `write` can be expected to be faster, since less kernel entries have to be performed.

Zero-copy DMA should only be used when an IOMMU is available. This ensures that memory is always mappable and the overhead of the fallback method is avoided.

When using copy DMA, the interface of the `dma_mem` pseudo-device should be extended. Instead of allocating "consistent" memory, the direction of the intended DMA-transaction should reflect on the memory allocation in the kernel. DDEKit-Linux might also profit from caching already-allocated DMA memory, instead of freeing it, alike to a pool-allocator.

Bibliography

[AMD] AMD. AMD I/O Virtualization Technology (IOMMU) Specification. URL `http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf`.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. *SIGOPS Oper. Syst. Rev.*, 29:267–283, December 1995.

[BWZ10] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Driver in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, June 2010. URL `http://people.csail.mit.edu/nickolai/papers/sbw-sud.pdf`.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, 2005. URL `http://lwn.net/Kernel/LDD3/`.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[DMA] Dynamic DMA mapping. URL `http://lwn.net/2001/0712/a/dma-interface.php3`.

[DRO] Dresden Realtime Operating System. `http://os.inf.tu-dresden.de/drops/`.

[EFD] eventfd. URL `http://lwn.net/Articles/226252/`.

[EG04] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In *9TH ASIA-PACIFIC COMPUTER SYSTEMS ARCHITECTURE CONFERENCE*, 2004.

[FFH+96] Bryan Ford, Bryan Ford, Mike Hibler, Mike Hibler, Jay Lepreau, Jay Lepreau, Patrick Tullmann, Patrick Tullmann, Godmar Back, Godmar Back, Stephen Clawson, and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151, 1996.

[Fri06] Thomas Friebel. Übertragung des Device-Driver-Environment-Ansatzes auf Subsysteme des BSD-Betriebsystemkerns. Diplomarbeit, TU Dresden, 2006.

[Gen] Genode DDEKit. `http://genode.org/documentation/api/dde_kit_index`.

[GRB+08] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *ASPLOS XIII: Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 13, March 2008.

[Hel01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS Architektur. Diplomarbeit, TU Dresden, 2001.

[HUR] HURD DDEKit. `http://www.gnu.org/software/hurd/user/zhengda.html`.

[Int] Intel. Intel® Virtualization Technology for Directed I/O. URL `http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`.

[Ipe] Iperf. URL `http://sourceforge.net/projects/iperf`.

[Jer75] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1975.

[Kan10] Antti Kantee. Rump Device Drivers: Shine On You Kernel Diamond. URL `http://ftp.netbsd.org/pub/NetBSD/misc/pooka/tmp/rumpdev.pdf`, 2010.

[LCFD+05] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20, 2005.

[LU04] Joshua LeVasseur and Volkmar Uhlig. A Sledgehammer Approach to Reuse of Legacy Device Drivers. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004.

[LWI] lwIP - A lightweight TCP/IP stack. URL `http://savannah.nongnu.org/projects/lwip/`.

[Mar] Martin Mareš. PCI Utilities. URL `http://mj.ucw.cz/pciutils.html`.

[Mar99] Kevin Thomas Van Maren. The Fluke Device Driver Framework. Master's thesis, University of Utah, 1999.

[Men03] Marek Menzer. Portierung des DROPS Device Driver Environment (DDE) für Linux 2.6 am Beispiel des IDE-Treibers. Großer Beleg, TU Dresden, 2003. URL `http://os.inf.tu-dresden.de/papers_ps/menzer-beleg.pdf`.

[MHW00] Ron Minnich, James Hendricks, and Dale Webster. The Linux BIOS. In *Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta, Georgia, USA, 2000. URL `http://www.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/minnichBIOS/minnichBIOS.pdf`.

[Moc05] Patrick Mochel. The sysfs Filesystem. In *OLS 2005*, 2005. URL `http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf`.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, California, USA, June 2007. URL `http://valgrind.org/`.

[PCI02] PCI Local Bus Specification Revision 2.3, 2002. URL `http://www.pcisig.com/specifications/conventional/`.

[Rit97] S. Ritchie. Systems programming in Java. *Micro, IEEE*, 17(3):30 –35, May/Jun 1997.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[SFD] Kernel events without kevents. URL `http://lwn.net/Articles/225714/`.

[SLO] SLOCCount. URL `http://www.dwheeler.com/sloccount/`.

[SMLE02] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[UIO07] UIO: user-space drivers, 2007. URL `http://lwn.net/Articles/232575/`.

[UML] User-mode Linux. URL `http://user-mode-linux.sourceforge.net/`.