Generic User-Level PCI Drivers

Hannes Weisbach, Björn Döbel, Adam Lackorzynski
Technische Universität Dresden
Department of Computer Science, 01062 Dresden
{weisbach,doebel,adam}@tudos.org

Abstract

Linux has become a popular foundation for systems with real-time requirements such as industrial control applications. In order to run such workloads on Linux, the kernel needs to provide certain properties, such as low interrupt latencies. For this purpose, the kernel has been thoroughly examined, tuned, and verified. This examination includes all aspects of the kernel, including the device drivers necessary to run the system.

However, hardware may change and therefore require device driver updates or replacements. Such an update might require reevaluation of the whole kernel because of the tight integration of device drivers into the system and the manyfold ways of potential interactions. This approach is time-consuming and might require revalidation by a third party. To mitigate these costs, we propose to run device drivers in user-space applications. This allows to rely on the unmodified and already analyzed latency characteristics of the kernel when updating drivers, so that only the drivers themselves remain in the need of evaluation.

In this paper, we present the Device Driver Environment (DDE), which uses the UIO framework supplemented by some modifications, which allow running any recent PCI driver from the Linux kernel without modifications in user space. We report on our implementation, discuss problems related to DMA from user space and evaluate the achieved performance.

1 Introduction

Several advantages make the Linux kernel an attractive OS platform for developing systems with real-time capabilities in areas as diverse as industrial control, mobile computing, and factory automation: The kernel supports many popular computing platforms out of the box, which provides a low barrier starting to develop software for it. Being open source, it can be easily adapted to the target platform's needs. A huge community of developers guarantees steady progress and fast response to problems.

Applying Linux in a real-time environment however leads to additional problems that need to be handled. We imagine a system where a computer controls a safety-critical industrial machine while in parallel providing non-real-time services. For example, it might provide work statistics through a web server running on the same machine. The RT_PREEMPT series of kernel patches [1] aims to provide the ability to do so. However, use of Linux in safety-critical systems would need additional audits and certifications to take place.

The web server in above example makes use of an in-kernel network device driver. Now, if the network driver needs to be upgraded for instance because of a security-related bugfix, the whole kernel or at least parts of it would need to be reaudited. These certifications incur high cost in terms of time and manual labor. They become prohibitively expensive when they need to be repeated every time a part of the system is upgraded.

Running device drivers in user space allows to circumvent recertification of the whole kernel by encapsulating the device driver in a user-level application. If, like in our example, the network is solely used by non-real-time work, it can be completely run outside the real-time domain and doesn't need to be certified at all.

Linux already comes with UIO, a framework for writing device drivers in user space [5]. However, these drivers still need to be rewritten from scratch using UIO. In this paper we propose an alternative technique: Using UIO and other available kernel mechanisms, we implement a Device Driver Environment (DDE) – a library providing a kernel-like interface at the user level. This approach allows for reusing unmodified in-kernel drivers by simply wrapping them with the library and running them at the user level.

In the following section, we introduce the general idea of the DDE and inspect the UIO framework with respect to its support of a generic user-level driver layer. We then discuss our implementation of a DDE for Linux in Section 3. Thereafter, we continue analyzing the special needs of Direct Memory Access (DMA) from user space in Section 4 and present a solution that requires only minimal kernel support. In Section 5 we evaluate our DDE implementation with an in-kernel e1000e network device driver running as user-space application.

2 User-Level Device Drivers for Linux

Device drivers are known to be one of the single most important sources of bugs in today's systems [4]. Combined with the fact that most modern operating systems run device drivers inside their kernel, it is not surprising that a majority of system crashes is caused by device drivers – Swift and colleagues reported in 2003 that 85% of Windows crashes may be attributed to device driver faults [24].

One way to improve reliability in this area is to separate device drivers from the kernel and run them as independent user-level applications. Doing so isolates drivers from each other and the remaining components and increases the chance that a faulting driver does not take down the rest of the system. Properly isolated drivers may be restarted after a crash as it is done in Minix3 [13]. Performance degradation resulting from moving drivers out of the kernel into user space is often considered a major disadvantage of this approach. However, it has been proven that user-level device drivers may achieve the same performance as if run in the kernel [16]. Further research showed that existing device drivers can be automatically retrofitted to run most of their critical code in user space and only keep performance-critical paths within the kernel [11].

Our ultimate goal is to provide a Device Driver Environment, a common runtime library that can be linked against arbitrary in-kernel device drivers in order to run them as user-level applications without modification. In this section we give an overview of the DDE approach and analyze Linux' UIO framework regarding its capabilities of supporting generic user-level device drivers.

2.1 The DDE Approach

Our approach for reusing in-kernel device drivers in user space is depicted in Figure 1. The source code of an unmodified native Linux device driver is linked against a wrapper library, the Device Driver Environment. The wrapper provides all functions the driver expects to be implemented originally by the Linux kernel. The DDE reimplements these functions solely using mechanisms provided by a device driver abstraction layer, called DDEKit.

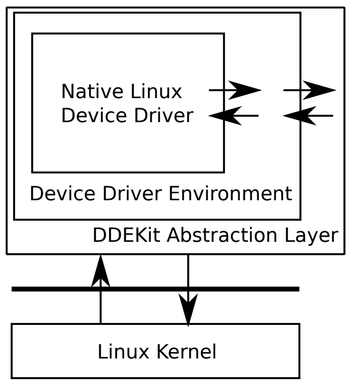

FIGURE 1: *DDE Architecture*

Only DDE knows about the intricate requirements of guest drivers. In turn, the DDEKit provides abstract driver-related functionality (device discovery and management of device resources, synchronization, threading, etc.) and implements it using functionality from the underlying host OS. Splitting development into these two layers allows to use a DDEKit for a certain host platform in connection with different guest DDE implementations as well as reuse the same guest DDE on a variety of hosts. This layering has allowed for implementations of DDE/DDEKit for different guest drivers (Linux, FreeBSD [10]) as well as different host platforms (Linux, L4/Fiasco [12], Genode [15], Minix3 [25], GNU/HURD [7]).

In this paper we focus on implementing a DDE for Linux PCI device drivers on top of the Linux kernel. To achieve this goal, it is necessary to understand the facilities at hand to perform device driver-related tasks from user space. The User-level IO framework (UIO) appears to be a good starting point for this.

2.2 UIO Overview

The Linux user-level IO framework (UIO) is an extension to the kernel that allows user-level drivers to access device resources through a file interface and is depicted in Figure 2. The interfacing is performed by the generic `uio_core`. In addition to that, UIO relies on a tiny device-specific driver stub, labelled `uio_dev` in the figure. During startup, this stub obtains information about the device's I/O resources and when encountering an interrupt takes care of checking whether the interrupt was raised by the device and handles the device-specific way of acknowledging the interrupt.

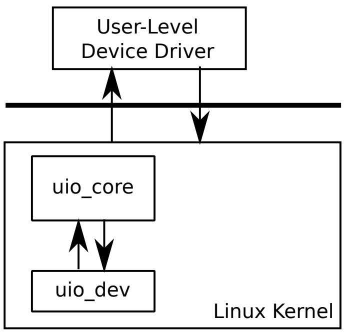

FIGURE 2: *UIO components and their interaction*

A user-level driver obtains access to the target device's resources through a `/dev/uioXXX` device file. Reading the device returns the number of (interrupt) events that occurred since the last read. Device I/O memory can be accessed by `mmap`'ing the device. UIO neither supports x86 I/O ports[1] nor direct memory access (DMA).

UIO for Generic User-Level Drivers

Our goal is to implement a DDE that allows generic PCI device drivers to be run in user space. This does not fit well with UIO's dependence on a device-specific stub driver. Unfortunately, there is no generic way to move acknowledgment of an interrupt out of the kernel. Instead, this is often highly device-specific and requires running in kernel mode.

As an exception, the situation improves with PCI devices that adhere to more recent versions of the PCI specification [21] (v2.3 or later). These devices allow generic detection of whether an interrupt is asserted using an *interrupt state* bit in the PCI config space. Furthermore, it is possible to generically disable interrupt delivery using an *interrupt disable* bit. This enables the implementation of a generic UIO PCI driver and removes the requirement of a device-specific driver stub.

The lack of support for user-level DMA is another issue that needs to be resolved in order to support arbitrary user-level PCI drivers. In the following sections we present the details of our implementation of a DDE for Linux.

3 A DDE For Linux

As described in Section 2.1, the user-space driver environment consists of two parts: a host-specific DDEKit providing a generic device driver abstraction and a guest-specific DDE that solely relies on the functionality provided by the DDEKit. For our implementation, we can build upon the already existing Linux-specific DDE for the L4/Fiasco microkernel [12]. In addition to that we need to implement a DDEKit for Linux as a host, which we describe in this section.

3.1 Anatomy of a DDEKit

The DDEKit's task is to provide a generic interface that suits the needs of guest device driver environments. To come up with this interface, we analyzed network, block, and character device drivers in two different kernels (Linux and FreeBSD) [10], resulting in a list of mechanisms all these drivers and their respective environments rely on.

The most important task of a device driver is managing I/O resources. Therefore, a driver abstraction layer needs to provide means to access and handle interrupts, memory-mapped I/O, and I/O ports. As most of the drivers we are concerned with are PCI device drivers, DDEKit also needs to provide ways to enumerate the system's PCI devices or at least discover the resources that are attached to a device. Additionally, means for dynamic memory management are crucial when implementing anything but the most simple device driver.

While most device drivers operate single-threaded, threading plays an important role in DDE implementations, because threads can be used to implement tasks such as interrupt handling, Linux SoftIRQs, as well as deferred activities (work queues). The existence of threading implies that synchroniza-

[1]Actually, UIO does not need to support I/O ports, because these can be directly accessed by the user application if it is given the right I/O permissions.

tion mechanisms such as locks, semaphores, and even condition variables need to be present.

Furthermore, a lot of drivers need a notion of time, which Linux drivers usually obtain by looking at the magic `jiffies` variable. Hence, DDEKit needs to support this. Apart from these features, in order to be useful, the DDEKit also provides means for printing messages and a link-time mechanism for implementing prioritized init-calls, that is functions that are automatically run during application startup before the program's main function is executed.

3.2 I/O Ports and Memory

In order to drive PCI devices and handle their resources, DDEKit needs means to discover devices at runtime. This is implemented using *libpci* [18], which allows scanning the PCI bus from user space. The located devices are then attached to a virtual PCI bus implemented by DDE. At runtime, any calls by the driver to the PCI subsystem use this virtual bus to perform their work.

After the virtual PCI bus is filled with the devices to be driven, information about the provided resources is obtained from `/sys/bus/pci/devices/.../resource`. Access to I/O ports is later granted by first checking whether the ports requested by the driver match the ones specified by the resource file, and thereafter granting the process port access using the `ioperm` system call. I/O memory resources are also validated and then made accessible by `mmap`ing the respective *sysfs* resource files.

3.3 Interrupt Handling

For managing interrupts, DDEKit/Linux makes use of the UIO interrupt handling mechanism, which supports generic interrupt handling through the `uio_pci_generic` module for all PCI devices supporting the PCI specification v2.3 or higher.

Once the driver requests an IRQ for a device, DDEKit locates the generic UIO driver's *sysfs* node (`/sys/bus/pci/drivers/.../new_id`). It then writes the PCI device's device and vendor IDs into this file and thereby makes `uio_pci_generic` become responsible for handling this device's interrupts.

Thereafter, a new interrupt handler thread is started. This thread performs a blocking read on the UIO file that was generated when attaching `uio_pci_generic` to the device. Whenever the read returns, at least one interrupt event has occurred and the handler function registered by the driver is executed.

The interrupt handler thread is the only one polling the UIO device file for interrupts. After successful return from the blocking read, the *sysfs* node for the device's PCI config space (`/sys/class/uio/.../config`) is written to disable IRQs while handling the interrupts. In order to avoid interrupt storms in the kernel while the user-level driver is executing its handler, the disabled interrupt is only turned on right before the interrupt thread becomes ready to wait for the next interrupt by reading the UIO device.

3.4 Threads and Synchronization

Threads are a fundamental building block of a DDE, because drivers may use a wide range of facilities that might be executed in parallel: soft-IRQs, kernel threads, and work queues are implemented by spawning a dedicated thread for each such object. Furthermore, threads are used for implementing interrupts as discussed in Section 3.3. However, not all of these activities are actually allowed to execute in parallel. Therefore, means for (blocking) synchronization are needed.

As DDEKit/Linux is implemented to run in Linux user space, we can make use of the full range functions provided by the `libpthread` API to implement threading as well as synchronization.

3.5 Timing

Linux device drivers use timing in two flavors: first, the `jiffies` counter is incremented with every clock tick. DDEKit/Linux emulates `jiffies` as a global variable. During startup, a dedicated `jiffies` thread is started that uses the libC's `nanosleep` to sleep for a while and thereafter adapt the `jiffies` counter accordingly. For the drivers we experimented with so far, it has proven sufficient to not tick with HZ frequency as the Linux kernel would, but instead only update the `jiffies` counter every 10th HZ tick. This might be adapted once a driver needs a finer granularity. Furthermore, as device drivers run as independent instances in user space, this can be configured for every device driver separately according to its needs and the `jiffies` counting overhead can even be completely removed for drivers that don't need this time source.

The second way Linux drivers use timing is through the add_timer group of functions that allows to program deferred events. DDEKit/Linux provides an implementation by spawning a dedicated timer thread for every driver instance. This thread manages a list of pending timers and uses a semaphore to block with a timeout until the next timer occurrence should be triggered. If the blocking semaphore acquisition returns with a timeout, the next pending timer needs to be handled by executing the handler function. Otherwise, an external thread has modified the timer list by either adding or removing a timer. In this case the timer thread recalculates the time to sleep until the next trigger and goes back to sleep.

3.6 Memory Management

Running in user space means that DDEKit/Linux may use LibC's malloc and free functions for internal memory management needs. However, this does not suffice for implementing Linux' memory management functions. Linux' kmalloc is internally already implemented using SLABs or one of their equivalents. Our implementation currently provides a specific SLAB implementation in DDEKit, but we plan to use Linux' original memory allocator in the future and only back it with page-granularity memory allocations provided from DDEKit.

Additionally, Linux drivers may use the group of get_free_pages functions to allocate memory with page granularity. DDEKit/Linux supports page granularity allocations through a function that uses mmap in order to allocate page-aligned memory.

A remaining problem is that drivers commonly acquire DMA-able memory in order to allow high amounts of data to be copied without CPU interaction. This is impossible by solely relying on user-level primitives. This means that an implementation of DMA allocation functions such as dma_alloc_coherent requires additional thought. We go on to discuss our solution to this problem in the following section.

4 Attacking the DMA Problem

In order for DMA to or from a memory region to work properly, the region needs to meet three criteria:

1. The region's *physical address* needs to be available as DMA does not use virtual addresses.
2. It needs to be *physically contiguous* so that no virtual-to-physical address translations need to be done during the DMA transfer.
3. It needs to be *pinned*, that is the region or parts of it must not be swapped out during the DMA transfer.

None of these criteria are met by user-level memory allocation routines such as malloc, posix_memalign or mmap, because they work on purely virtual addresses and the underlying kernel is free to map those pages anywhere it wants.

As it is necessary to get kernel support for handling DMA, we implemented a small kernel module providing an interface to the in-kernel DMA API. The module supports two modes: *copy-mode* provides a simple translation layer between user and kernel pages for DMA and *zero-copy mode* facilitates an IOMMU to improve DMA performance.

4.1 Copy-DMA

Our kernel module for supporting DMA from user space closely collaborates with the uio_core as shown in Figure 3. The uio_dma module is notified by the uio_core when a device is bound to it and creates an additional device node /dev/uio-dma which user-level drivers can use to obtain DMA-able memory for a specific device[2].

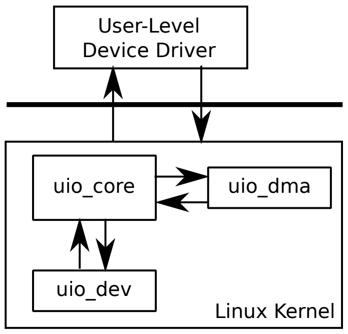

FIGURE 3: *Introducing the uio_dma module*

Linux device drivers can allocate DMA memory either using dma_alloc_coherent or they can request to map DMA memory for a certain buffer in virtual memory and a DMA direction (send/receive) using {dma,pci}_map_single.

[2]The notification is necessary so that the uio_dma module has access to the respective UIO PCI device data structure.

A naïve idea would be to simply implement a device driver that allows allocating DMA memory from the kernel and then use it from user space. However, on many platforms it is possible to use any physical memory for DMA. Therefore, many drivers do not explicitly allocate DMA memory upfront, but instead simply start DMA from arbitrary memory regions, even from their stack. This means the DMA allocator driver would have to provide all dynamic memory allocations for the user space driver. Not only would this circumvent the convenience of managing user space memory using libC's `malloc` and `free`, but it would also decrease the possibility of using GDB with a user space driver, because such kernel memory would not be `ptrace`able.

DDE's implementation of `dma_alloc_coherent` performs an `mmap` on the `uio-dma` device which in turn allocates DMA-able memory in the kernel and establishes a mapping to user space so that upon return from the system call the driver can use this memory area for DMA.

For the `map_single` family of functions an `ioctl` on the `uio-dma` device is used to send a virtual user address and the DMA direction to the kernel module. The system call returns the physical address the driver can then use to initiate DMA.

If upon a DMA_MAP `ioctl` the direction indicates that data shall be sent from user space, the kernel module allocates a DMA-able kernel buffer and copies the user data into this buffer before returning the DMA buffer's physical address. If DMA shall be done from the device into a user buffer, the `ioctl` only allocates a DMA buffer in the kernel and delays copying data from the DMA buffer to the user buffer until the buffer is unmapped using `dma_unmap_single`. It is safe to do so, because only after this function call the DMA can safely be assumed to be finished and therefore the user-level driver should not touch the buffer beforehand anyway.

4.2 DMA With Fewer Copies

While the *copy-DMA* method works without any further support than the one that is already present within the kernel, more recent hardware featuring an IOMMU can be used to get rid of the copying steps between user and kernel buffers.

Copy-DMA uses different kernel- and user-level buffers because it needs to ensure that the kernel buffers that are effectively used for the DMA operation are in fact physically contiguous, which cannot be guaranteed for the user-level buffers.

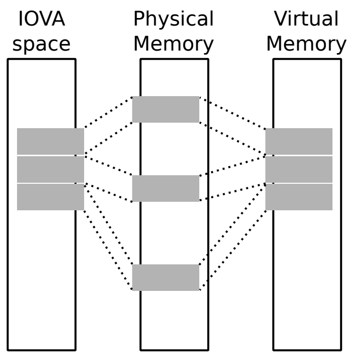

FIGURE 4: *Mapping DMA buffers using an IOMMU*

Using an IOMMU comes to the rescue here. In this case we can use an arbitrary buffer that is virtually contiguous (which includes every buffer allocated using `malloc`). The `uio_dma` module upon encountering the DMA `ioctl` then only needs to run through the list of pages forming the buffer and perform an equally contiguous mapping into the device's IOVA space. Thereafter, the user-level driver can use the IOVA address returned from the `ioctl` call and program DMA without needing to care about physical contiguity.

A minor intricacy arises because of the fact that user-level buffers do not always start and end at a page boundary. This means that multiple DMA buffers may share the same page, so that upon unmapping one of the buffers, the `uio_dma` module cannot safely remove the IOVA mapping as other DMA buffers may still contain the same page. Therefore, `uio_dma` uses reference counting to detect when a page may really be unmapped.

5 Case Studies

There are three interesting questions concerning user-level device drivers:

1. Does running the driver in user space modify the real-time capabilities of a PREEMPT_RT kernel?
2. How does the user space driver's performance compare with an in-kernel driver?
3. Which benefits can be gained by running a driver in user space and using existing profiling and debugging tools?

In this section we try to answer these questions using a real-world example. We downloaded the Linux e1000e network interface driver from the Intel website [6] and compiled it to run in user space.

For all our experiments we used a quad-core Intel Core i7 running at 2.8 GHz with 2 GB of RAM. The operating system was a Linux 3.0.1-rt11 kernel with the PREEMPT_RT option switched on. We tested the e1000e driver with an Intel 82578DC Gb ethernet card.

5.1 Real-Time Operation

To evaluate the influence of running PCI drivers in user space on the system's real-time behavior, we used the `cyclic_test` utility provided by OSADL [20]. Figure 5 shows the maximum latencies for several scenarios we tested.

Each group has four bars corresponding to threads running on the 4 CPUs in our test machine. The group labelled *no_load* shows the latencies for running `cyclic_test` on the idle system running with `idle=poll` to mitigate power management effects. For the group labelled *hi_load* we set each CPU's load to 100% and reran `cyclic_test`. Thereafter, we added network load to the system by running the IPerf UDP benchmark [8] between the test machine and a remote PC. The groups with labels **_e1000e* show the latency for using network through the in-kernel e1000e driver. The groups labelled **_user* give latencies obtained for running the experiment with the e1000e driver in user space using DDE.

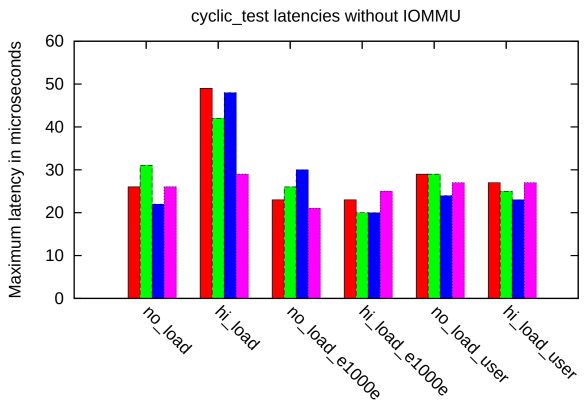

FIGURE 5: *Maximum cyclic_test latencies for the non-IOMMU scenario*

Additionally, we tried to figure out whether turning the machine's IOMMU on or off makes a difference and therefore reran our experiments with the IOMMU turned on. The results for these experiments are shown in Figure 6.

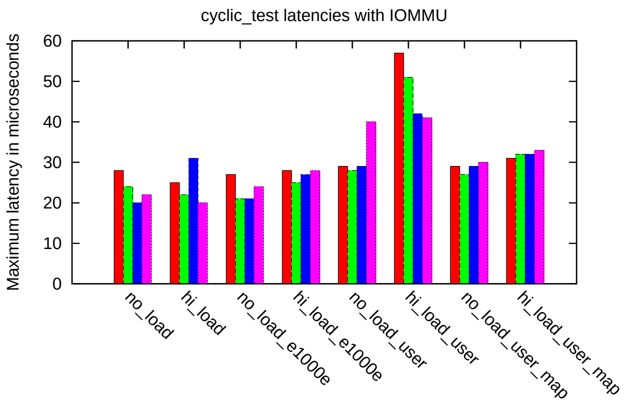

FIGURE 6: *Maximum cyclic_test latencies for the IOMMU scenario*

In addition to the experiments also present in the no-IOMMU case, we added two more bar groups labelled **_user_map*. These groups show maximum latencies obtained when using the no-copy version of the `uio_dma` module.

In both setups we see that the maximum latencies using user-level device drivers are within the bounds of the other measurements. Although we observe a peak in the latency for *hi_load_user*, this peak is within the bounds of the unmodified measurements (e.g., *hi_load* in the previous experiment). We conclude that running device drivers in user space using DDE has no influence on the real-time capabilities of the system.

5.2 UDP/TCP Performance

To evaluate the performance of user-level DDE drivers, we linked our user-level e1000e driver to the lwIP stack [9] and then ran a UDP and TCP throughput benchmark while connected to an external computer. For comparison, we also ran the same benchmark using the in-kernel device driver and the builtin Linux TCP stack.

Figure 7 shows the average and maximum throughputs achieved in these experiments. For UDP it is notable, that even though the network link is a 1Gb NIC, IPerf was only able to saturate 800 MBit/s. Furthermore, user-level and kernel stacks perform equally well. However, the CPU utilization for the user-level driver is higher: the kernel stack ran at about 50% utilization, while the user stack consumed 80%.

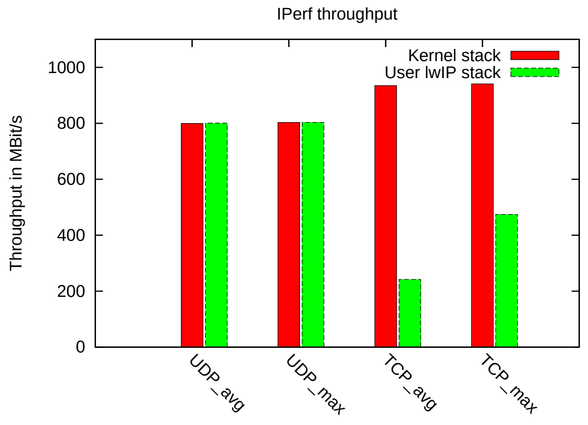

FIGURE 7: *IPerf throughput results*

TCP performance is unfortunately much worse for the user-level TCP stack than with the in-kernel one. We are still investigating these issues and right now attribute this to problems with the lwIP stack.

5.3 Testing, Debugging, and Profiling

So far we showed that user-level device drivers allow for increased isolation at acceptable speed while not influencing the system's real-time capabilities. Another advantage of running them as user applications is the availability of debugging and profiling tools that ease driver development. In this section we introduce two examples where user-level tools could be applied to kernel code and helped us find problems in our implementation of DDE.

Debugging DDE

While working on the user-level e1000e device driver, we experienced hangs in TCP connectivity and started debugging them. With the help of the GDB debugger we were able to figure out that the problem occurred when the driver was in NAPI polling mode and ran out of its network processing budget.

In the Linux kernel, the driver at this point voluntarily reschedules, giving other kernel activity the chance to run. This was improperly implemented within the DDE as it simply returned from the Soft-IRQ handler. In our case however, it would have been necessary to raise the Soft-IRQ again. This did not happen and so the driver went to sleep until it got woken up by the next interrupt.

Profiling DDE

When we initially ran the e1000e driver in user space, performance was by far not as convincing as in the experiments described in Section 5.2.

Using Valgrind's [19] Callgrind profiler, we were able to investigate where the performance went. We were caught by surprise by the result: DDE manages a list of virtual-to-physical mappings for all allocated memory. This list is used to implement the `virt_to_phys` lookup mechanism. This is implemented as a linked list and the assumption was that this would suffice, because there would never be many mappings stored in this list and calls to `virt_to_phys` would take place rather less frequently.

Callgrind's output however told us that this function accounted for a huge amount of execution time. With this knowledge we were able to take a closer look at what regions were registered in this list and found out that in many cases, we did not need to store this information at all, thereby reducing the amount of time spent searching the virt-phys mappings.

6 Related Work

Our work relates to the problem of reusing existing device drivers when designing a new operating system. LeVasseur et al. proposed to use per-driver virtual machines to reuse and isolate device drivers [17]. Poess showed that the DDE approach also works for binary device drivers [22], which is not yet implemented in our system. Friebel implemented a DDE for FreeBSD device drivers which uses the split DDEKit/DDE architecture [10]. Building upon this, it should now also be possible to use FreeBSD device drivers within Linux user space. Boyd-Wickizer's SUD framework [2] also allows running Linux drivers in user space but focusses more on security and isolation and instead of the real-time capabilities of the system.

Our work is also related to Schneider's device driver validation mechanism [23]. By wrapping drivers in user-level applications, we can use the system's native analysis and profiling tools in order to observe driver behavior and identify security violations. The RUMP framework for NetBSD has similar goals as our work and allows debugging and developing device drivers for NetBSD in user space [14]. However, it is not intended to be used for actually running drivers at user space in production systems.

Chipounov proposed to perform heavyweight instruction-level tracing and symbolic execution in order to generate device-specific code that can be dropped into existing per-OS device driver skeletons [3]. While this approach eases device driver reuse, applying it to a real-time kernel has the same drawbacks as native in-kernel drivers in that they still need to be revalidated every time an update is applied.

7 Conclusion

In this paper we presented a Device Driver Environment that allows executing generic Linux in-kernel PCI drivers as user-level applications on top of Linux. This is achieved by implementing the DDE as a wrapper library implementing the facilities expected by in-kernel drivers at user space using off-the-shelf kernel mechanisms such as UIO and *sysfs*. With the help of a small kernel module our framework also supports DMA from user space.

Using this framework, we were able to run the widely used e1000e network interface driver in user space on a PREEMPT_RT kernel. Experiments using cyclic_test showed that the real-time latencies of the system were not influenced by the fact that the driver was running from user space. Furthermore, it was possible to use common Linux program analysis tools such as the GDB debugger and Valgrind to profile and debug drivers.

The DDEKit for Linux is available for download at `http://os.inf.tu-dresden.de/ddekit/`.

Acknowledgments

We'd like to thank several people whose hard work within the recent years has made design and implementation of the Device Driver Environment possible. Thank you, Christian Helmuth, Thomas Friebel, and Dirk Vogt. Carsten Weinhold provided valuable hints on improving this paper.

This work was partially supported by the German Research Association (DFG) within the Special Purpose Program 1500, project title ASTEROID.

References

[1] Linux RT project. `http://www.kernel.org/pub/linux/kernel/projects/rt/`.

[2] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX ATC'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[3] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys '10: Proceedings of the 5th European Conference on Computer Systems*, pages 167–180, New York, NY, USA, 2010. ACM.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[5] Jonathan Corbet. UIO: user-space drivers. `https://lwn.net/Articles/232575/`, 2007.

[6] Intel Corp. Network adapter driver for Gigabit PCI based network connections for Linux. `http://downloadcenter.intel.com`, 2010.

[7] Zheng Da. DDE for GNU/HURD. `http://www.gnu.org/software/hurd/dde.html`.

[8] Jon Dugan and Mitch Kutzko. IPerf TCP/UDP bandwidth benchmark. `http://sourceforge.net/projects/iperf/`, 2011.

[9] Adam Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001:20, SICS – Swedish Institute of Computer Science, February 2001. Master's thesis.

[10] Thomas Friebel. Uebertragung des Device-Driver-Environment-Ansatzes auf Module des BSD-Betriebssystemkerns. Master's thesis, TU Dresden, 2006.

[11] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In *ASPLOS'08: Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–178, Seattle, Washington, USA, March 2008. ACM Press, New York, NY, USA. http://doi.acm.org/10.1145/1346281.1346303.

[12] TU Dresden OS Group. DDE/DDEKit for Fiasco+L4Env. `http://wiki.tudos.org/DDE/DDEKit`, 2006.

[13] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 41–50, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Antti Kantee. Rump device drivers: Shine on you kernel diamond. `http://ftp.netbsd.org/pub/NetBSD/misc/pooka/tmp/rumpdev.pdf`, 2010.

[15] Genode Labs. Genode dde_kit. `http://genode.org/documentation/api/dde\_kit\_index`.

[16] 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.

[17] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *In Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 17–30, 2004.

[18] Martin Mares. PCI Utilities. `http://mj.ucw.cz/pciutils.html`, 2010.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[20] OSADL. Cyclic test utility. `https://www.osadl.org/Realtime-test-utilities-cyclictest-and-s.rt-test-cyclictest-signaltest.0.html`, 2011.

[21] PCI SIG. PCI Local Bus Specification. `http://www.pcisig.com/specifications/conventional/conventional_pci_23/`, 2002.

[22] Bernhard Poess. Binary device driver reuse. Master's thesis, Universitaet Karlsruhe, 2007.

[23] Fred Schneider, Dan Williams, Patrick Reynolds, Kevin Walsh, and Emin Gun Sirer. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation OSDI '08*, December 2008.

[24] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 37(5):207–222, 2003.

[25] Andrew Tanenbaum, Raja Appuswamy, Herbert Bos, Lorenzo Cavallaro, Cristiano Giuffrida, Tomáš Hrubý, Jorrit Herder, Erik van der Kouwe, and David van Moolenbroek. MINIX 3: Status Report and Current Research. *;login: The USENIX Magazine*, 35(3), June 2010.