User-level Device Drivers: Achieved Performance

Ben Leslie[1], Peter Chubb[1], Nicholas Fitzroy-Dale[1], Stefan Götz[2], Charles Gray[1], Luke Macpherson[1], Daniel Potts[1], Yueting Shen[1], Kevin Elphinstone[1], and Gernot Heiser[1]

[1]National ICT Australia and the School of Computer Science and Engineering, University of NSW, Kensington 2052, Australia.
E-mail: firstname.lastname@nicta.com.au
[2]Wilhelm-Schickard-Institute for Computer Science, University of Tübingen, D - 72076 Tübingen, Germany.
E-mail: stefan.goetz@uni-tuebingen.de

Abstract

Running device drivers as unprivileged user-level code, encapsulated into their own process, has often been proposed as a technique for increasing system robustness. However, in the past, systems based on user-level drivers have generally exhibited poor I/O performance. Consequently, user-level device drivers have never caught on to any significant degree. In this paper we demonstrate that it is possible to build systems which employ user-level device drivers, without significant performance degradation, even for high-bandwidth devices such as Gigabit Ethernet.

1 Introduction

Device drivers are one of the most critical parts of every operating system (OS). Traditionally they are implemented as part of the kernel, and thus execute in privileged mode, with full access to all system resources. This simplifies driver implementation and minimises overheads. However, given the proliferation of devices (and classes of different devices) keeping all of them in the kernel is leading to a rapid growth of kernel code. For example, of the roughly 4 million lines of code comprising the Linux kernel, more than 50% are device drivers.

Since the early 1990s, when microkernels were popular, there have been a number of attempts to build systems that run their drivers outside the kernel, as unprivileged user-level code [11, 14]. However, none of those early attempts has made much lasting impact. Typically researchers have found that the extra context switches introduced by user-level drivers lead to significant overall performance degradation. As a consequence, user-level drivers remain the exception in mainstream systems. In general, they are used only for devices where performance is not critical or the number of context switches is small compared to the work the driver does (the Linux X server is an example).

In this paper we make a new case for user-level drivers, and demonstrate the feasibility of user-level drivers via an implementation in Linux. We also show that the

I/O performance achieved with user-level drivers can approach that of Linux with traditional in-kernel drivers. Our results show that it is worthwhile taking another serious look at running drivers as unprivileged user-level code.

The remainder of this paper is structured as follows. Section 2 justifies the use of out-of-kernel drivers by examining their advantages. Section 3 discusses issues in supporting user-level device drivers, and presents our user-level driver framework for Linux. Section 4 presents an evaluation of the performance of the framework compared with normal Linux. Section 5 discusses related work, which is followed by conclusions and planned future work.

2 Motivation

There are good reasons why one might want to run device drivers at user level. An important one is *ease of development*: If a driver is a normal user process, it can be debugged and profiled like any other user program. Additionally, fault source identification is greatly simplified when faulty drivers are removed from the kernel. In-kernel driver faults can cause kernel malfunctions in unrelated kernel components, and so identifying the original source of the fault can be extremely difficult.

Maintainability arguments also strongly favour user-level drivers. In particular in a system like Linux, where the kernel and its internal interfaces changes rapidly, keeping drivers (which depend on those interfaces) consistent with the kernel is difficult and error-prone. A recent study examined the use of global variables in the Linux kernel source, and the dependencies introduced by them [21]. It found that the number of dependencies of the core kernel on other kernel code (mostly drivers) is increasing *exponentially* with the kernel version. The authors concluded that this will eventually lead to Linux becoming unmaintainable.

A user-level driver API does not share global variables with the kernel. In consequence, drivers coded to such an API would be more portable across kernel releases and would minimise the dependency of the core kernel on their correct behaviour.

While in-kernel drivers are on the one hand able to bypass the API and can break abstractions, they must, on the other hand, adhere to a *restrictive programming model*. In traditional Unices and Linux, for example, drivers are written in C, because the runtime support for other languages is not available in-kernel. It is possible to port enough of the C++ runtime environment to allow a subset of C++ to be used, but the increased complexity of driver plus environment increases the effort involved to a prohibitive degree. User-level drivers could be written in any language that can bind to the exported API, including even high-level scripting languages. This has the potential to greatly simplify driver development, at least for drivers whose performance is not critical.

While handling an interrupt, a driver has an even more restricted programming model — it runs on the stack of the process that was interrupted, and it may not block. This requires very careful resource management to avoid unfairly blocking the current process, or deadlocking the kernel. User-level drivers run in their own context avoiding the issue of blocking in the interrupt handler and simplifying deadlock prevention.

A further advantage of user-level drivers is that their *release schedule can be decoupled* from that of the kernel. In-kernel drivers have to be compiled for a particular kernel; when the kernel is updated, the end-user has to either recompile the driver, or obtain a new one from the vendor. If the driver is in user space, it depends only on the user-driver API, and standard Linux and POSIX APIs, and so the same driver binary

can continue to be used.

More recent work on user-level drivers [27] was motivated by performance considerations, typically in the context of high-speed networks. With such networks, performance is often limited by the overheads associated with the mode switches required for the communication between in-kernel drivers and user-level clients. Performing all device handling in the client's address space reduces overheads, provided the drivers use polling rather then interrupt-driven I/O. In general, this can be done securely only with specialised NICs providing support for secure user-level drivers, or in situations where it is acceptable for a single application to monopolise a device. We view this application of user-level drivers as complimentary motivation, but not the focus of our approach.

Some of the software-engineering advantages of user-level drivers could be realised by supporting user-level drivers as well as traditional in-kernel drivers. For example, drivers could be developed and debugged at user level and integrated into the kernel at a later time (possibly for improved performance). This has the potential of improving the quality and reliability even of drivers in the kernel, if they have originally been developed for the (stricter) user-level API.

However, keeping all drivers at user level has potential benefits for *system stability*. A recent study of Linux kernel code has shown that the defect density of device drivers is three to seven times that of other parts of the kernel [8]. This is both a reflection of the inherent complexity of driver code and the fact that much of it is written by people not necessarily very familiar with the internal operating system structure. The resulting negative impact on system reliability is not restricted to open source systems: it has been reported that device drivers were responsible for 27% of crashes in Windows 2000, compared to only 2% for the rest of the kernel [25].

A crashing in-kernel driver often takes the rest of the system down with it, but a system may be able to survive a crashed user-level driver by restarting it. The arguments made in favour of recursive restartability [5] apply fully to user-level drivers. This includes better control over resource consumption (and protection against resource leaks) as normal OS resource management can be applied to user-level drivers as to any user process. Hence, user-level drivers have the potential to improve system reliability.

A fundamental problem with in-kernel drivers is that *they must be trusted completely* by the system. As the Stanford study [8] shows, this trust is totally unjustified. With the present convergence of computing and entertainment this is likely to get worse. People regularly buy additional peripherals for their computers, be it a wireless mouse, joy-stick or a digital camera, and load the drivers supplied with the gadget into their system, thereby implicitly (and mostly unknowingly) trusting the manufacturer of the device (and whoever the manufacturer may have out-sourced driver development to). While we are not aware of any cases of malicious device drivers yet, it is probably only a matter of time until they appear, and systems are poorly guarded against such threats. Manufacturer certification of drivers is unlikely to help, as it is unreasonable to expect that any manufacturer can really make any guarantees on the correctness of non-trivial drivers.

One possible solution to the problem of trusting dubious code is to *ensure* that the code can be trusted. This can be achieved by the use of type-safe languages [1], transactions [23], software-based fault isolation [28] or proof-carrying code [17]. These techniques are, however, not directly applicable to device drivers, which interface directly to hardware. The complex hardware interfaces, featuring plenty of side-effects, and complications such as devices executing scripts from memory, are beyond the state of the art of such software techniques (although such methods may become feasible

for driver verification one day). Furthermore, it seems unrealistic to expect all drivers to be re-written in a type-safe language or to the restrictive coding standards that could make some of these techniques applicable. Finally, it is difficult with such software techniques to address the resource management problems of in-kernel drivers, while user-level drivers enable the use of standard resource-management techniques.

The alternative solution is to accept that drivers are untrustworthy and equip the system with mechanisms for dealing with this fact. An obvious prerequisite for untrusted device drivers is that drivers execute without kernel privileges and outside the kernel's address space, i.e., at user level. By itself, however, isolating the device driver from the kernel is not enough to enable completely untrusted device drivers. Firstly, the system itself relies on device drivers for its operation (booting, swapping etc.) We do not address this issue here, other than by noting that there are ways to solve it. A system could have essential user-level drivers statically linked into the boot image. Furthermore, the SUNDR [16] approach of encoding and signing file data in order to avoid trusting the file server could be extended to untrusted disk and network drivers.

Secondly, device drivers performing direct memory access can bypass the OS protection mechanism even when running at user level. This is a problem which can be solved by using more sophisticated I/O hardware, which is already available in some commercial computer systems [9]. Taking advantage of such hardware completely encapsulates the driver and allows it to access only explicitly authorised memory and resources. Again, utilisation of such hardware support is not subject of this paper.

For the remainder of this paper, we focus on the performance aspects of user-level device drivers, as adequate performance is a prerequisite for the widespread adoption of user-level device drivers. Note that we do not tackle the problem of protecting the system from errant driver DMA, we leave this aspect for future work. We believe that encapsulating drivers (at user-level) without restricting DMA still provides many of the benefits motivating user-level drivers, such as improved robustness and enforced modularity. In addition, others have demonstrated a 99% reduction in driver-induced system crashes when drivers are encapsulated without DMA protection, but still run in privileged mode [24]. Our approach provides stronger encapsulation by running drivers in user-mode.

3 Design & Implementation

In this section we discuss the design and implementation of our user-level device drivers for Linux. We first identify and discuss general issues that arise in supporting device drivers at user level as compared with approaches taken to support drivers in traditional monolithic kernels. For each issue, we then describe how we resolve it in our user-level driver support for Linux.

3.1 Interrupts

An *interrupt service routine* (ISR) is responsible for reacting quickly and efficiently to device events. It is invoked almost directly via a hardware-defined exception mechanism that interrupts the current flow of processor execution and enables the potential return to that flow after completion of the ISR.

In general, the length of the ISR should be minimised so as to maximise the burst rate of device events that can be achieved, to reduce ISR invocation latency of all ISRs (assuming they are mutually exclusive), and to minimise the overall CPU processing

required for a given number of device events. Any additional per-interrupt processing can have a negative impact on all three aspects of interrupt processing. Gigabit Ethernet hardware can easily generate events at a rate that traditional in-kernel drivers struggle to process, and thus the hardware usually provides a mechanism to limit the interrupts generated. Such high interrupt-rate devices illustrate that there is little scope for significantly increasing in interrupt processing overhead.

3.1.1 Linux User-level Interrupt Delivery

The normal way to hook a driver onto an interrupt inside a kernel driver is to call `request_irq()` with a callback function. The kernel then sets up its internal tables to call the ISR when an interrupt arrives. ISRs steal the kernel stack from the process running when the interrupt arrives; they are fairly limited in what they can do.

There are several ways user processes could be informed of interrupt events: map an interrupt onto a signal; use an event queue, similar to that used in the USB input device support; or allow a user thread to wait for an interrupt by reading a special file.

Signals are the slowest mechanism, due to requiring an extra context switch, while the other two should be similar. The third mechanism was chosen since it was the simplest to implement.

Information about interrupts is exported in vanilla Linux 2.6.*x* via `/proc`; there is a subdirectory `/proc/irq/`*n*`/` for each handled interrupt. We extended this so that *all* potential interrupts have a directory. When a user process opens `/proc/irq/`*n*`/irq`, a ten-line interrupt handler is installed for interrupt *n.* At `close()` time, the handler is deinstalled. (All file descriptors are closed when a process dies, so if the driver dies, the interrupt handler will be deinstalled. An example of simplified resource management enabled by user-level device drivers). Access control is presently very unsophisticated: any process running as root can install an interrupt handler, provided that no handler is presently registered for the interrupt.

When an interrupt occurs, the in-kernel handler disables the interrupt and increments a per-interrupt semaphore.

When the user process does a `read()` on the file, the value of the semaphore, if non-zero, is returned. Otherwise the interrupt is enabled, and the process sleeps on the semaphore. Once an interrupt is received, the driver does whatever is necessary to acknowledge the interrupt on the card.

3.2 Device Mapping and DMA

Device drivers require access to the registers of devices they manage. In-kernel drivers potentially have access to all I/O ports and physical memory, including device registers. Drivers running at user-level require a method for obtaining access to device registers and memory. On Linux the */dev/mem* file provides access to the physical memory. Device drivers call `mmap()` to map the appropriate section of */dev/mem* into their address space.[1]

Bus-mastering DMA-based devices access physical memory directly using I/O-bus addresses, which may be physical memory addresses or some mapping of I/O-bus addresses to physical addresses. DMA-based devices create two issues: translation and pinning.

[1] This mechanism was already available in Linux.

Translation Buffers specified by user-level applications are identified using virtual addresses. DMA-capable devices require these addresses to be translated into I/O-bus addresses. This translation can be done simply and quickly by an in-kernel device driver by accessing the page tables stored within the kernel address space. Additionally, some drivers deal only with the kernel address space and can directly use kernel addresses (or some fixed offset) for DMA.

The Linux kernel provides functions (`pci_map_`*xxx*) to map (`struct page *`, `offset`, `len`) tuples to (`busaddr`, `len`) tuples. We added to Linux the means to access these functions from user-level via a new system call. In addition to performing the mapping, the system call keeps track of the mappings so that when the user-level process crashes or exits they can be torn down cleanly.

In the case where the Linux kernel generates and receives DMA-based transactions to and from user-level drivers, the addresses can be readily converted by the kernel into or from I/O bus addresses to alleviate the need for any translation within the driver. This is the scenario for the encapsulated block and network user-level device drivers we developed, which are described in Section 3.4.1 and 3.4.2.

Pinning DMA-capable devices access physical memory directly without any mediation via a MMU. Coordination between the page replacement policy and the device driver is required to avoid the situation where a page is swapped out and the underlying frame is recycled for another purpose while an outstanding DMA is yet to complete. Preventing pages from being swapped out is generally termed *pinning* the page in memory. Pinning is handled by in-kernel bookkeeping that indicates to the page replacement algorithm that the frame is pinned.

User-level drivers of DMA-based devices also require pinning. Pinning functionality could be made available by a system call API to update the in-kernel bookkeeping. Such an API should be combined with the API for translation to minimise system calls. One can envisage a single system call that unpins the current I/O buffer, pins the next I/O buffer, and supplies the buffer's I/O bus address for DMA.

Some architectures do possess I/O MMUs which could potentially simplify this issue by disabling (and postponing) I/O-bus access to replaced pages.

As was the case for translation, our particular user-level block and network drivers rely on the Linux kernel providing pinned I/O buffers, hence no pinning functionality was required for our framework.

3.3 Generating and Consuming Work

A device driver provides an interface to allow the kernel to direct the driver to perform work. A driver also expects an interface provided by the kernel in order to hand off work to the kernel or signal completion of activities. The following sections discuss these interfaces that are important to driver performance.

3.3.1 Providing Work to the Driver

Drivers provide an interface for the kernel to enqueue work to be performed by the device. This involves passing the driver a work descriptor that describes the work to be performed. The descriptor may be a data structure (or a reference to one) or arguments to a function call. The work descriptor identifies the operation and any data (buffers) required to perform the work. Drivers and the kernel share the kernel address space which enables fast transfer (by reference) and access to descriptors and buffers.

In moving drivers to user level in Linux, work descriptors can still be efficiently delivered to the driver by sharing memory between the kernel and the user-level driver. There are obvious concurrency issues in managing data structures in shared memory. We make use of lock-free queues (implemented with circular buffers) which allow work to be enqueued for a driver without requiring explicit interaction with the driver on every operation. This encourages a batching effect where several local lock-free operations follow each other, and finally the recipient driver is activated.

3.3.2 Offloading Work

Drivers also produce work for components of the kernel. A common example is a network driver receiving packets and therefore generating work for an IP stack. Like enqueueing work for the driver itself, an efficient mechanism is required in the reverse direction to enqueue work for, and activate (or continue) a kernel component such as an IP stack. When in-kernel, work descriptors and buffers can be handled in a similar manner to enqueueing work for the driver, i.e. descriptors and buffers can be transferred and accessed directly in the kernel's address space. Once work is enqueued for a kernel component, the component requires activation via a synchronisation primitive. Again, the primitive can rely on the shared kernel address space to mark a component runnable and place it on the appropriate scheduler queue.

A user-level driver can also offload work to (or continue work in) the kernel by enqueing descriptors in a shared memory region, and then activating kernel activity via a system call.

3.4 Linux Specifics

Up until this point, we have discussed user-level device driver support in Linux generally. The following two sections describe specifics of supporting a user-level IDE driver and a user-level network driver.

3.4.1 Block loopback

In Linux there is no standard way for a user-level program to act as a block device. We implemented a small kernel module which allows user-level device drivers to hook into the standard Linux block layer. The module presents a filesystem that has pairs of directories: a master and a slave. When the filesystem is mounted, creating a file in the master directory creates a set of block device special files, one for each potential partition, in the slave directory.

The file in the master directory can then be used to communicate via a very simple protocol between a user-level block device and the kernel's block layer. The block-device special files in the slave directory can then be opened, closed, read, written or mounted, just as any other block device.

In this model the user-level block device is passed bus addresses, which can be used without translation when performing DMA, rather than needing to call back into the kernel to pin and translate the address.

It is important to note that this module is a generic block device module, and is not tied to the specific device drivers we have implemented.

3.4.2 Network loopback

Unlike block devices, Linux has an existing mechanism to allow user-level programs to act as a network device and inject packets into the standard protocol stack. To use this, user-level programs open a special device and perform an `ioctl()` call to register as a new network device in the system. Packets are then transferred between the program and kernel using the standard `read()/write()` file interface. Initially we used this interface for our network device, however as expected the performance was quite poor due to the excessive copying of packet data.

To avoid this problem we created a more efficient interface which avoids the copying overhead. We modelled the interaction between the kernel and the user-level device driver on the standard network-device to driver interaction. On initialisation we setup a transmit ring and receive ring in an area of shared memory.[2] When the packets need to be transmitted, the kernel module pins them in memory and adds a descriptor to the transmit ring. In the same way, when the driver receives a packet, it is placed onto the receive ring. The network loopback structure is shown in Figure 1

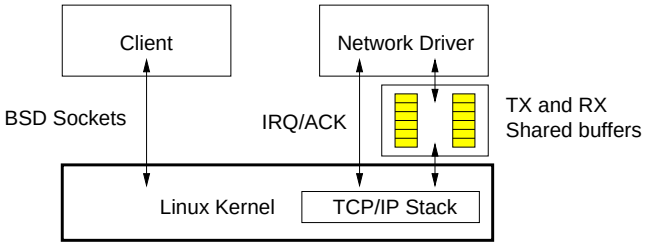

Figure 1: Linux network loopback structure

To inform the user-level device driver that work is available, we simulate an interrupt, by incrementing a semaphore associated with the device driver's interrupt file descriptor. The kernel checks for received packets each time the user-level device drivers returns from an interrupt handler.

4 Evaluation & Results

We chose IDE disk and Gigabit ethernet as the devices to experimentally evaluate our device drivers and infrastructure to run them at user-level. We chose disk as it is a moderate device in terms of data bandwidth it produces and consumes, and the rate of interrupts it generates. Gigabit Ethernet was chosen as it is a high bandwidth device with potentially very high interrupt generation rates that are a challenge for conventionally structured operating systems to cope with. As such, they represent a good target for evaluating our driver's performance under high loads. Success in efficiently supporting Gigabit Ethernet would make our approach suitable for a very wide variety of devices.

For all our experiments we used a HP ZX2000, 900 MHz Itanium 2, with 1 GB RAM, and Linux 2.6.6 as the operating system platform. For the disk benchmarks we used a Maxtor Diamondmax Plus 9 80GB ATA-133 7200rpm disk, for the network benchmarks we used an Intel PRO/1000 (82545GM) Gigabit Ethernet controller.

[2]The format of the receive and transmit ring is independent of the underlying device's receive and transmit ring.

4.1 Disk

To evaluate disk read performance, we developed a simple benchmarking application that reads a 64 MB contiguous range of 512 byte sectors from the start of the disk. We measure the elapsed time (using the CPU cycle counter) for each benchmark run and calculate throughput in MiB/s (1 MiB = 2^{20} bytes). During the benchmark, a low priority idle process is used to measure available CPU to calculate CPU utilisation during the benchmark. The idle thread is implemented using a tight loop which calculates the difference between the current and last cycle count values (Δt), and adds this to a running total. If Δt is greater than the maximum number of cycles required to perform one iteration of the loop, accounting for TLB and cache misses, the result is accounted to a context switch, and therefore not included in the total.

The benchmark was repeated and varied such that each benchmark run used a different requested transfer size. The transfer size used initially was one sector, and then it was increased a sector at a time (1, 2, 3, 4, etc..) in each subsequent run until 32 sectors are requested for each transfer, after which the transfers are increased by increasingly larger increments until 8192 sectors[3]. Each run at a particular requested transfer size was repeated 10 times.

We measured the following configurations:

Linux Kernel The benchmark runs as a normal application on the Linux kernel with our IDE driver in kernel[4]. The benchmark uses `/dev/hda` to bypass the file system and access the drive directly. The device is opened using `O_DIRECT` to avoid unnecessary kernel copying by DMA-ing directly into the benchmarking application's buffer.

Linux User This configuration is the same as the Linux Kernel benchmark, except that our IDE driver runs as a user-level application on the system. The IDE driver application accepts and responds to requests from the Linux kernel as described in Section 3.4.1

Figure 2 shows these benchmark results. The lines beginning at the bottom left corner and rising to the top right corner represent achieved disk throughput (in MiB/s), while the downward-sloping lines represent measured CPU utilisation. When comparing the results for Linux in-kernel versus the user-level configuration, we see almost indistinguishable results for the throughput curves. The CPU utilisation curves are also almost indistinguishable when the requested transfer size is 128 sectors or greater. Below 128 sectors we see that the extra overhead of context switching between the benchmark application and the user-level IDE driver manifests itself as a small increase in CPU utilisation, which it the worst case is a 17% increase in CPU utilisation (52% to 61%) for the single sector transfer request size.

4.2 Network

In this section we evaluate our user-level driver framework in the networking context. Running the device driver for a Gigabit Ethernet controller at user level is likely to have

[3]The precise transfer sizes beyond 32 were 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384, 448, 512, 640, 768, 896, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584, 4096, 5120, 6144, 7168, and 8192 sectors

[4]We replaced the standard Linux IDE driver with ours to ensure a fair comparison as the standard driver's performance was sufficiently below ours to significantly bias the results against the in-kernel case.

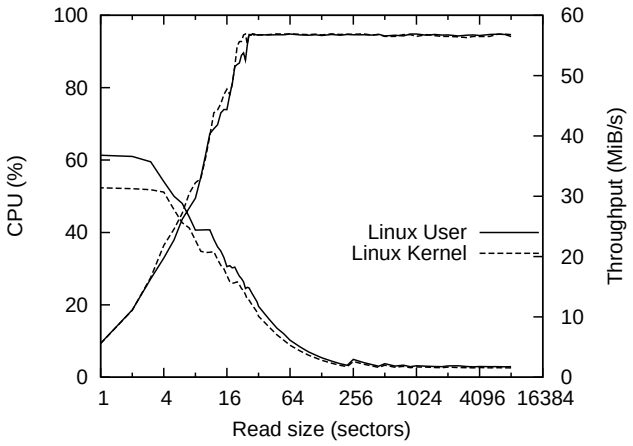

Figure 2: IDE Read throughput and CPU utilisation

the most detrimental affect on performance, and thus this section contains the results most critical to the practicability of user-level drivers.

The following configurations are compared in this scenario:

Linux Kernel This configuration uses our driver[5] in-kernel together with the Linux IP stack. All benchmarking software required on the machine under testing runs as a normal application in a process.

Linux User This configuration uses our driver running encapsulated as a Linux process. The IP stack remains within the Linux kernel, and communicates with the driver application via the specialised file descriptor interface. Benchmarking software required on the machine under test runs as a normal application in another process. This configuration was described in detail in Section 3.4.2 and Figure 1.

The benchmark we used to compare in-kernel versus user-level network driver performance was UDP-echo. We chose UDP echo as a benchmark as it consists almost entirely of driver and IP stack processing, and thus will exaggerate any differences in driver overheads between the two configurations. The setup we used is shown in figure 3. We used a locally-developed *ipbench*[6] software suite to perform the benchmarks.

Each client, an identically configured 2.5GHz Celeron with ample memory and two network interfaces, applies a load of 1024 byte UDP packets on the target machine which echoes the contents of the packet back to the senders. The senders count the successful replies to determine the achieved echo throughput. The combined applied load is varied from 10 Mb/s to above 900 Mb/s[7]. CPU utilisation is monitored on the target machine using our standard method described in Section 4.1. In one configuration, the target machine has our driver in kernel, in the other configuration, our driver runs at user-level.

Figure 4 shows the UDP echo throughput and CPU utilisation results. The solid lines are results from the driver in-kernel; the dashed lines are the results from the driver

[5]Our network driver was found to have identical performance to the standard Linux E1000 driver. However, we still used our driver in-kernel to avoid any bias in the experiments.

[6]Ipbench is available at http://ipbench.sourceforge.net

[7]The 1000 Mb/s load is beyond wire speed and results in some packets being dropped by the switch.

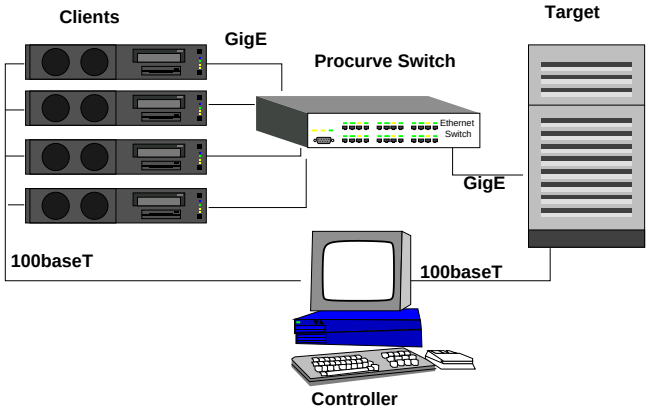

Figure 3: Network benchmarking setup

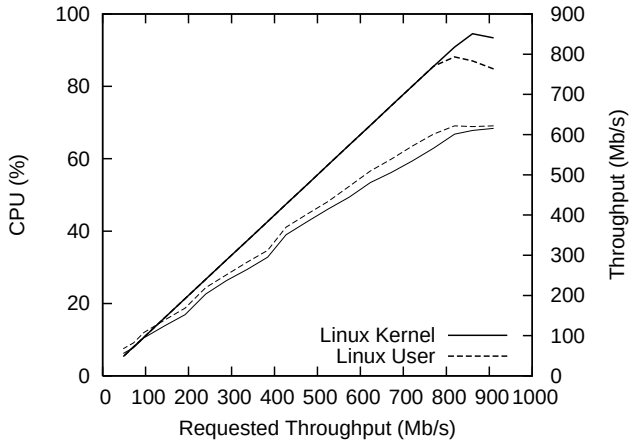

Figure 4: UDP echo throughput and CPU utilisation

running at user-level. The lower pair of lines represent CPU utilisation (the left scale) at each requested throughput level (the lower scale). The upper pair of lines represent achieved echo throughput (the right scale) at each level of requested throughput.

Examining achieved throughput, we see that both the in-kernel and user-level driver configurations achieve the requested throughput up until high loads. The in-kernel driver peaks at 850 Mb/s throughput, after which throughput drops slightly with increasing applied load. In the user-level configuration, the throughput peaks at 790 Mb/s prior to dropping off with increasing applied load. The user-level driver configuration achieves 93% of the achieved throughput of the in-kernel configuration.

Examining the CPU utilisation under load, we can see the extra CPU overhead we introduce by running the driver at user-level. Surprisingly, the largest difference in CPU utilisation is 4% of the overall CPU capacity at a load of 720 Mb/s. This represents a 7% increase in CPU utilisation.

This outstanding result can be attributed to both the carefully constructed and highly optimised interface between the user-level driver and the Linux kernel, and the hardware mechanisms provided by the Ethernet controller to reduce the number

of interrupts generated by packet transmission and reception.

4.3 Summary

For moderate bandwidth and interrupt rate devices like disk, we see that running device drivers at user-level results in small increases in CPU utilisation in micro-benchmarks. For fast networks with high packet rates, we see that hardware-provided interrupt reduction mechanisms result in very small differences in throughput and CPU utilisation (7%) when comparing user-level with in-kernel device drivers.

5 Related Work

Liedtke's L3 system employed user-level device drivers since 1988 [15] and is probably the first system to have done so. L3 allowed drivers to access device registers and used IPC messages to deliver interrupts to the driver. DMA was restricted to trusted drivers, other user-level drivers did not have to be trusted. While performance was claimed to be good, the only data given was a comparison of IPC cost with disk block transfer times, as well as some figures on serial port performance. In particular, no performance data was provided for network interfaces. Furthermore, L3 used an in-kernel driver for the hard disk (as this was required for booting and used DMA and thus needed to be trusted anyway). Consequently it is not possible to draw any conclusions from the literature on the real performance of L3 user-level drivers.

Mach 3 used a similar model to support user-level drivers [11]. Interestingly, the main motivation was performance: Mach previously lacked a zero-copy interface to kernel drivers, and zero-copy could be achieved by running the device driver in the same address space as the client (protocol stack of file system server, both presumably trusted subsystems). Other reasons given for user-level drivers were preemptability, location transparency and easier code sharing between related drivers. The authors reported significant improvements in Ethernet throughput, which was, presumably, a result of removing data copying between driver and client. Throughput improvements were also reported for SCSI disks. It appears that most performance gains in Mach 3 resulted from implementing device drivers cleanly, and avoiding unnecessary copying. No comparisons were done with monolithic systems, and no information on CPU utilisation (which might give an indication of the communication-induced overhead) was provided. It is well known that Mach had significantly worse overall performance than monolithic Unix systems [6], hence it is difficult to draw general conclusions on the performance of user-level drivers from the Mach experience.

Golub et al. [12] implemented a different user-level driver model in Mach 3, also with the aim of sharing code, as part of the IBM Workspace OS project. Part of their motivation was to support running concurrent operating systems on the same microkernel. Their model actually leaves the lowest-level parts of the drivers (what in Linux lingo would be called the "top half driver") in the kernel, dynamically loaded by the (trusted) bottom-half driver. The two parts of the driver communicate via shared memory and a semaphore. DMA control was also completely done by the kernel. No performance data was published.

The Raven kernel [20] implemented a device driver model similar to Mach 3 [11], also motivated by the elimination of copying between kernel and user space. Interrupts are delivered to drivers as an upcall. In order to ensure that critical sections remain atomic, a global variable is used to indicate to the kernel that the driver is holding a lock. The driver must be trusted to follow the protocol and inform the kernel when

releasing the lock. The authors measured interrupt latencies and found that invoking a user-level driver was significantly more costly than a kernel driver. They expected this overhead to be amortised by not copying data, but presented no results on overall I/O performance.

Hunt [13] implemented user-level drivers for Windows NT. The motivation for the work was easing driver development by enabling driver development in the application execution environment, instead of the restricted and complex driver environment that exists in-kernel in NT. The user-level drivers are supported by an in-kernel driver proxy that redirects I/O request packets (IRPs) from the device proxy registered in the kernel, to the user-level driver for servicing. The design suffers from significant performance degradation, and does not support drivers requiring access to hardware.

A user-level driver model also exists for the Fluke kernel [26]. The main motivation for user-level drivers in Fluke was to reconcile the execution environment of the Fluke kernel (a non-blocking kernel) with that of legacy drivers. The in-kernel environment was unsuitable for driver execution, leading to the natural solution of moving drivers to user mode where the driver execution environment can be emulated. The obtained performance results showed up to a 100% increase in CPU overhead. The culprits identified were IPC performance of the Fluke kernel, and overheads of the runtime environment (marshaling costs).

Schaelicke [22] proposes a user-level I/O hardware architecture. The architecture provides secure access to device hardware at user-level via specialist hardware. The motivation being to remove the operating system from the path to the hardware and deal directly with the hardware itself. Briefly, the architecture provides a safe method to enqueue requests directly to the device, an I/O TLB for protection and enabling direct DMA to applications, and a light-weight notification mechanism. His simulation results show a significant performance improvement of a 100-fold reduction in CPU overhead. We view this work as complimentary, and believe our framework could take advantage of such hardware if available.

Similarly, Pratt [19] proposes *user-safe devices* using similar hardware proposals (safe application access to hardware and in-device memory translation) to remove the operating system from the application-to-hardware path. However, the motivation is this case is to improve quality of service by removing the operating-system overhead in device management which has proved difficult to account for.

Work at the University of Washington [24] attempts to encapsulate device drivers by introducing protection domains, called *nooks*, within the kernel's address space. This has the advantage of potentially fewer changes required to existing drivers; however, the authors found that tight integration of Linux drivers with the kernel made this approach difficult. A nook is essentially a user-level process with read-only access to kernel data. This helps when running unmodified legacy drivers, but compared to a well-designed driver API that minimises communication provides marginal performance advantages. They reported a significant performance degradation in both throughput and CPU utilsation for network. Compared to our approach *nooks* uses significantly more lines of code.[8]. The *nooks* wrapper code is also highly depedant on the Linux kernel internals which given the rapid change of Linux will make it very hard to maintain.

Reducing network latency, particularly for high-performance computing applications, has been the main motivation for work on user-level network drivers. The idea is

[8] 22,226 lines of kernel code in Nooks compared to 2058 lines of kernel code in our user-level driver framework

to perform device management in the client's address space and thus remove the need to perform mode or context switches for network processing. This approach has been used for connection-oriented networks, such as ATM [10, 27], and special-purpose networks [2–4]. Removing the kernel from all packet handling generally requires a switch from an interrupt-driven to a polling I/O model, which only makes sense if the network is saturated, or the application has some knowledge on when the NIC is likely to be ready. In order to maintain protection the NICs must provide support for multiplexing packets securely between different clients (as is the case with connection-oriented networks, or NICs that support OS-installed filters [18]). Our approach is more general in that it neither gives up interrupt-driven I/O nor is it limited to special kinds of devices.

The *Palladium* approach of running Linux kernel extensions at an intermediate privilege level [7] could, in principle, be used for device drivers without significant performance impact. While this approach could protect the kernel (to a degree) from buggy drivers, it would not protect applications, which still run at a lower privilege level.

6 Conclusions

Operating systems that run device drivers at user-level stand to benefit in many ways, such as improved maintainability, flexibility, stability, driver fault recovery and resource management. The perceived significant performance penalty in running drivers at user-level has discouraged the adoption of this approach in general, a perception that seemed warranted given the past work exploring this approach.

We have investigated the performance of user-level device drivers by modifying Linux to enable the running of network and disk device drivers at user-level. We have compared the performance user-level drivers with the same drivers running in Linux kernel itself. Our results show the following.

- For modest bandwidth and interrupt rate devices such as hard drives, the system-level performance penalty for running device drivers at user-level is insignificant.
- High interrupt rate devices such as Gigabit Ethernet cause significant interrupt-related overhead even for in-kernel device drivers. Hardware-provided mechanisms for reducing interrupt rates (and hence overhead) can be equally applied to user-level device drivers to mitigate some of the extra overhead of invoking drivers running at user-level. When using interrupt hold-off, we see modest increases in CPU utilisation, and little change in system throughput when comparing user-level to in-kernel drivers.

We cannot attribute the good performance to a simple set of factors compared to previous user-level driver frameworks. The good performance has resulted from a systematic and thorough approach to designing and implementing a framework that avoids extra copying, heavily uses shared structures to deliver and consume work, avoids synchronisation where possible via lock-free structures, batches work into minimise notification events between components, and heavily optimises the event notification mechanisms. The batching effect created by the interrupt hold-off mechanisms of modern network controllers also contributed.

We believe we have made a strong case for running device drivers at user-level and demonstrated that doing so is not significantly detrimental to performance.

The potential benefits of running device drivers at user-level are great, as argued in Section 2, and when combined with little or no performance penalty, strongly argues

for their wide-spread adoption in general purpose operating systems. In the case of open-source systems, such as Linux, there is a further advantage: user-level device drivers provide protection of intellectual property of device manufacturers. User-level device drivers can be distributed binary-only, without violating the licensing conditions of the GPL-ed kernel.

7 Future Work

We will continue to refine our user-level driver support for Linux and investigate the performance of more complex workloads in the presence of multiple user-level device drivers. The goal is a Linux system with all drivers running at user-level, with little negative affects on performance.

Some platforms now provide I/O memory management units that can be used to protect main memory from device driver DMA. We plan to explore the integration of this kind of protection within our framework to completely encapsulate and isolate drivers from the rest of the system.

Acknowledgements

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

References

[1] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, pages 267–284, Copper Mountain, CO, USA, Dec 1995.

[2] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena H. Mesarina. Virtual-memory-mapped network interfaces. *IEEE Micro*, 15(1):21–28, 1995.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, Feb 1995.

[4] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Manlyn sender-managed interface architecture. In *2nd OSDI*, pages 245–259, Seattle, WA, USA, Oct 1996.

[5] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *8th HotOS*, pages 125–130, 2001.

[6] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *14th SOSP*, pages 120–133, Asheville, NC, USA, Dec 1993.

[7] Tzi-cher Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th SOSP*, pages 140–153, Kiawah Island, SC, USA, Dec 1999.

[8] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[9] Digital Equipment Corporation. *DIGITAL Semiconductor 21174 Core Logic Chip Technical Reference Manual*, 1997.

[10] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM*, pages 2–13, Sep 1994.

[11] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *USENIX Mach Symp.*, 1991.

[12] David B. Golub, Guy G. Sotomayor, Jr, and Freeman L. Rawson III. An architecture for device drivers executing as user-level tasks. In *USENIX Mach III Symp.*, pages 153–171, 1993.

[13] Galen C. Hunt. Creating user-mode device drivers with a proxy. In *1st USENIX Windows NT WS*, 1997.

[14] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *Operat. Syst. Rev.*, 25(2):51–62, Apr 1991.

[15] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *Operat. Syst. Rev.*, 25(2):51–62, Apr 1991.

[16] David Mazières and Dennis Shasha. Don't trust your file server. In *8th HotOS*, pages 113–118, Elmau, Germany, May 2001.

[17] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd OSDI*, pages 229–243, Seattle, WA, USA, Oct 1996.

[18] Ian Pratt and Keir Fraser. Arsenic: A user-accessible Gigabit Ethernet interface. In *20th INFOCOM*, Apr 2001.

[19] Ian A. Pratt. *The User-Safe Device I/O Architecture*. PhD thesis, King's College, University of Cambridge, Aug 1997.

[20] D. Stuart Ritchie and Gerald W. Neufeld. User level IPC and device management in the Raven kernel. In *2nd USENIX WS Microkernels & other Kernel Arch.*, pages 111–125, San Diego, CA, USA, Sep 1993.

[21] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux kernel. *IEE Proc.: Softw.*, 149:18–23, 2002.

[22] Lambert Schaelicke. *Architectural Support for User-Level I/O*. PhD thesis, University of Utah, 2001.

[23] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *2nd OSDI*, pages 213–228, Nov 1996.

[24] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *19th SOSP*, Bolton Landing (Lake George), New York, USA, Oct 2003.

[25] Michael M. Swift, Steven Marting, Henry M. Levy, and Susan G. Eggers. Nooks: An architecture for reliable device drivers. In *10th SIGOPS Eur. WS*, pages 101–107, St Emilion, France, Sep 2002.

[26] Kevin Thomas Van Maren. The Fluke device driver framework. MSc thesis, University of Utah, Dec 1999.

[27] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *15th SOSP*, pages 40–53, Copper Mountain, CO, USA, Dec 1995.

[28] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *14th SOSP*, pages 203–216, Asheville, NC, USA, Dec 1993.