Get More Device Drivers out of the Kernel!

Peter Chubb*
National ICT Australia
and
The University of New South Wales
peterc@gelato.unsw.edu.au

Abstract

Now that Linux has fast system calls, good (and getting better) threading, and cheap context switches, it's possible to write device drivers that live in user space for whole new classes of devices. Of course, some device drivers (Xfree, in particular) have always run in user space, with a little bit of kernel support. With a little bit more kernel support (a way to set up and tear down DMA safely, and a generalised way to be informed of and control interrupts) almost any PCI bus-mastering device could have a user-mode device driver.

I shall talk about the benefits and drawbacks of device drivers being in user space or kernel space, and show that performance concerns are not really an issue—in fact, on some platforms, our user-mode IDE driver out-performs the in-kernel one. I shall also present profiling and benchmark results that show where time is spent in in-kernel and user-space drivers, and describe the infrastructure I've added to the Linux kernel to allow portable, efficient user-space drivers to be written.

*This work was funded by HP, National ICT Australia, the ARC, and the University of NSW through the Gelato programme (http://www.gelato.unsw.edu.au)

1 Introduction

Normal device drivers in Linux run in the kernel's address space with kernel privilege. This is not the only place they can run—see Figure 1.

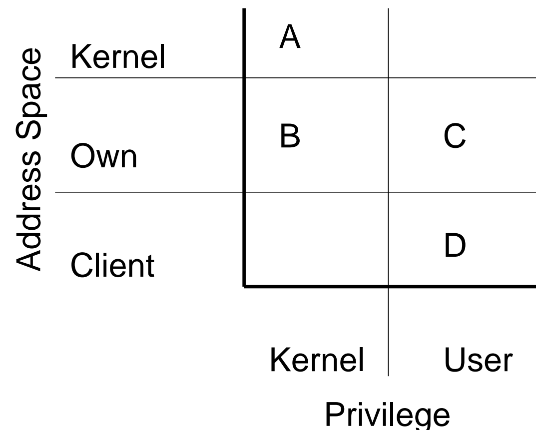

Figure 1: Where a Device Driver can Live

Point A is the normal Linux device driver, linked with the kernel, running in the kernel address space with kernel privilege.

Device drivers can also be linked directly with the applications that use them (Point B)—the so-called 'in-process' device drivers proposed by [Keedy, 1979]—or run in a separate process, and be talked to by an IPC mechanism (for example, an X server, point D).

They can also run with kernel privilege, but with a separate kernel address space (Point C) (as in the Nooks system described by [Swift et al., 2002].

2 Motivation

Traditionally, device drivers have been developed as part of the kernel source. As such, they *have* to be written in the C language, and they have to conform to the (rapidly changing) interfaces and conventions used by kernel code. Even though drivers can be written as modules (obviating the need to reboot to try out a new version of the driver[1]), in-kernel driver code has access to all of kernel memory, and runs with privileges that give it access to all instructions (not just unprivileged ones) and to all I/O space. As such, bugs in drivers can easily cause kernel lockups or panics. And various studies (e.g., [Chou et al., 2001]) estimate that more than 85% of the bugs in an operating system are driver bugs.

Device drivers that run as user code, however, can use any language, can be developed using any IDE, and can use whatever internal threading, memory management, etc., techniques are most appropriate. When the infrastructure for supporting user-mode drivers is adequate, the processes implementing the driver can be killed and restarted almost with impunity as far as the rest of the operating system goes.

Drivers that run in the kernel have to be updated regularly to match in-kernel interface changes. Third party drivers are therefore usually shipped as source code (or with a compilable stub encapsulating the interface) that has to be compiled against the kernel the driver is to be installed into.

This means that everyone who wants to run a third-party driver also has to have a toolchain and kernel source on his or her system, or obtain a binary for their own kernel from a trusted third party.

Drivers for uncommon devices (or devices that the mainline kernel developers do not use regularly) tend to lag behind. For example, in the 2.6.6 kernel, there are 81 drivers known to be broken because they have not been updated to match the current APIs, and a number more that are still using APIs that have been deprecated.

User/kernel interfaces tend to change much more slowly than in-kernel ones; thus a user-mode driver has much more chance of not needing to be changed when the kernel changes. Moreover, user mode drivers can be distributed under licences other than the GPL, which may make them more attractive to some people[2].

User-mode drivers can be either closely or loosely coupled with the applications that use them. Two obvious examples are the X server (XFree86) which uses a socket to communicate with its clients and so has isolation from kernel and client address spaces and can be very complex; and the Myrinet drivers, which are usually linked into their clients to gain performance by eliminating context switch overhead on packet reception.

The Nooks work [Swift et al., 2002] showed that by isolating drivers from the kernel address space, the most common programming errors could be made recoverable. In Nooks, drivers are insulated from the rest of the kernel

[1] except that many drivers currently cannot be unloaded

[2] for example, the ongoing problems with the Nvidia graphics card driver could possibly be avoided.

by running each in a separate address space, and replacing the driver ↔ kernel interface with a new one that uses cross-domain procedure calls to replace any procedure calls in the ABI, and that creates shadow copies of any shared variables in the protected address space of the driver.

This approach provides isolation, but also has problems: as the driver model changes, there is quite a lot of wrapper code that has to be changed to accommodate the changed APIs. Also, the value of any shared variable is frozen for the duration of a driver ABI call. The Nooks work is uniprocessor only; locking issues therefore have not yet been addressed.

Windriver [Jungo, 2003] allows development of user mode device drivers. It loads a proprietary device module `/dev/windrv6`; user code can interact with this device to setup and teardown DMA, catch interrupts, etc.

Even from user space, of course, it is possible to make your machine unusable. Device drivers have to be trusted to a certain extent to do what they are advertised to do; this means that they can program their devices, and possibly corrupt or spy on the data that they transfer between their devices and their clients. Moving a driver to user space does not change this. It does however make it less likely that a fault in a driver will affect anything other than its clients

3 Existing Support

Linux has good support for user-mode drivers that do not need DMA or interrupt handling — see, e.g., [Nakatani, 2002].

The `ioperm()` and `iopl()` system calls allow access to the first 65536 I/O ports; and, with a patch from Albert Calahan[3] one can map the appropriate parts of */proc/bus/pci/...* to gain access to memory-mapped registers. Or on some architectures it is safe to `mmap()` */dev/mem*.

It is usually best to use MMIO if it is available, because on many 64-bit platforms there are more than 65536 ports—the PCI specification says that there are 2^{32} ports available—(and on many architectures the ports are emulated by mapping memory anyway).

For particular devices—USB input devices, SCSI devices, devices that hang off the parallel port, and video drivers such as XFree86—there is explicit kernel support. By opening a file in */dev*, a user-mode driver can talk through the USB hub, SCSI controller, AGP controller, etc., to the device. In addition, the *input* handler allows input events to be queued back into the kernel, to allow normal event handling to proceed.

libpci allows access to the PCI configuration space, so that a driver can determine what interrupt, IO ports and memory locations are being used (and to determine whether the device is present or not).

Other recent changes—an improved scheduler, better and faster thread creation and synchronisation, a fully preemptive kernel, and faster system calls—mean that it is possible to write a driver that operates in user space that is almost as fast as an in-kernel driver.

4 Implementing the Missing Bits

The parts that are missing are:

[3] `http://lkml.org/lkml/2003/7/13/258`

1. the ability to claim a device from user space so that other drivers do not try to handle it;
2. The ability to deliver an interrupt from a device to user space,
3. The ability to set up and tear-down DMA between a device and some process's memory, and
4. the ability to loop a device driver's control and data interfaces into the appropriate part of the kernel (so that, for example, an IDE driver can appear as a standard block device), preferably without having to copy any payload data.

The work at UNSW covers only PCI devices, as that is the only bus available on all of the architectures we have access to (IA64, X86, MIPS, PPC, alpha and arm).

4.1 PCI interface

Each device should have only a single driver. Therefore one needs a way to associate a driver with a device, and to remove that association automatically when the driver exits. This has to be implemented in the kernel, as it is only the kernel that can be relied upon to clean up after a failed process. The simplest way to keep the association and to clean it up in Linux is to implement a new filesystem, using the PCI namespace. Open files are automatically closed when a process exits, so cleanup also happens automatically.

A new system call, `usr_pci_open(int bus, int slot, int fn)` returns a file descriptor. Internally, it calls `pci_enable_device()` and `pci_set_master()` to set up the PCI device after doing the standard filesystem boilerplate to set up a vnode and a `struct file`.

Attempts to open an already-opened PCI device will fail with `-EBUSY`.

When the file descriptor is finally closed, the PCI device is released, and any DMA mappings removed. All files are closed when a process dies, so if there is a bug in the driver that causes it to crash, the system recovers ready for the driver to be restarted.

4.2 DMA handling

On low-end systems, it's common for the PCI bus to be connected directly to the memory bus, so setting up a DMA transfer means merely pinning the appropriate bit of memory (so that the VM system can neither swap it out nor relocate it) and then converting virtual addresses to physical addresses.

There are, in general, two kinds of DMA, and this has to be reflected in the kernel interface:

1. Bi-directional DMA, for holding scatter-gather lists, etc., for communication with the device. Both the CPU and the device read and write to a shared memory area. Typically such memory is uncached, and on some architectures it has to be allocated from particular physical areas. This kind of mapping is called *PCI-consistent*; there is an internal kernel ABI function to allocate and deallocate appropriate memory.
2. Streaming DMA, where, once the device has either read or written the area, it has no further immediate use for it.

I implemented a new system call[4], `usr_pci_map()`, that does one of three things:

[4] Although multiplexing system calls are in general deprecated in Linux, they are extremely useful while developing, because it is not necessary to change every architecture-dependent *entry.S* when adding new functionality

1. Allocates an area of memory suitable for a PCI-consistent mapping, and maps it into the current process's address space; or

2. Converts a region of the current process's virtual address space into a scatterlist in terms of virtual addresses (one entry per page), pins the memory, and converts the scatterlist into a list of addresses suitable for DMA (by calling `pci_map_sg()`, which sets up the IOMMU if appropriate), or

3. Undoes the mapping in point 2.

The file descriptor returned from `usr_pci_open()` is an argument to `usr_pci_map()`. Mappings are tracked as part of the private data for that open file descriptor, so that they can be undone if the device is closed (or the driver dies).

Underlying `usr_pci_map()` are the kernel routines `pci_map_sg()` and `pci_unmap_sg()`, and the kernel routine `pci_alloc_consistent()`.

Different PCI cards can address different amounts of DMA address space. In the kernel there is an interface to request that the dma addresses supplied are within the range addressable by the card. The current implementation assumes 32-bit addressing, but it would be possible to provide an interface to allow the real capabilities of the device to be communicated to the kernel.

4.2.1 The IOMMU

Many modern architectures have an IO memory management unit (see Figure 2), to convert from physical to I/O bus addresses—in much the same way that the processor's MMU converts virtual to physical addresses—allowing even thirty-two bit cards to do single-cycle DMA to anywhere in the sixty-four bit memory address space.

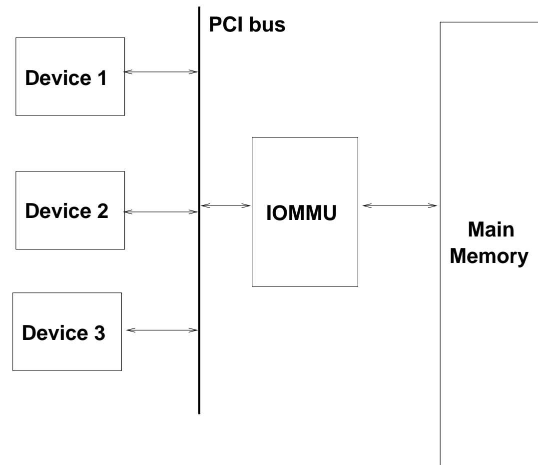

Figure 2: The IO MMU

On such systems, after the memory has been pinned, the IOMMU has to be set up to translate from bus to physical addresses; and then after the DMA is complete, the translation can be removed from the IOMMU.

The processor's MMU also protects one virtual address space from another. Currently shipping IOMMU hardware does not do this: all mappings are visible to all PCI devices, and moreover for some physical addresses on some architectures the IOMMU is bypassed.

For fully secure user-space drivers, one would want this capability to be turned off, and also to be able to associate a range of PCI bus addresses with a particular card, and disallow access by that card to other addresses. Only thus could one ensure that a card could perform DMA only into memory areas explicitly allocated to it.

4.3 Interrupt Handling

There are essentially two ways that interrupts can be passed to user level.

They can be mapped onto signals, and sent asynchronously, or a synchronous 'wait-for-signal' mechanism can be used.

A signal is a good intuitive match for what an interrupt *is*, but has other problems:

1. One is fairly restricted in what one can do in a signal handler, so a driver will usually have to take extra context switches to respond to an interrupt (into and out of the signal handler, and then perhaps the interrupt handler thread wakes up)
2. Signals can be slow to deliver on busy systems, as they require the process table to be locked. It would be possible to short circuit this to some extent.
3. One needs an extra mechanism for registering interest in an interrupt, and for tearing down the registration when the driver dies.

For these reasons I decided to map interrupts onto file descriptors. */proc* already has a directory for each interrupt (containing a file that can be written to to adjust interrupt routing to processors); I added a new file to each such directory. Suitably privileged processes can open and read these files. The files have open-once semantics; attempts to open them while they are open return -1 with `EBUSY`.

When an interrupt occurs, the in-kernel interrupt handler masks just that interrupt in the interrupt controller, and then does an `up()` operation on a semaphore.

When a process reads from the file, then kernel enables the interrupt, then calls `down()` on a semaphore, which will block until an interrupt arrives.

The actual data transferred is immaterial, and in fact none ever is transferred; the `read()` operation is used merely as a synchronisation mechanism.

Obviously, one cannot share interrupts between devices if there is a user process involved. The in-kernel driver merely passes the interrupt onto the user-mode process; as it knows nothing about the underlying hardware, it cannot tell if the interrupt is *really* for this driver or not. As such it always reports the interrupt as 'handled'.

This scheme works only for level-triggered interrupts. Fortunately, all PCI interrupts are level triggered.

If one really wants a signal when an interrupt happens, one can arrange for a `SIGIO` using `fcntl()`.

It may be possible, by more extensive rearrangement of the interrupt handling code, to delay the end-of-interrupt to the interrupt controller until the user process is ready to get an interrupt. As masking and unmasking interrupts is slow if it has to go off-chip, delaying the EOI should be significantly faster than the current code. However, interrupt delivery to userspace turns out not to be a bottleneck, so there's not a lot of point in this optimisation (profiles show less than 0.5% of the time is spent in the kernel interrupt handler and delivery even for heavy interrupt load — around 1000 cycles per interrupt)

5 Driver Structure

The user-mode drivers developed at UNSW are structured as a preamble, an interrupt thread, and a control thread (see Figure 3).

The preamble:

1. Uses *libpci.a* to find the device or devices it is meant to drive,

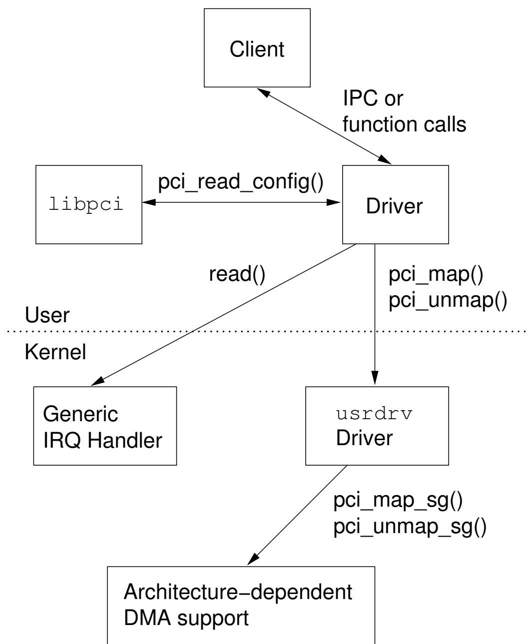

Figure 3: Architecture of a User-Mode Device Driver

2. Calls `usr_pci_open()` to claim the device, and
3. Spawns the interrupt thread, then
4. Goes into a loop collecting client requests.

The interrupt thread:

1. Opens */proc/irq/***irq***/irq*
2. Loops calling `read()` on the resulting file descriptor and then calling the driver proper to handle the interrupt.
3. The driver handles the interrupt, calls out to the control thread(s) to say that work is completed or that there has been an error, queues any more work to the device, and then repeats from step 2.

For the lowest latency, the interrupt thread can be run as a real time thread. For our benchmarks, however, this was not done.

The control thread queues work to the driver then sleeps on a semaphore. When the driver, running in the interrupt thread, determines that a request is complete, it signals the semaphore so that the control thread can continue. (The semaphore is implemented as a pthreads mutex).

The driver relies on system calls and threading, so the fast system call support now available in Linux, and the NPTL are very important to get good performance. Each physical I/O involves at least three system calls, plus whatever is necessary for client communication: a `read()` on the interrupt FD, calls to set up and tear down DMA, and maybe a `futex()` operation to wake the client.

The system call overhead could be reduced by combining DMA setup and teardown into a single system call.

6 Looping the Drivers

An operating system has two functions with regard to devices: firstly to drive them, and secondly to abstract them, so that all devices of the same class have the same interface. While a standalone user-level driver is interesting in its own right (and could be used, for example, to test hardware, or could be linked into an application that doesn't like sharing the device with anyone), it is much more useful if the driver can be used like any other device.

For the network interface, that's easy: use the tun/tap interface and copy frames between the driver and */dev/net/tun*. Having to copy slows things down; others on the team here are planning to develop a zero-copy equivalent of tun/tap.

For the IDE device, there's no standard Linux way to have a user-level block device, so I im-

plemented one. It is 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 bloc 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.

I didn't bother implementing ioctl; it was not necessary for our performance tests, and when the driver runs at user level, there are cleaner ways to communicate out-of-band data with the driver, anyway.

7 Results

Device drivers were coded up by [Leslie and Heiser, 2003] for a CMD680 IDE disc controller, and by another PhD student (Daniel Potts) for a DP83820 Gigabit ethernet controller. Daniel also designed and implemented the tuntap interface.

7.1 IDE driver

The disc driver was linked into a program that read 64 Megabytes of data from a Maxtor 80G disc into a buffer, using varying read sizes. Measurements were also made using Linux's in-kernel driver, and a program that read 64M of data from the same on-disc location using `O_DIRECT` and the same read sizes.

We also measured write performance, but the results are sufficiently similar that they are not reproduced here.

At the same time as the tests, a low-priority process attempted to increment a 64-bit counter as fast as possible. The number of increments was calibrated to processor time on an otherwise idle system; reading the counter before and after a test thus gives an indication of how much processor time is available to processes other than the test process.

The initial results were disappointing; the user-mode drivers spent far too much time in the kernel. This was tracked down to `kmalloc()`; so the `usr_pci_map()` function was changed to maintain a small cache of free mapping structures instead of calling `kmalloc()` and `kfree()` each time (we could have used the slab allocator, but it's easier to ensure that the same cache-hot descriptor is reused by coding a small cache ourselves). This resulted in the performance graphs in Figure 4.

The two drivers compared are the new CMD680 driver running in user space, and Linux's in-kernel SIS680 driver. As can be seen, there is very little to choose between them.

The graphs show average of ten runs; the standard deviations were calculated, but are negligible.

Each transfer request takes five system calls to do, in the current design. The client queues work to the driver, which then sets up DMA for the transfer (system call one), starts the transfer, then returns to the client, which then sleeps on a semaphore (system call two). The interrupt thread has been sleeping in `read()`, when the controller finishes its DMA, it cause an interrupt, which wakes the interrupt thread (half of system call three). The interrupt thread then tears down the DMA (system call four), and starts any queued and waiting activity, then signals the semaphore (system call five) and

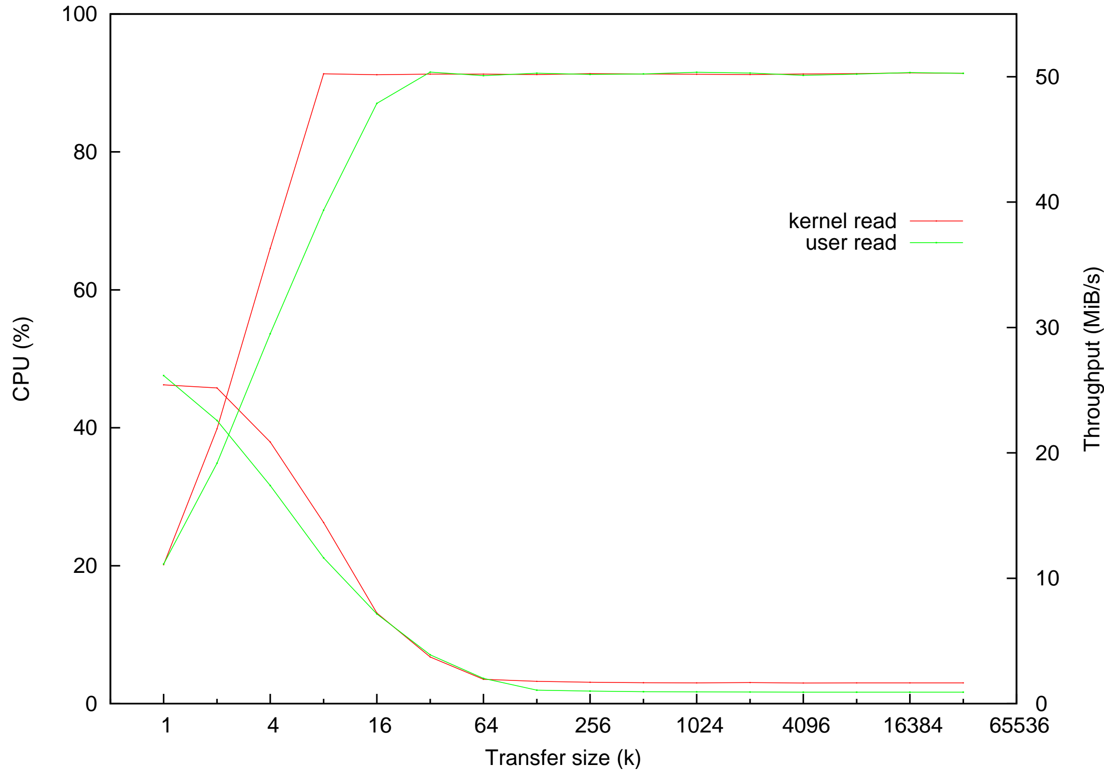

Figure 4: Throughput and CPU usage for the user-mode IDE driver on Itanium-2, reading from a disk

goes back to read the interrupt FD again (the other half of system call three).

When the transfer is above 128k, the IDE controller can no longer do a single DMA operation, so has to generate multiple transfers The Linux kernel splits DMA requests above 64k, thus increasing the overhead.

The time spent in this driver is divided as shown in Figure 5.

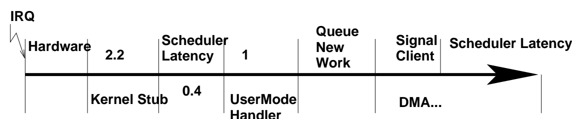

Figure 5: Timeline (in μseconds)

7.2 Gigabit Ethernet

The Gigabit driver results are more interesting. We tested these using [ipbench, 2004] with four clients, all with pause control turned off. We ran three tests:

1. Packet receive performance, where packets were dropped and counted at the layer immediately above the driver
2. Packet transmit performance, where packets were generated and fed to the driver, and
3. Ethernet-layer packet echoing, where the protocol layer swapped source and destination MAC-addresses, and fed received packets back into the driver.

We did not want to start comparing IP stacks, so none of these tests actually use higher level protocols.

We measured three different configurations: a standalone application linked with the driver, the driver looped back into */dev/net/tap* and the standard in-kernel driver, all with interrupt holdoff set to 0, 1 or 2. (By default, the normal kernel driver sets the interrupt holdoff to 300 μseconds, which led to too many packets being dropped because of FIFO overflow) Not all tests were run in all configurations—for example the linux in-kernel packet generator is sufficiently different from ours that no fair comparison could be made.

For the tests that had the driver residing in or feeding into the kernel, we implemented a new protocol module to count and either echo or drop packets, depending on the benchmark.

In all cases, we used the amount of work achieved by a low priority process to measure time available for other work while the test was going on.

The throughput graphs in all cases are the same. The maximum possible speed on the wire is given for raw ethernet by $10^9 \times p/(p+38)$ bits per second (the parameter 38 is the ethernet header size (14 octets), plus a 4 octet frame check sequence, plus a 7 octet preamble, plus a 1 octet start frame delimiter plus the minimum 12 octet interframe gap; p is the packet size in octets). For large packets the performance in all cases was the same as the theoretical maximum. For small packet sizes, the throughput is limited by the PCI bus; you'll notice that the slope of the throughput curve when echoing packets is around half the slope when discarding packets, because the driver has to do twice as many DMA operations per packet.

The user-mode driver ('Linux user' on the graph) outperforms the in-kernel driver ('Linux orig')—not in terms of throughput, where all the drivers perform identically, but in using *much* less processing time.

This result was so surprising that we repeated the tests using an EEpro1000, purportedly a card with a much better driver, but saw the same effect—in fact the achieved echo performance is worse than for the in-kernel ns83820 driver for some packet sizes.

The reason appears to be that our driver has a fixed number of receive buffers, which are reused when the client is finished with them—they are allocated only once. This is to provide congestion control at the lowest possible level—the card drops packets when the upper layers cannot keep up.

The Linux kernel drivers have an essentially unlimited supply of receive buffers. Overhead involved in allocating and setting up DMA for these buffers is excessive, and if the upper layers cannot keep up, congestion is detected and the packets dropped in the protocol layer—after significant work has been done in the driver.

One sees the same problem with the user mode driver feeding the tuntap interface, as there is no feedback to throttle the driver. Of course, here there is an extra copy for each packet, which also reduces performance.

7.3 Reliability and Failure Modes

In general the user-mode drivers are very reliable. Bugs in the drivers that would cause the kernel to crash (for example, a null pointer reference inside an interrupt handler) cause the driver to crash, but the kernel continues. The driver can then be fixed and restarted.

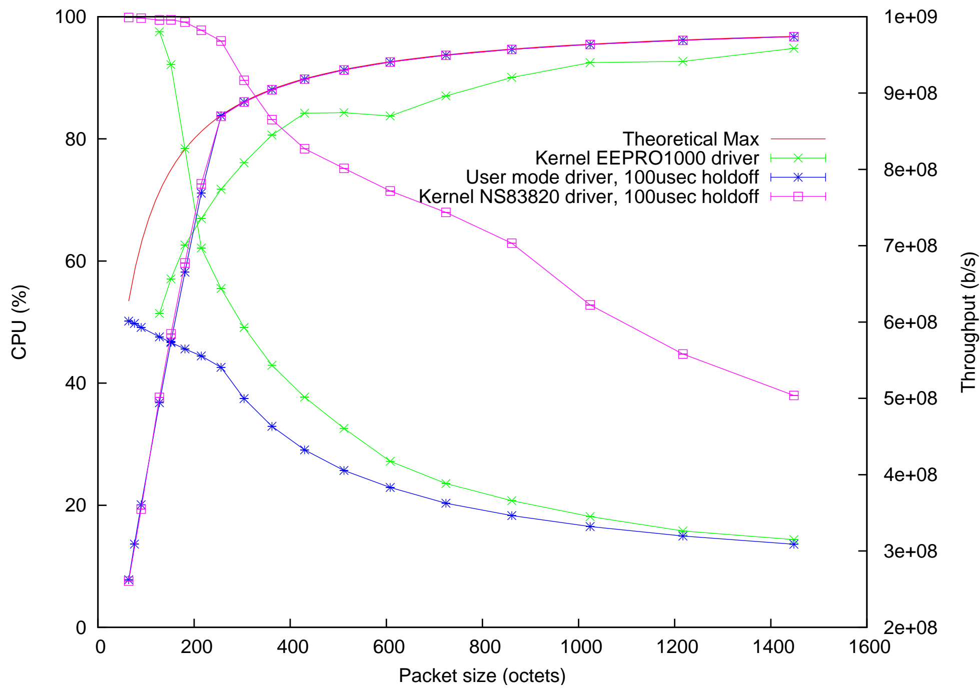

Figure 6: Receive Throughput and CPU usage for Gigabit Ethernet drivers on Itanium-2

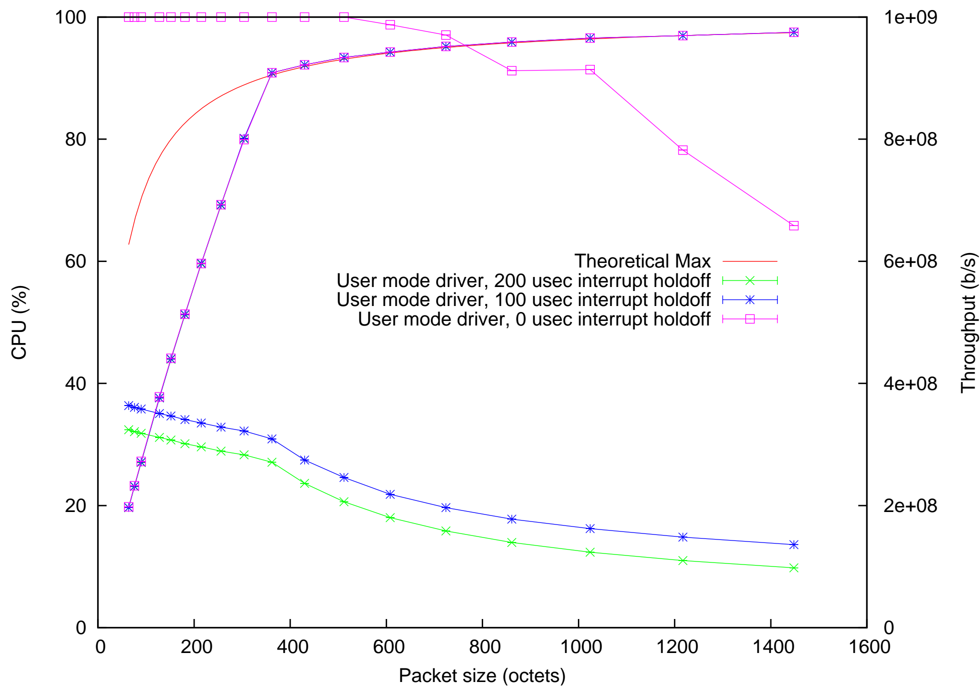

Figure 7: Transmit Throughput and CPU usage for Gigabit Ethernet drivers on Itanium-2

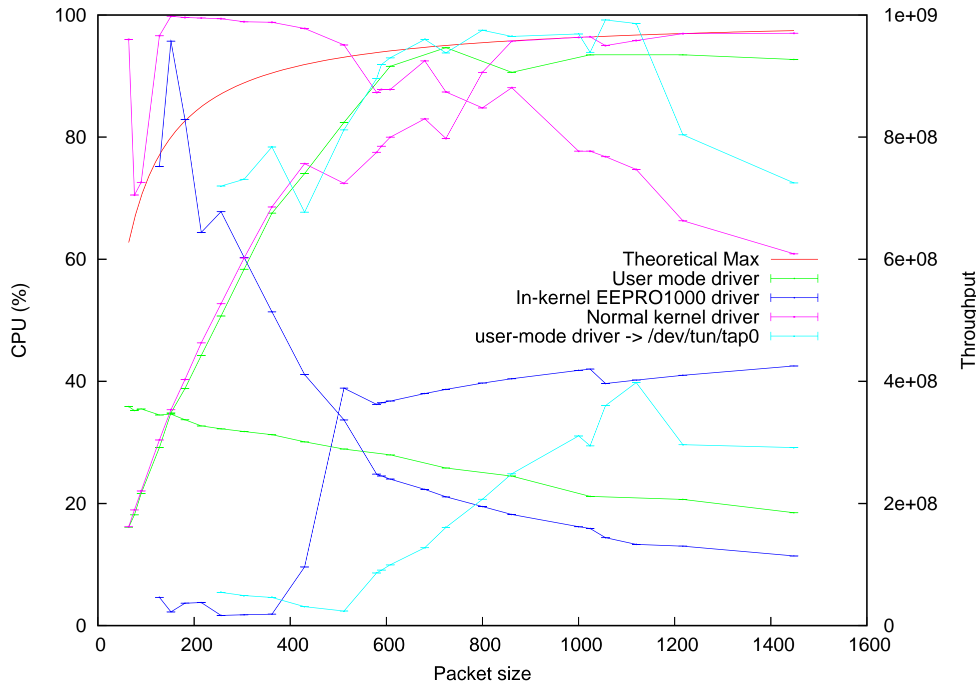

Figure 8: MAC-layer Echo Throughput and CPU usage for Gigabit Ethernet drivers on Itanium-2

8 Future Work

The main foci of our work now lie in:

1. Reducing the need for context switches and system calls by merging system calls, and by trying new driver structures.
2. A zero-copy implementation of tun/tap.
3. Improving robustness and reliability of the user-mode drivers, by experimenting with the IOMMU on the ZX1 chipset of our Itanium-2 machines.
4. Measuring the reliability enhancements, by using artificial fault injection to see what problems that cause the kernel to crash are recoverable in user space.
5. User-mode filesystems.

In addition there are some housekeeping tasks to do before this infrastructure is ready for inclusion in a 2.7 kernel:

1. Replace the ad-hoc memory cache with a proper slab allocator.
2. Clean up the system call interface

9 Where d'ya Get It?

Patches against the 2.6 kernel are sent to the Linux kernel mailing list, and are on `http://www.gelato.unsw.edu.au/patches`

Sample drivers will be made available from the same website.

10 Acknowledgements

Other people on the team here did much work on the actual implementation of the user level drivers and on the benchmarking infrastructure. Prominent among them were Ben Leslie

(IDE driver, port of our dp83820 into the kernel), Daniel Potts (DP83820 driver, tuntap interface), and Luke McPherson and Ian Wienand (IPbench).

References

[Chou et al., 2001] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. (2001). An empirical study of operating systems errors. In *Symposium on Operating Systems Principles*, pages 73–88. http://citeseer.nj.nec.com/article/chou01empirical.html.

[ipbench, 2004] ipbench (2004). ipbench — a distributed framework for network benchmarking.

ipbench.sf.net.

[Jungo, 2003] Jungo (2003). Windriver.

www.jungo.com/windriver.html.

[Keedy, 1979] Keedy, J. L. (1979). A comparison of two process structuring models. MONADS Report 4, Dept. Computer Science, Monash University.

[Leslie and Heiser, 2003] Leslie, B. and Heiser, G. (2003). Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, Operating Systems and Distributed Systems Group, School of Computer Science and Engineering, The University of NSW. CSE techreports website, ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0303.pdf.

[Nakatani, 2002] Nakatani, B. (2002). ELJOnline: User mode drivers.

http://www.linuxdevices.com/articles/AT5731658926.html.

[Swift et al., 2002] Swift, M., Martin, S., Leyand, H. M., and Eggers, S. J. (2002). Nooks: an architecture for reliable device drivers. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France.