Towards Untrusted Device Drivers

Ben Leslie and Gernot Heiser

UNSW-CSE-TR-0303

March, 2003

disy@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia

Abstract

Device drivers are well known to be one of the prime sources of unreliability in today's computer systems. We argue that this need not be, as drivers can be run as user-level tasks, allowing them to be encapsulated by hardware protection. In contrast to prior work on user-level drivers, we show that on present hardware it is possible to prevent DMA from undermining this encapsulation. We show that this can be done without unreasonably impacting driver performance.

1 Introduction

Device drivers are of critical importance in any computer system. Not only are drivers essential for performing I/O; since they normally execute in privileged mode, they are also critical to the stability and reliability of the system, as any driver bug can damage the integrity of the system.

A recent study of bugs in Linux kernel code showed that the defect density of device drivers is three to seven times that of other parts of the kernel [CYC+01]. 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 [SMLE02].

This instability of device drivers has led many researchers to try and find methods in which the impact of a faulty device driver can be minimised. A common way of doing this is to move device drivers out of the kernel into user land where they are confined by normal memory protection hardware. Although this approach is suitable for some simple devices, it fails to adequately isolate devices that can perform direct memory access (DMA). This report shows how the features found in modern platforms can be used to better isolate devices capable of performing DMA.

An initial design and implementation of device drivers along these lines is described in [Les02]. Here we summarise the aspects of this work relevant to reducing trust in device drivers and present some performance results.

2 I/O MMUs

Currently the most popular I/O bus found in computers, ranging from entry level desktops through to high-end servers, is the PCI bus. The base PCI specification defines a 32-bit address space which devices can access using DMA. Unfortunately this has proved a limitation on modern 64-bit platforms which are capable of using more than 4Gb of physical memory.[1]

To avoid this problem, modern architectures provide an I/O memory management unit (MMU), which provides a mapping between 64-bit physical addresses and 32-bit PCI DMA addresses.

The I/O MMU works in a similar way to a normal MMU. The translations are stored in a page table in the host's main memory. When a device performs DMA the translation is looked up in the host's page table. If there is no valid translation available, a fault is raised and the DMA is cancelled. If successful, an I/O *translation look-aside buffer* (IOTLB) caches valid translations, so a page table lookup is not required for every DMA.

We can restrict the areas of memory that a device can perform DMA to by limiting the number of entries in the I/O page table to those explicitly requested by

[1]The PCI specification does provide a 64-bit extension, however relatively few devices support it.

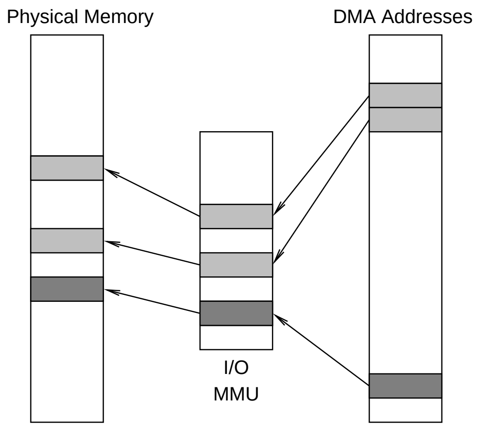

Figure 1: DMA addresses translated through an I/O MMU.

the driver.

3 Mungi user level drivers

We are using the Mungi system [HEV+98] running on both the Alpha DS-20 and HP rx2600 (Itanium 2) as a test-bed for our driver work. Mungi is based on the idea of a single address space [CLFL94], and supports a component model with hardware-enforced encapsulation for providing secure user-level extensions to the kernel [EH01]. Mungi components hide their instance data from external access (other than via declared method interfaces). Otherwise, the system places no restrictions on components and their implementation, in particular, any language can be used to implement a component.

We are implementing device drivers as Mungi components. The driver model [Les02] is asynchronous, with request completion signalled to the client code by the driver invoking the client's completion method. Similarly, the kernel delivers interrupts to the driver by invoking the driver's interrupt method. The driver uses a system call to request from the kernel pinned pages to be used as DMA buffers.

Due to their complexity, device drivers are often reused from existing systems rather than rewritten for a new system. Although this reuse of code is attractive, past experience has shown that this is by no means a straightforward process. We found it was easier to implement a set of drivers from scratch, based on a driver model that was cleanly integrated with the Mungi model.

Encapsulation of a driver inside a protected component is enough to contain many (non-Byzantine) faults. If, for example, the driver de-references an invalid pointer, this will result in a protection error, which the kernel handles by re-initialising the driver. It can also alert the administrators to the misbehaving driver (and the driver can later be substituted with an improved version by simply registering a replacement component).

However, encapsulation is not sufficient for protecting the system from some faults, and in particular malicious behaviour, as the driver can normally use the device's DMA mode to overwrite arbitrary regions of physical memory.

In order to limit the damage that can be caused by DMA, we make use of the IO MMU supported by the zx1 (used in the HP rx2600) and Tsunami (used in the DS20) chipsets, which interface the processor to the PCI bus and system memory in our workstations. We use this I/O MMU to restrict DMA access to specified I/O buffers, in a fashion that is completely transparent to the device driver: When the driver needs to DMA a buffer, it requests the chipset driver to pin the buffer in physical memory. The chipset driver uses a system call to request the kernel to pin the pages, and then establishes PCI mappings for them; when the page is un-pinned, that mapping is removed. A driver for a PCI device can therefore only DMA to one of the pinned pages; any attempt to DMA to some other memory region results in a fault which invokes the kernel. Hence, the rest of the system is completely protected from misbehaving PCI device drivers, and only the chipset driver needs to be completely trusted.

This protection is also extremely useful during the development of new drivers. During the development of a gigabit driver for the Mungi platform, multiple bugs were found where the driver was incorrectly performing DMA. This class of bugs would be more difficult and time consuming to find without the effective use of the IO MMU.

Note that buggy or malicious drivers can still corrupt data in I/O buffers during the transfer between client and device. Techniques as they are proposed for SUNDR [MS01] could be used as a protection here. Furthermore, as there exists only one PCI address space which is shared by all PCI devices, one driver could corrupt or sniff another's buffers. However, this danger is minimised by sparsity: a driver would have to guess the location of other buffers in the (admittedly small) 32-bit PCI address space. Other drivers can help to minimise this risk by unmapping their buffers as soon as the respective I/O operations are completed. Hence, present hardware is not quite sufficient to make device drivers totally untrusted, but what can be done is already a huge improvement over the present situation.

4 Experience

For some initial performance evaluation of our user-level drivers, we used a very conservative setup which results in a high number of protection domain crossings.

The test setup uses three user-level Mungi tasks: a client, an IDE driver, and the chipset driver, each in their own protection domain. For each disk request, the user task invokes the IDE driver, which invokes the PCI driver to request pinning of the buffer (and mapping into the PCI address space). The IDE driver then initiates the I/O and finally returns to the client. If the IDE driver has outstanding requests queued at the time a new user request comes in, it simply queues the new request and returns immediately.

When the device has completed transfer, an interrupt is delivered directly to the IDE driver. For requests exceeding the size of an individual device transfer, the IDE driver invokes the PCI driver to unpin the buffer of the completed request and pin the buffer for the next request, which it then initiates. On an interrupt indicating the transfer of the final block, the IDE driver invokes the PCI driver to unpin the buffer and then invokes the client's completion method.

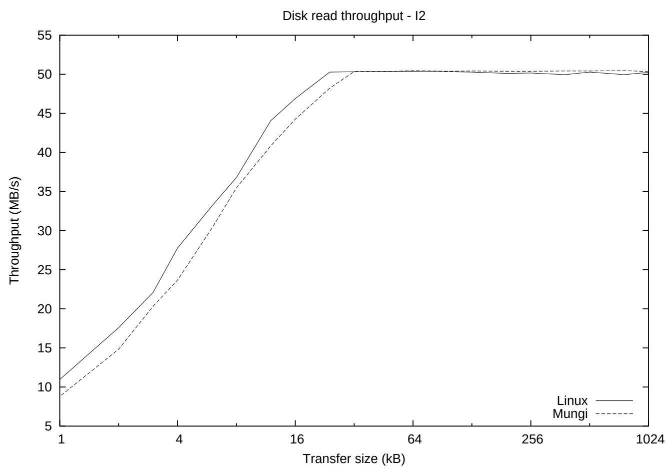

Figure 2: Disk read throughput

Figure 2 shows the throughput achieved for different request sizes. For large transfer size the throughput is quite similar for both systems, however as the transfer size decreases the context-switching overhead of the user-level drivers starts to have an effect, reducing throughput by up to 22 % (at extremely small transfer sizes).

Figure 3 shows CPU utilisation resulting from I/O processing (measured by instrumenting the idle loop). As is to be expected, Linux consumes less CPU for small transfers, a result of the context switching overhead of the user-level drivers. Mungi does however, perform, slightly better for larger transfer sizes which is probably the result of the cleaner drivers which might result in a smaller overall cache footprint. This indicates that a lean design might help to offset some of the inherent context-switching costs.

5 Related work

A number of previous systems have moved the network protocol stacks from kernel to user level while leaving the device driver in the kernel [TNML93,MB93,EM95]. Other approaches provided user code direct access to network interfaces in order

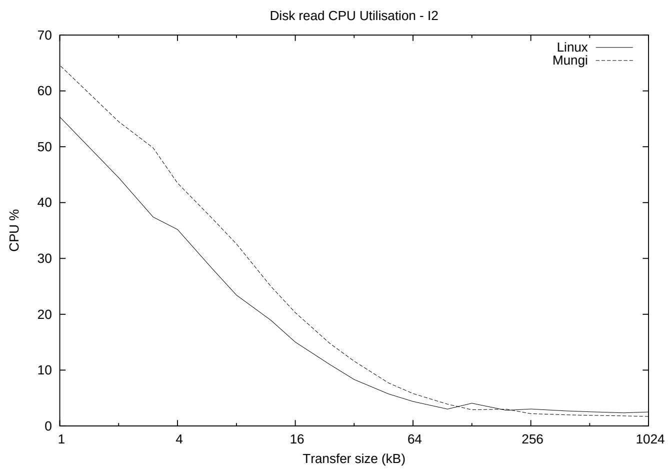

Figure 3: CPU utilisation

to minimise latency for fine-grained communication in high-performance clusters [vEBBV95, Dam98].

Earlier work with real user-level drivers in Mach [GSR93] and Fluke [VM99] experienced significant performance problems, apparently resulting from the IPC costs in those kernels.

The *Palladium* approach of running Linux kernel extensions at an intermediate privilege level [CVP99] 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. Pratt proposed an I/O device architecture that would allow the Nemesis system to run device drivers at user level [Pra97], but in the absence of devices conforming with this architecture, drivers are still in the kernel.

Work at the University of Washington [SMLE02] 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. In spite of nooks being somewhat half-way between normal in-kernel and user-level drivers, the cost of nooks is significant — the paper reports more than doubling the interrupt latency. The inherent overhead of nooks is essentially the same as that of user-level drivers, and it is therefore not clear what their advantage is over what we consider the cleaner approach of running drivers as proper user-level processes.

Significantly these approaches either fail to recognise the problem presented by DMA, or pass it off as an unimportant issue.

6 Conclusions

This report has shown that it is possible to encapsulate device drivers as unprivileged user-level processes. By utilising the IO MMU hardware available in modern platforms, we have shown that this encapsulation can be extended to control even DMA access, making it impossible for device drivers to access data other than DMA buffers registered with the kernel. Initial benchmarks indicate that all this might be possible with tolerable overhead, although a more detailed analysis will be required. We believe that this represents a significant step towards systems that do not need to trust their drivers.

References

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, 1994.

[CVP99] Tzi-cher Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on OS Principles (SOSP)*, pages 140–153, Kiawah Island, SC, USA, December 1999.

[CYC$^+$01] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on OS Principles (SOSP)*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[Dam98] Stefanos N. Damianakis. *Efficient Connection-Oriented Communication on High-Performance Networks*. Phd thesis, Princeton University, 1998.

[EH01] Antony Edwards and Gernot Heiser. Components + Security = OS Extensibility. In *Proceedings of the 6th Australasian Computer Systems Architecture Conference (ACSAC)*, pages 27–34, Gold Coast, Australia, January 2001. IEEE CS Press.

[EM95] Aled Edwards and Steve Muir. Experiences implementing a high performance TCP in user-space. In *Proceedings of the ACM Conference on Communications (SIGCOMM)*, 1995.

[GSR93] David B. Golub, Guy G. Sotomayor, Jr, and Freeman L. Rawson III. An architecture for device drivers executing as user-level tasks. In *Proceedings of the USENIX Mach III Symposium*, pages 153–171, 1993.

[HEV$^+$98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[Les02] Ben Leslie. Mungi device drivers. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2002. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[MB93] Chris Maeda and Brian N. Bershad. Protocol-service decomposition for high-performance networking. In *Proceedings of the 14th ACM Symposium on OS Principles (SOSP)*, pages 244–255, Asheville, NC, USA, December 1993.

[MS01] David Mazières and Dennis Shasha. Don't trust your file server. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 113–118, Elmau, Germany, May 2001.

[Pra97] Ian A. Pratt. *The User-Safe Device I/O Architecture*. PhD thesis, King's College, University of Cambridge, August 1997.

[SMLE02] Michael M. Swift, Steven Marting, Henry M. Levy, and Susan G. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th SIGOPS European Workshop*, pages 101–107, St Emilion, France, September 2002.

[TNML93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1:554–565, 1993.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on OS Principles (SOSP)*, pages 40–53, Copper Mountain, CO, USA, December 1995.

[VM99] Kevin Thomas Van Maren. The Fluke device driver framework. Msc thesis, University of Utah, December 1999.