uvNIC: Rapid Prototyping Network Interface Controller Device Drivers

Matthew P. Grosvenor
University of Cambridge Computer Laboratory
matthew.grosvenor@cl.cam.ac.uk

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management – *Network communication*

General Terms

Design, Experimentation, Verification.

Keywords

Hardware, Device Driver, Emulation, Userspace, Virtualisation

1. INTRODUCTION

Traditional approaches to NIC driver design focus on commodity network hardware, which exhibit slow moving feature sets and long product life cycles. The introduction of FPGA based network adapters such as [1][2] alter the status-quo considerably. Whereas traditional ASIC based NICs may undergo minor driver interface revisions over a timespan of years, FPGA based NIC interfaces can be totally reimplemented in months or even weeks. To the driver developer this presents a considerable challenge: Driver development cannot seriously begin without hardware support, but is now expected to take place simultaneously with hardware development.

To solve this problem, I present the userspace, virtual NIC framework (uvNIC). uvNIC implements a custom virtual NIC as a standard userspace application. To the driver developer, it presents a functional equivalent to a physical device. Only minor modifications are required to switch a uvNIC enabled driver over to operating on real hardware. To the hardware designer, uvNIC presents a rapid prototyping environment for initial specifications and a fully functional model against which HDL code can later be verified.

2. Design and Implementation

Typical NIC device drivers implement two interfaces; a device facing PCI interface and kernel facing network stack interface. Ordinarily, a device driver would send/receive packets by interacting with real hardware over the PCI interface. Instead of (or addition to) regular PCI operations, uvNIC forwards interactions with hardware to the uvNIC virtual NIC application. This application implements a software emulation of the hardware NIC and responds appropriately by sending and receiving packets over a commodity device operated in raw socket mode.

Implementing the uvNIC PCI virtualisation layer is not trivial. OS kernels are designed with strict one way dependencies. That is, userspace applications are dependent on the kernel, the kernel is dependent on the hardware. Importantly, the kernel is not designed for, nor does it easily facilitate dependence on userspace applications. For the uvNIC framework, this is problematic. The virtual NIC should appear to the driver as a hardware device, but to the kernel it appears as a userspace application.

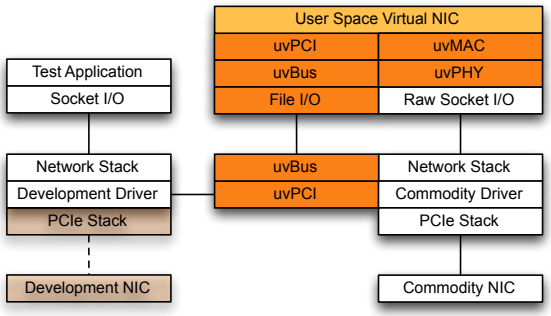

Figure 1: The uvNIC framework design.

Figure 1. illustrates the uvNIC implementation in detail. At the core is a message transport layer (uvBus) that connects the kernel and the virtual device. uvBus uses file I/O operations (`open()`, `ioctl()`, `mmap()`) to establish shared memory regions between the kernel and userspace. Messages are exchanged by enqueuing and dequeueing fixed size packets into lockless circular buffers. Message delivery order is strictly maintained. uvBus also includes an out of band, bi-directional signalling mechanism for alerting message consumers about incoming data. Userspace applications signal the kernel by calling `write()` with a 64 bit signal value, likewise, the kernel signals userspace by providing a 64 bit response to `poll()`/`read()` system calls.

A lightweight PCIe like protocol (uvPCI) is implemented on top of uvBus. uvPCI implements posted (non-blocking) write and non-posted (blocking) read operations in both kernel and userspace. In kernel space, non-posted reads are implemented by spinning and kept safe with timeouts and appropriate calls to `yield()`. An important aspect of uvPCI is that it maintains read and write message ordering in a manner that is consistent with hardware PCIe implementations.

In addition to basic PCI read and write operations, uvPCI implements x86 specific PCIe restrictions such as 64 bit register reads/writes, message signalled interrupt generation and 128B, 32bit aligned DMA operations. DMA operations appear to the driver as they would in reality. That is, data appears in DMA mapped buffers asynchronously without the driver's direct involvement.

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.
ACM 978-1-4503-1419-0/12/08.

In practice, uvPCI implements a functionally equivalent, parallel implementation of the PCI stack. Driver writers perform little more than a search/replace and a recompile to switch over to using the real PCI stack.

3. Related Work

Userspace networking has a long history [3][4] and continues to be employed widely, especially in high performance situations [5][6]. Whilst uvNIC shares the basic concept of implementing a part of the network stack in userspace, it is distinct from previous attempts because it implements the network hardware in userspace software rather than parts of network driver or IP stack as is commonly the case. It is crucial to note that network performance is not the primary goal of uvNIC, rather, the primary goal is rapid prototyping at the software/hardware interface.

The structure and function of uvNIC is highly similar to the virtual NICs found in hypervisors and virtual machines (VMs). Both VMware [6] and Xen [7] expose virtualised hardware devices to their guest OSes and hence the guest OS drivers. It is possible that custom hardware could be designed and written in a VM context instead of uvNIC. However, the approach has the distinct disadvantage that development and integration of a new virtual device into a VM is a complex and time consuming task. This is in direct opposition to the stated goal, which is which is to aid rapid prototyping of the NIC and device driver.

uvNIC is also similar to File System in User Space (FUSE) [9] systems. Like uvNIC, FUSE requires that kernel become dependent on userspace applications. In contrast, however, FUSE systems are simpler than uvNIC because no direct emulation of hardware timing, ordering and consistency parameters are required.

4. Preliminary Results

A uvNIC virtual network device has been implemented on a Linux 2.6 host. The virtual device was tested with simple linux tools such as `ping` and `traceroute` and found to be functional. This test confirmed that the uvNIC framework is suitable for writing simple, but functional network devices and device drivers. Since performance was not a consideration, `iperf` tests were not performed.

Additionally, a uvNIC driver has been written for and tested against a NetFGPA 10G card [2] which was running a simple register and MSI interface firmware module. This test confirmed that the uvNIC framework is capable of writing simple device drivers that are portable to real hardware platforms.

Efforts are currently underway to port the feature rich Intel IXGBE 10G driver to a uvNIC virtual device using an Intel e1000 1G as a physical device. This comprehensive test will demonstrate uvNIC's ability to augment simple physical hardware with additional virtual functionality.

5. Conclusions and Next Steps

uvNIC is a simple but unique approach to a real problem. Work so far has indicated that the approach is valid and functional. Future steps may include migrating the virtualised hardware model to a cycle accurate simulation of hardware RTL code or using the framework as a measurement and debugging tool for real device drivers. uvNIC affords designers a unique opportunity to rapidly explore the NIC software/hardware interface at low cost. It is hoped that this ability will lead to faster and more stable NIC designs.

More information on the uvNIC project can be found at http://www.cl.cam.ac.uk/research/srg/netos/mrc/projects/uvNIC

6. Acknowledgements

This work was jointly supported by the EPSRC INTERNET Project EP/H040536/1 and the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

7. REFERENCES

[1] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. 2007. NetFPGA--An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education* (MSE '07). IEEE Computer Society, Washington, DC, USA, 160-161.

[2] NetFPGA 10G Project, *NetFPGA website*, http://www.netfpga.org

[3] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: a user-level network interface for parallel and distributed computing. *SIGOPS Oper. Syst.* Rev. 29, 5 (December 1995), 40-53.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. 1995. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro15*, 1 (February 1995), 29-36.

[5] Ian Pratt, Keir Fraser. 2001. Arsenic: A user-accessible gigabit ethernet interface. *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies,* INFOCOM01.

[6] David. Riddoch, Steven. Pope. 2008. OpenOnload, A user-level network stack. *Google Tech Talk,* http://www.openonload.org/openonload-google-talk.pdf

[7] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. 2001. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, Yoonho Park (Ed.). USENIX Association, Berkeley, CA, USA, 1-14.

[8] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. 2006. Optimizing network virtualization in Xen. In Proceedings of the annual conference on USENIX '06 Annual Technical Conference (ATEC '06). USENIX Association, Berkeley, CA, USA, 2-2.

[9] Miklos Szeredi. 2012, *File System in User Space*. http://fuse.sourceforge.net