UNIVERSITY OF CAMBRIDGE
800 YEARS
1209 ~ 2009

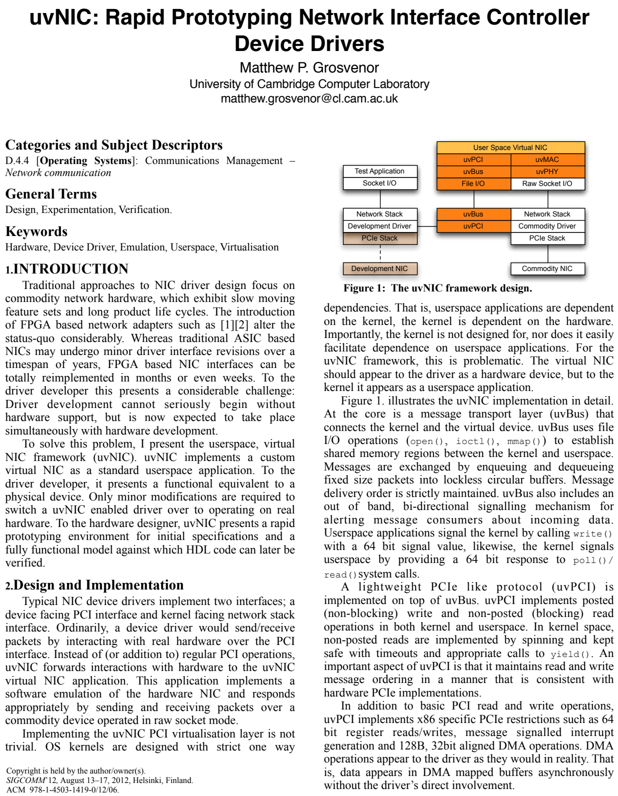
uvNIC

Rapid Prototyping Network Devices

Matthew. P. Grosvenor
Andrew Moore
Robert Watson

Computer Laboratory

But wait, there’s more...



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

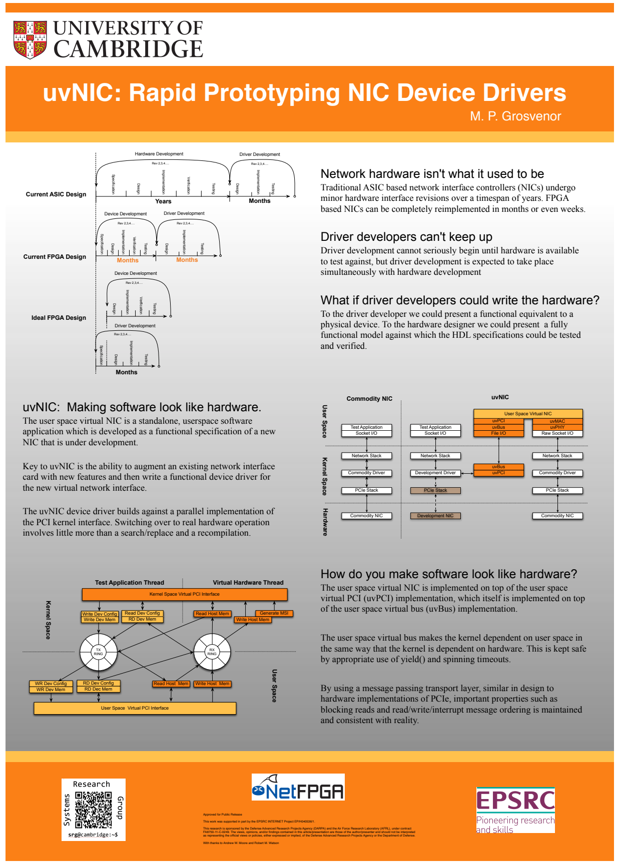
Figure 1: The uvNIC framework design.

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. 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.



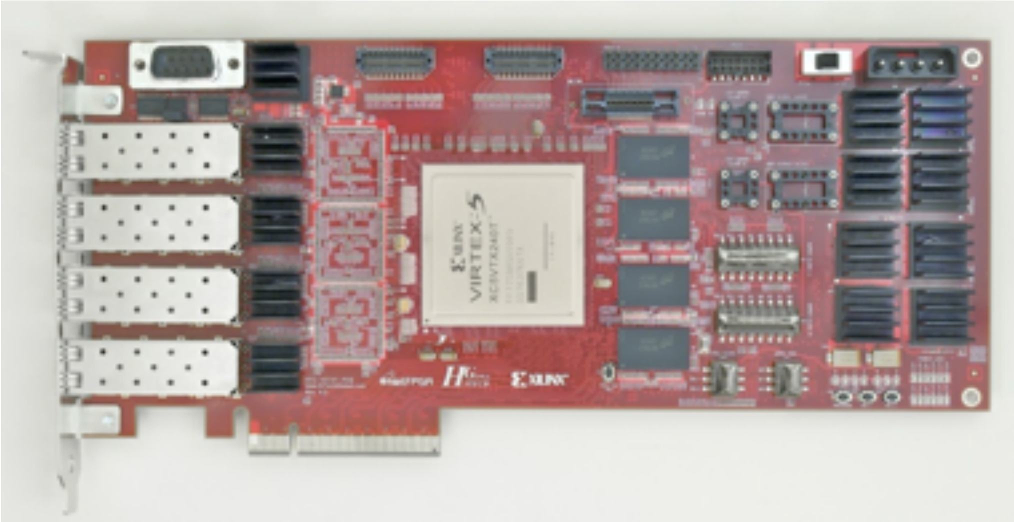
Thursday, 12 July 12

Motivation

- Network latency
- How long does a packet take to traverse network components like switches, routers, firewalls, NICs, OSes etc

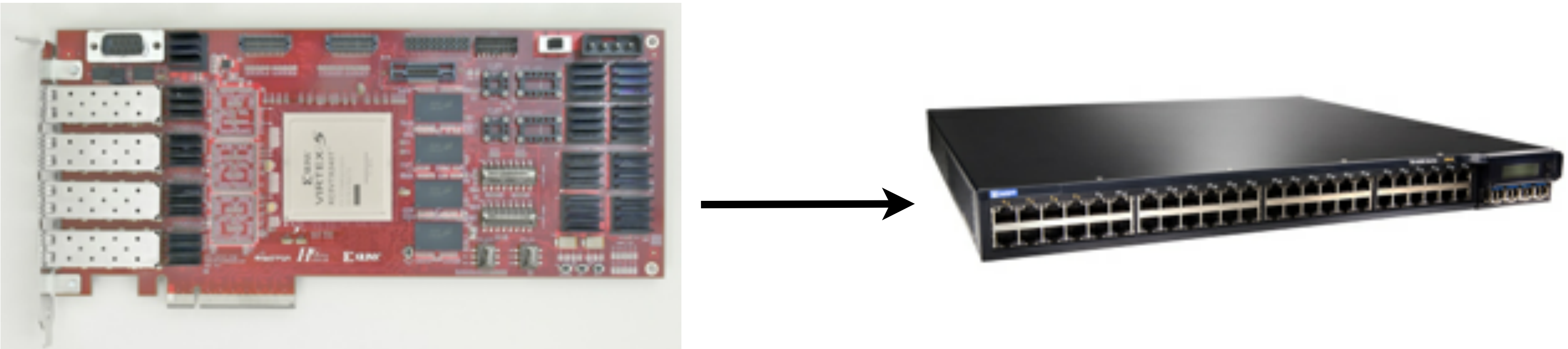
NetFPGA

- 4x 10G Network ports (SFP+)
- Programable FPGA fabric
- PCIe 8x Connector
- RAM & other things.



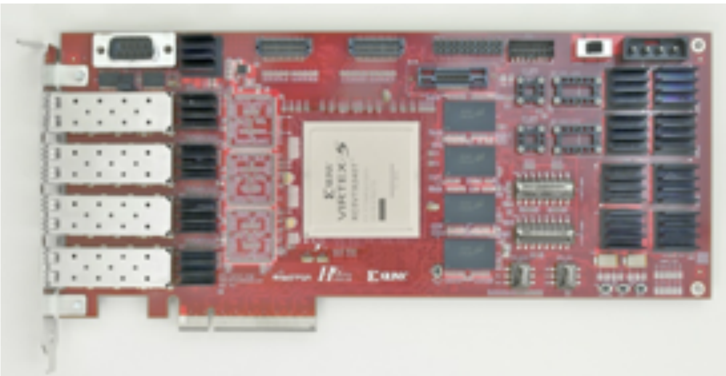
What’s the problem?

- Can implement a switch



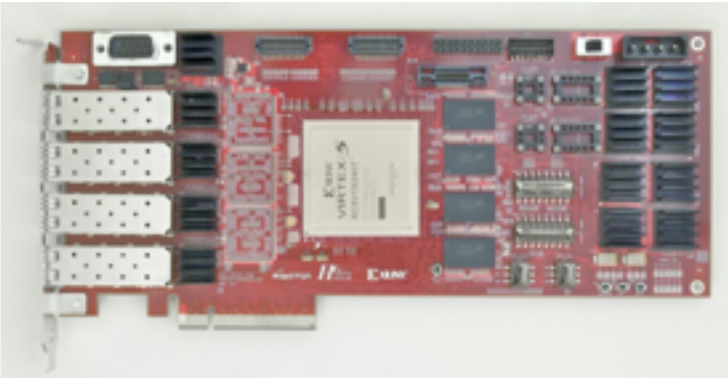
What's the problem?

- Can implement a high performance network card



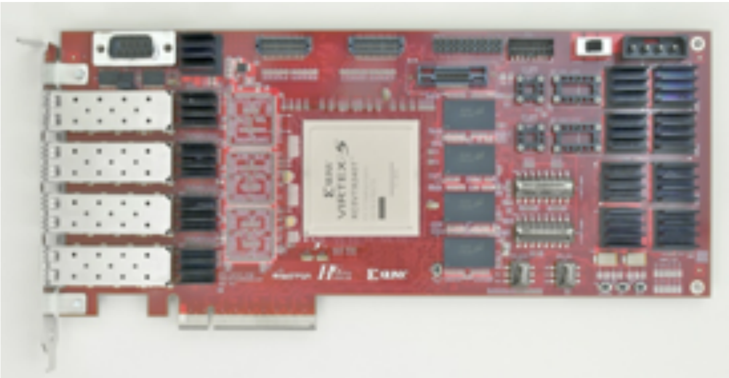
What's the problem?

- Can implement a network monitoring device



What’s the problem?

- Can implement a router



What's the problem?

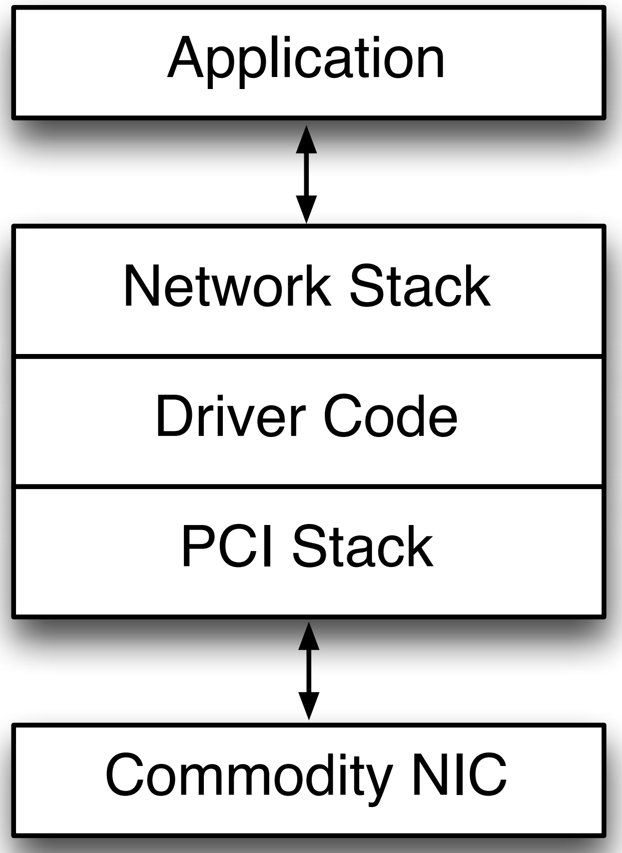
- Can implement just about any network device you can think of ....

→ ?

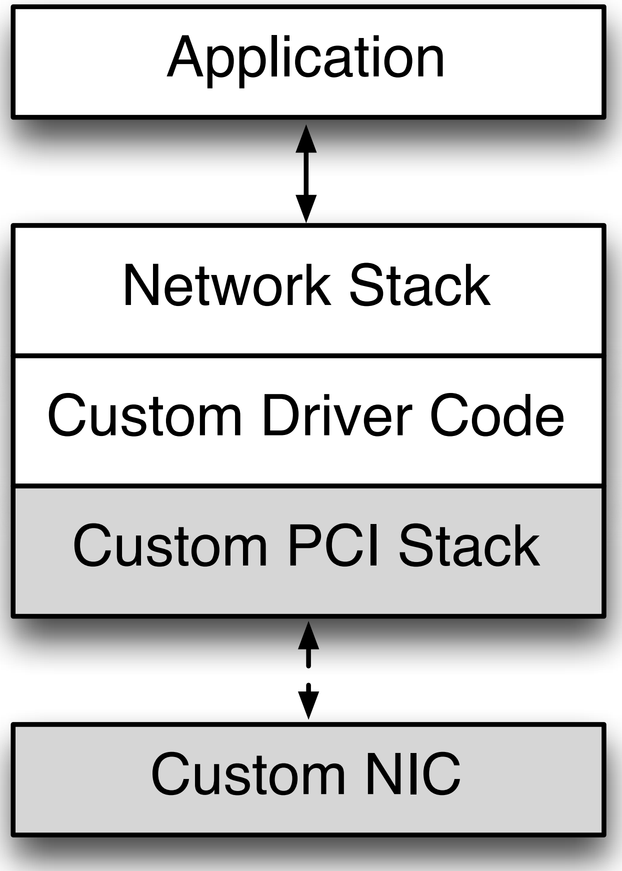
What's the problem?

- How do we write a device driver for a device that can be any device?
- How do we write that driver quickly?
- How do we prototype the device functionality? Fast?
- How can we explore different arrangements of hardware software interface without having to build it all?

A very brief introduction to NIC drivers (Linux/Unix)



Hijacking `struct pci_dev`

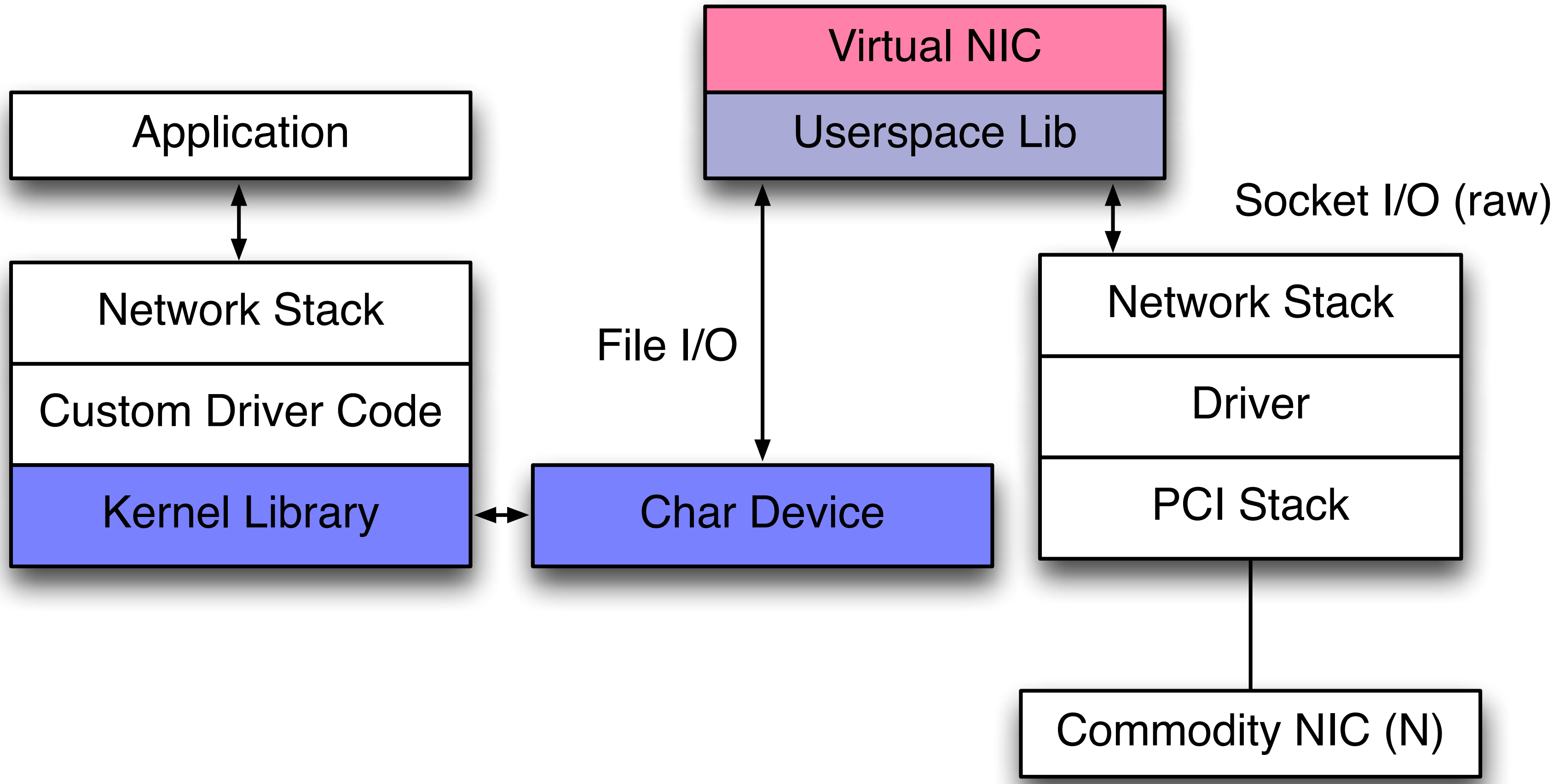


Introducing uvNIC

- uvNIC is a software implementation PCI express, implemented in a way that a device driver is (almost) unaware of its existence.

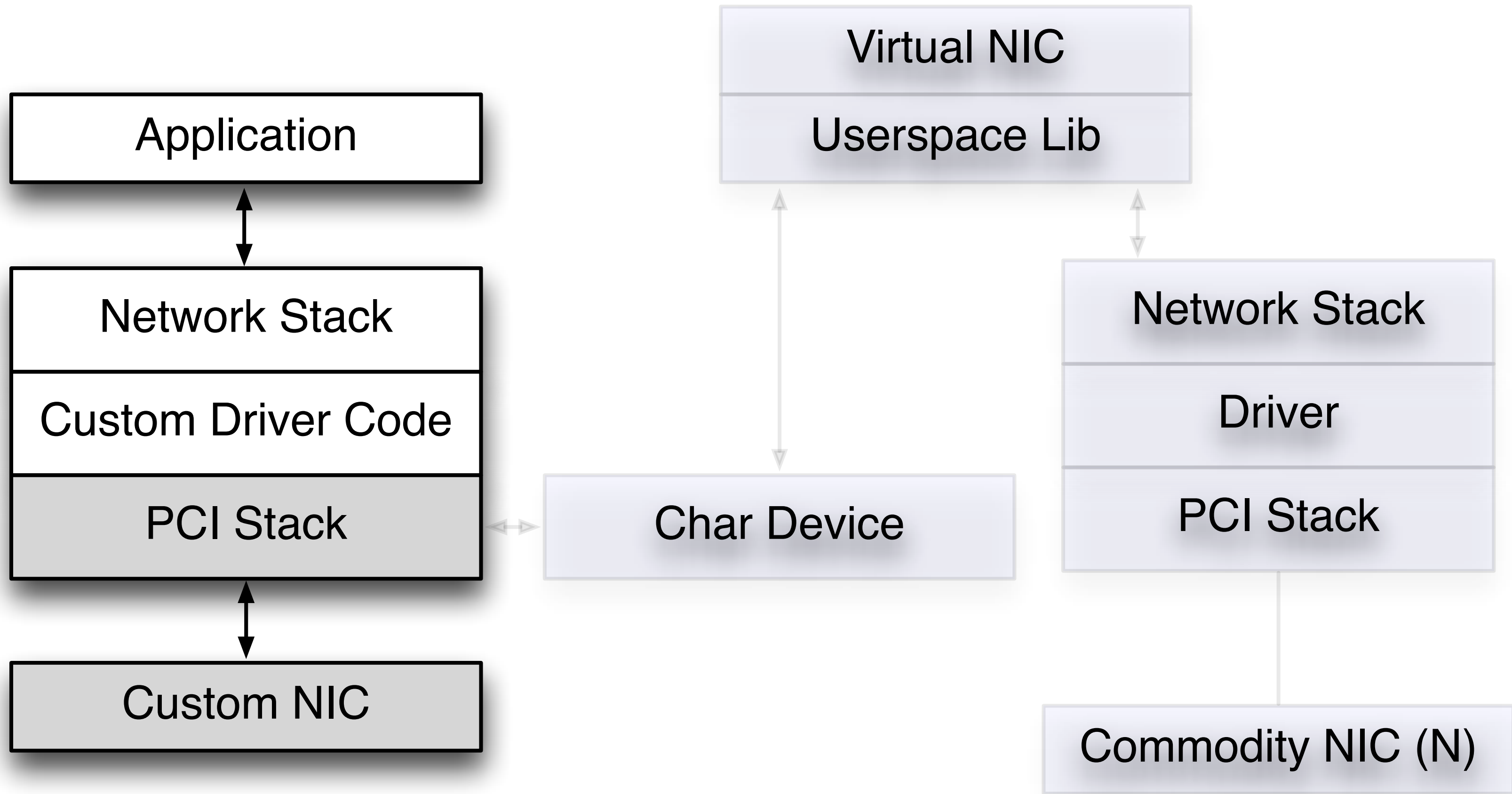
Introducing uvNIC

The userspace virtual NIC



Introducing uvNIC

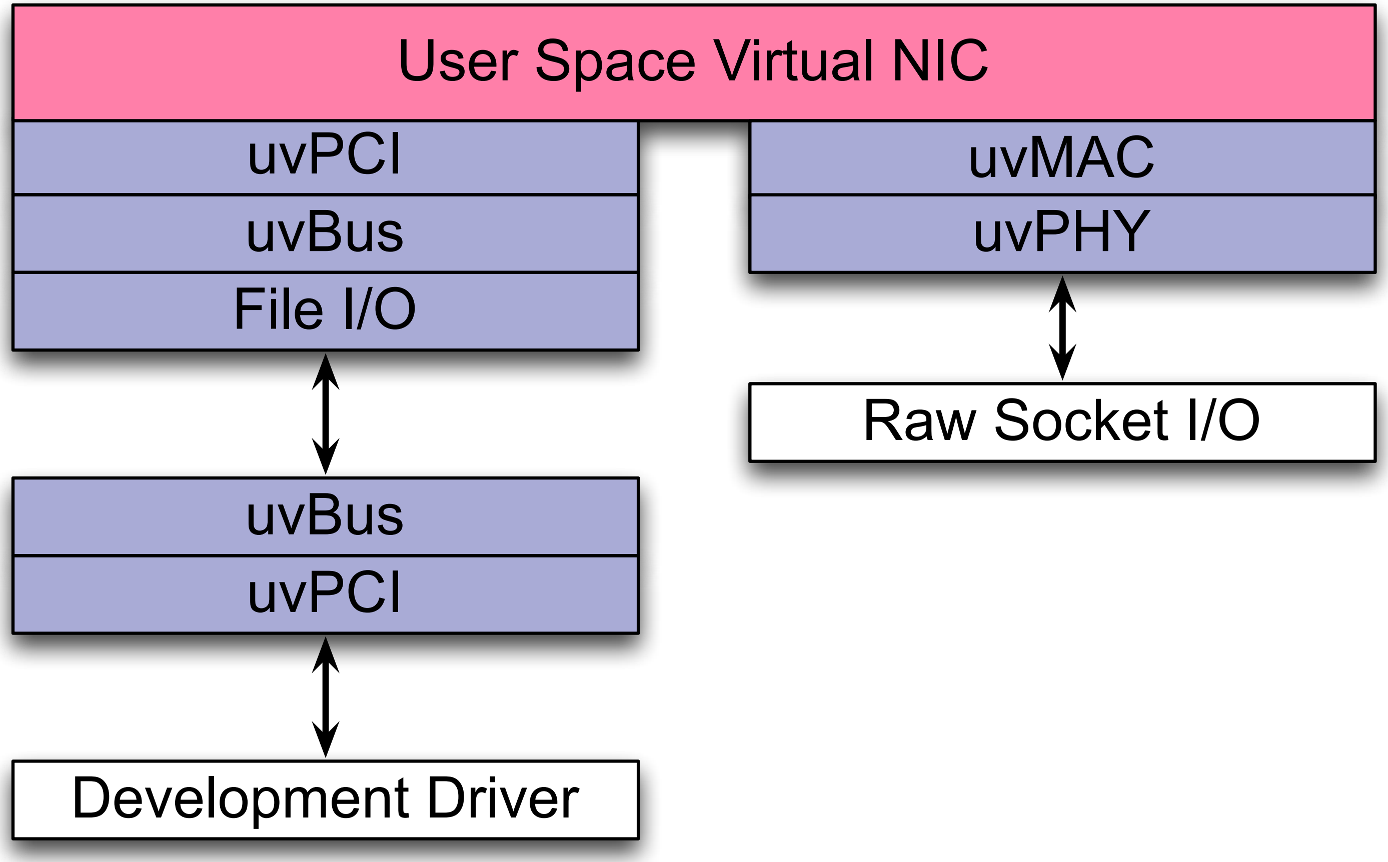
The userspace virtual NIC



Faking PCI(e): What needs to be done?

- Require 5 functions to fake PCI(e) hardware
  - Write Register
  - Read Register
  - Read DMA
  - Write DMA
  - Interrupt request

Faking PCI(e): How does it work?



Faking PCI(e): uvBus

- Simple message bus.
- Connects kernel and userspace.
- Implemented as shared memory ring buffer.
- Transmits 128B messages, much like PCIe.
- Guarantees order and delivery.
- Rejects new messages if the ring buffer is full.

Faking PCI(e): uvPCI

- Implemented over uvBus
- Implements blocking read and non blocking write (like PCIe)
- Uses timeouts and yield to keep the kernel from blocking forever
- Bi-directional
- Implements configuration space reads and writes

Faking PCI(e) the nasty details: uvPCI x86 specifics

- Host to device reads/writes limited to 64bits
- Device to host reads/writes limited to 128B
- Interrupts implemented as write messages to a special address
- Message signalled interrupts (MSI) only

Preliminary Results

- Built a (very) simple virtual network card and driver for it
  - 1 packet queue with 1 slot for TX
  - 1 packet queue with 1 slot for RX
  - Relatively painless process
- Built a test driver and switched over for a simple hardware design
  - Register reads/writes
  - IRQs
- In progress...
  - Backporting an Intel IXGBE 10G NIC to run on uvNIC virtual hardware

Results Summary

- Rapid prototyping network devices
- Quickly exploring the software/hardware interface
- Painless transition to real hardware

uvNIC: General Points

- What uvNIC **IS**:
  - A fast way to build device drivers that actually work
  - A fast way to prototype network devices
  - A fast way to prototype arrangements of hardware/software interface:
    - Register layout/policy
    - DMA policy
    - IRQ policy,
    - Transaction formats
    - Queues, Descriptors, Rings, Offload etc etc etc

uvNIC: General Points

- What uvNIC is **NOT**:
  - Safe: Kernel data structures exposed to userspace arbitrarily
  - Safe: Kernel has a contract with userspace.
  - Complete: Only a minimal implementation of PCIe functions supporting the functionality required to make NICs work.
  - High performance: **This is not and was never the goal**
  - A replacement for Xen like devices: This is all about rapid prototyping

uvNIC: With thanks to our sponsors

Matthew. P. Grosvenor

Supervisors:
Andrew Moore
Robert Watson

NetFPGA

EPSRC
Pioneering research and skills

Systems Research Group
srg@cambridge:~$

This work was supported in part by the EPSRC INTERNET Project EP/H040536/1.

This research is sponsored by 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.

uvNIC: Demo (sort of)

- Full duplex TX/RX over uvNIC

```
$ ping -I uvNICnet0 172.16.84.2
PING 172.16.84.2 (172.16.84.2) from 172.16.84.161 uvNICnet0: 56(84) bytes of data.
64 bytes from 172.16.84.2: icmp_req=1 ttl=128 time=0.217 ms
64 bytes from 172.16.84.2: icmp_req=2 ttl=128 time=0.241 ms
64 bytes from 172.16.84.2: icmp_req=3 ttl=128 time=0.247 ms
64 bytes from 172.16.84.2: icmp_req=4 ttl=128 time=0.274 ms
--- 172.16.84.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.217/0.244/0.274/0.027 ms
```

Faking PCI(e): The software perspective

```
#include <linux/pci.h>

struct pci_driver
struct pci_dev

pci_register_driver()
pci_unregister_driver()
pci_enable_device()
pci_disable_device()
pci_set_drvdata()
pci_get_drvdata()
pci_enable_msi()
pci_disable_msi()
pci_set_master()
pci_clear_master()

request_irq()
disable_irq()

...
```

Faking PCI(e): The software perspective

```
#include <linux/pci.h>

struct pci_driver
struct pci_dev

pci_register_driver()
pci_unregister_driver()
pci_enable_device()
pci_disable_device()
pci_set_drvdata()
pci_get_drvdata()
pci_enable_msi()
pci_disable_msi()
pci_set_master()
pci_clear_master()

request_irq()
disable_irq()

...
```

Faking PCI(e): The software perspective

```
#include <linux/pci.h>

struct pci_driver
struct pci_dev

pci_register_driver()
pci_unregister_driver()
pci_enable_device()
pci_disable_device()
pci_set_drvdata()
pci_get_drvdata()
pci_enable_msi()
pci_disable_msi()
pci_set_master()
pci_clear_master()

request_irq()
disable_irq()

...
```

→

```
#include <linux/uvn.h>

struct uvn_driver
struct uvn_dev

uvn_register_driver()
uvn_unregister_driver()
uvn_enable_device()
uvn_disable_device()
uvn_set_drvdata()
uvn_get_drvdata()
uvn_enable_msi()
uvn_disable_msi()
uvn_set_master()
uvn_clear_master()

uvn_request_irq()
uvn_disable_irq()

...
```

uvNIC: How does it really work?