uvNIC: Rapid Prototyping NIC Device Drivers
Matthew P. Grosvenor
University of Cambridge Computer Laboratory
mwp@cardiff.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, Userpace, 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 [8] alter the status-quo considerably. Whereas traditional ASIC based NICs may undergo minor driver interface revisions over a timeframe of years, FPGA based NIC interfaces can be totally re-implemented 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 to 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 manipulate packets by initiating with real hardware over the PCI interface. Instead of (or in addition to) regular PCI operations, uvNIC presents a virtual PCI interface. This virtual interface is implemented as a standard userspace application. 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 layer is dependent on the hardware. To solve this problem, I present 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

3. uvNIC: Making software look like hardware.

The uvNIC device driver builds against a parallel implementation of the new virtual network interface.

network stack

Figure 1: The uvNIC framework design

The uvNIC device driver builds against a parallel implementation of the new virtual network interface.

network stack

How do you make software look like hardware?

The user space virtual NIC is implemented on top of the general purpose uvBus implementation, which could be implemented on top of the user space virtual bus (uvBus) implementation.

The user space virtual bus enables the kernel dependent on user space in the same way. This is a software implementation of the interface enabling to be implemented on end-to-end.

by using a message passing transport layer, similar to design to hardware implementations of PCIe, important protocol such as device identification could be implemented using software and hardware using asynchronous, fault entirely.

Development NIC

Implementation

Driver development cannot keep up

Driver development cannot keep up with the hardware, but the uvNIC framework allows for the driver developer to test against, but driver development is expected to take place simultaneously with hardware development.

What if driver developers could write the hardware?

To the driver developer we could present a functional equivalent to a physical device. To the hardware designer we could present a fully functional model against which the HDL specifications could be tested and verified.
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)

Application

Network Stack

Driver Code

PCI Stack

Commodity NIC
Hijacking `struct pci_dev`

Diagram:

- Application
- Network Stack
- Custom Driver Code
- Custom PCI Stack
- Custom NIC
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

- Application
- Network Stack
- Custom Driver Code
- Kernel Library

- Virtual NIC
- Userspace Lib
- File I/O

- Socket I/O (raw)

- Network Stack
- Driver
- PCI Stack
- Commodity NIC (N)
The userspace virtual NIC

- Application
- Network Stack
- Custom Driver Code
- PCI Stack
- Custom NIC

Virtual NIC
- Userspace Lib

Virtual NIC
- Network Stack
- Driver
- PCI Stack
- Char Device

Commodity NIC (N)
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?

User Space Virtual NIC

- uvPCI
- uvBus
- File I/O

uvBus

uvPCI

Development Driver

uvMAC

uvPHY

Raw Socket I/O
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

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.
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
#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/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_enable_msi()

uvn_disable_msi()

uvn_set_master()

uvn_clear_master()

uvn_request_irq()

uvn_disable_irq()

...

Thursday, 12 July 12
uvNIC: How does it really work?

Test Application Thread

Kernel Space Virtual PCI Interface

Write Dev Config
Write Dev Mem

Read Dev Config
RD Dev Mem

Read Host Mem

Generate MSI

Virtual Hardware Thread

Write Host Mem

Kernel Space

User Space

WR Dev Config
WR Dev Mem

RD Dev Config
RD Dec Mem

Read Host Mem

Write Host Mem

User Space Virtual PCI Interface