Between a rock and a hardware place: Ultraviolet and the case for Userspace Virtual hardware

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

1. INTRODUCTION

At the boundary of all operating systems are device drivers, software modules, usually run in fully privileged mode, that connect specific hardware devices with the more general kernel API. Device drivers are highly complex pieces of software. They have to contend with asynchronous event notifications from both hardware and software (often simultaneously), subtle kernel threading and locking models, complex hardware offload state management and notoriously poorly documented interfaces to both the kernel and the hardware itself. Additionally, device drivers are often written to support a multitude of subtle hardware variations. The popular Intel 1G (e1000) and 10G (ixgbe) Linux device drivers both support more than 25 different PCI endpoint identifiers[1] each.

Faced with this range and complexity of devices, driver developers rarely test their device drivers against all possible devices and all device variants thereof. A recent study found that over two dozen Linux and BSD driver patches were marked as "compiled tested only"[1]. Yet, despite their complexity and minimal testing, device drivers are by far the largest contributor to kernel code line counts. Approximately 60%[2] of the Linux 3.9 kernel source code is device driver code alone. This figure is likely to be higher in the Windows kernel with its anecdotally larger device support than Linux. Worryingly, device diver code has been found to contain 3-7x more errors per line of code than the rest of than other kernel code [2]. It is not surprising then that these subtle, complex, fragile, error prone and highly privileged modules are a major source of failure. Over 75% of all system crashes reported in Windows XP were due to device driver failures [3]. An analysis of defects in a range of Linux device drivers revealed that device protocol violations was leading the cause of failure, accounting for nearly 40% of all reported bugs [4]. Device protocol violations occur when the driver behaves in a way that violates the required hardware-software interface usually resulting in an indeterminate state in hardware and/or a causing catastrophic crash in the kernel.

In this already challenging and failure prone environment, a new class of devices is adding further complexity to the hardware-software interface. Traditional approaches to hardware design center on commodity, application specific integrated circuits (ASICs). Due to their design expense, ASICs exhibit slow moving feature sets and long product life cycles often taking years to develop and release. Whilst writing drivers for ASIC based devices is already problematic, the hardware-software interfaces are relatively static, changing infrequently and incrementally over time. The relatively recent introduction of field programable gate array (FPGA) based devices such as [5][6] alter the status-quo considerably. Whereas traditional ASIC based designs undergo minor revisions over a timespan of years, FPGA based interfaces can be totally reimplemented in months or even weeks. Furthermore, FPGA based device's reconfigurable flexibility makes then especially attractive for in-field reconfiguration. FPGAs are rapidly reaching a price point suitable for wide scale deployment. Ongoing research [7] suggests that it is now, or soon will be, feasible to equip an entire datacenter (i.e. many 1000s of machines) with FPGA enabled devices.

The fragile, error prone and complex device drivers described above are now expected to have their interfaces updated, modified and rewritten regularly to meet the deadlines and demands of this new, fast moving, hardware category. To the driver developer this presents both a challenge and an opportunity. Whilst it is now more challenging than ever to write and maintain device drivers, for the first time, device driver writers are being given an active voice in the construction of the hardware-software interface while it is being constructed. FPGA based devices can be reprogramed to whatever hardware-software interface the device designer and the driver writer agree upon. The obvious challenge here is, how to go about designing these interfaces in such a way that we minimize errors, account for rapidly increasing device variability, provide a suitable testing framework and, offer a practical solution to both software facing driver writers and hardware facing hardware designers.

To address these challenges, this paper presents Ultraviolet (UV), a library and framework for rapid development and testing of new hardware devices as well as debugging and long term regression testing of older devices.

2. ULTRAVIOLET KEY CONCEPTS

Device drivers typically implement two interfaces; a device facing bus interface and kernel facing data interface. Figure 1a illustrates this arrangement for a typical Network Interface Controller (NIC) device and driver.

[1] Results from analysis of Linux 3.9-rc1 source tree as at 15 April 2013. Tests performed by counting the instances of PCI device endpoints present in the PCI_DEVICE_TABLE in e1000_main.c and ixgbe_main.c.

[2] Results from analysis of Linux 3.9-rc1 source tree as at 15 April 2013. Tests performed by running the *cloc* utility (http://cloc.sourceforge.net) on full the full source tree and the *drivers/* subdirectory. Lines of code (LOC) count excludes comments and blank lines but includes all 22 languages represented in the tree. The 3.9 Kernel release (candidate) has 11.3 million LOC and 6.3 million LOC in the *drivers/* subdirectory.

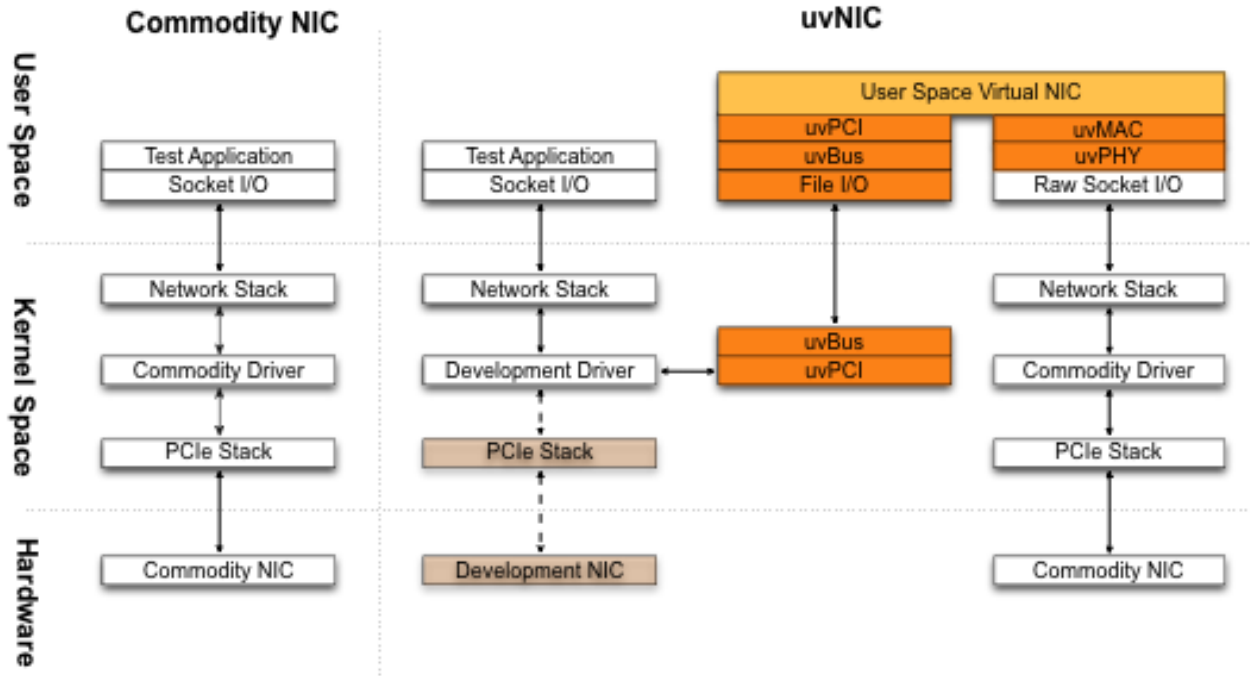

Figure 1a - Arrangement of a typical network interface controller (NIC) hardware and driver; Figure 1b - Arrangement of a the uvNIC driver, virtual PCIe bus, virtual hardware and underlying hardware.

An application uses the kernel API (e.g. sockets or file I/O) to send/receive data to the kernel. The kernel uses an internal API (e.g. netif/skbufs) to communicate with an appropriate device driver. The device driver then sends/receives data by interacting with the hardware over a bus interface (e.g. PCIe). The driver configures and queries registers by issuing read/write requests to the bus. The device responds to these requests by performing direct memory access (DMA) operations and causing interrupts to occur. Finally, the device driver marshals this data back into a form acceptable to the kernel (e.g. an skbuf) and allows the kernel to communicate data back to the application.

The key concept behind Ultraviolet, is intercept the software hardware interface, by hijacking the communication bus (e.g. PCIe) at a source level and routing that communication out of the kernel into a userspace virtual (UV) hardware application, leaving the device driver unmodified. A virtual hardware application implements the functionality of a hardware device without the requirement for the physical device to be present. The device driver is therefore unaware that it is running against a virtual hardware device and so behaves as it would were there a real hardware device attached. This means that the driver can be tested against hardware devices that are not available, are not attached or do not yet exist. It also provides a convenient mechanism to sketch out and test the error prone hardware-software interface. That is, the layout of registers, the protocol for communication and DMA and the operation and synchronisation of state machines in both the device and the driver. Because UV intercepts bus communication it is perfectly located to detect invalid address access violations from both the device and the driver, issuing warnings rather than crashing the machine as is likely with real hardware.

3. PROOF OF CONCEP: THE USERSPACE VIRTUAL NIC (UVNIC)

As a proof of the Ulatraviolet concept, I have implemented and tested a complete UV version of the PCI express bus communications sublayer in the linux kernel. Over the top of this, I have written and ported a collection of real and imaginary userspace virtual network interface controllers (uvNICs). There is no specific conceptual reason for choosing PCIe and NICs as a starting point other than familiarity with both the hardware and software in the NetFPGA 10G project. Experiences with Ultraviolet so far suggest that it is highly amenable to implementing USB, IEEE1394 and other communication protocols as well.

A high level view of the uvNIC proof of concept is illustrated in figure 1b. Unlike figure 1a, instead of (or addition to) regular PCI operations, uvPCI forwards interactions with PCI hardware over a uvBus device to a userspace 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. This gives the driver the impression that a real hardware device is running and connected to a real network even though no such device exists.

Implementing the a virtual hardware device and PCI virtualisation layer is not trivial. Operating system kernels are designed with strict one way dependencies in mind. 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 (i.e. blocking) on userspace applications. For the ultraviolet framework, this is problematic. The virtual device should appear to the driver as a hardware device, but to the kernel it appears as a userspace application. Solving this problem ultimately turned out to be easier than expected, so long as care consideration of threading behaviour was taken into account (described later).

Figure 2. illustrates the uvNIC proof of concept implementation in more detail. At the core is the uvBus message transport layer. UvBus connects the kernel and

the virtual device by using standard file I/O operations (`open()`, `ioctl()`, `mmap()`). It establishes a pair of shared memory regions between the kernel and user-space. Messages are exchanged by enqueuing and de-queueing fixed size packets into the lockless circular buffers in the shared memory regions. 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 user-space by providing a 64 bit response to `poll()/read()` system calls.

A lightweight PCIe like protocol (uvPCI) is implemented on top of uvBus. Much like real PCI, uvPCI implements non-blocking write (posted) and blocking read (non-posted) operations in both kernel and userspace. The most important of these operations is blocking reads in kernel space. When using hardware implementations of PCI, if the CPU issues a read instruction to PCI, it halts progress until the response is received. Emulating this behaviour is critical to the functioning of uvPCI but is not trivial. It requires that the kernel block waiting for a response from userspace, inverting the usual order of dependence in the kernel. This is implemented in uvPCI by spinning on uvBus shared memory device and kept safe by using timeouts (in case the device software has crashed) and appropriate calls to `yield()`. It is vitally important that the driver thread in the kernel call yield, as it gives the virtual hardware thread an opportunity to run. The system will deadlock rapidly if this is not the case. An important aspect of uvPCI is that it maintains read and write message ordering in a manner that is consistent with hardware PCIe implementations. Although a software implementation of a hardware device will never be as fast a the real thing, the strict ordering of messages ensure that the device driver is exercised though nearly all possible states. In some senses, the slowness of the hardware to react will exercise the driver's correctness more so than fully functioning hardware. One major shortcoming of Ultraviolet is that it is possible that timing issues only see at full speed will not be found in the virtual environment. This is discussed further in Section 6 - Future Work.

In addition to basic PCI read and write operations, uvPCI implements a suite of x86 specific PCIe restrictions and functions. A complete PCI configuration space is implemented including 32bit and 64 bit memory and I/O base address registers along with commands to read and write the configuration space. This is important as it is the mechanism by which drivers determine and map the register space of the device. Register reads/writes are also implemented and limited to 32bits / 64bits according to the configuration space above. Interrupts are message signalled and those messages share the same infrastructure as regular messages. This ensures that ordering in the system is strictly maintained. Finally, PCI enforces limits on the maximum DMA fetch size and alignment. Many PCIe root hubs implement 128B, 32bit aligned DMA operations as does uvPCIx86. Since DMA operations are initiated by the virtual hardware thread, they enter the kernel as a different thread to the device driver itself. This means that DMA transactions appear to the driver as they would in reality. That is, data appears and is removed from DMA mapped buffers asynchronously without the driver's direct involvement.

The final and most important layer of the uvPCI environment is the device driver interface. It is critical to the goals of Ultraviolet that the device driver remain unmodified or nearly so. With substantial effort it is possible that Ultraviolet could be built deep into the kernel, allowing some sort of kernel option to switch between virtual and physical device operation. For implementation expedience this path was not taken. Instead, uvPCI requires that driver writers replace `<linux/pci.h>` with `<uvPCI/pci.h>` and that all calls to PCI functions are prepended with the letters "uv". For example, `pci_enable_msix()` is instead written as `uvpci_enable_msix()`. A simple search and replace for all instaces of "uv" with the empty string is all that is required to "port" a uvPCI device over to real PCI. UvPCI implements a functionally equivalent, parallel implementation of the PCI stack. Porting both the production NetFPGA 10G driver and the more complex Intel 10G IXGBE driver to use uvNIC have both proved to be light work.

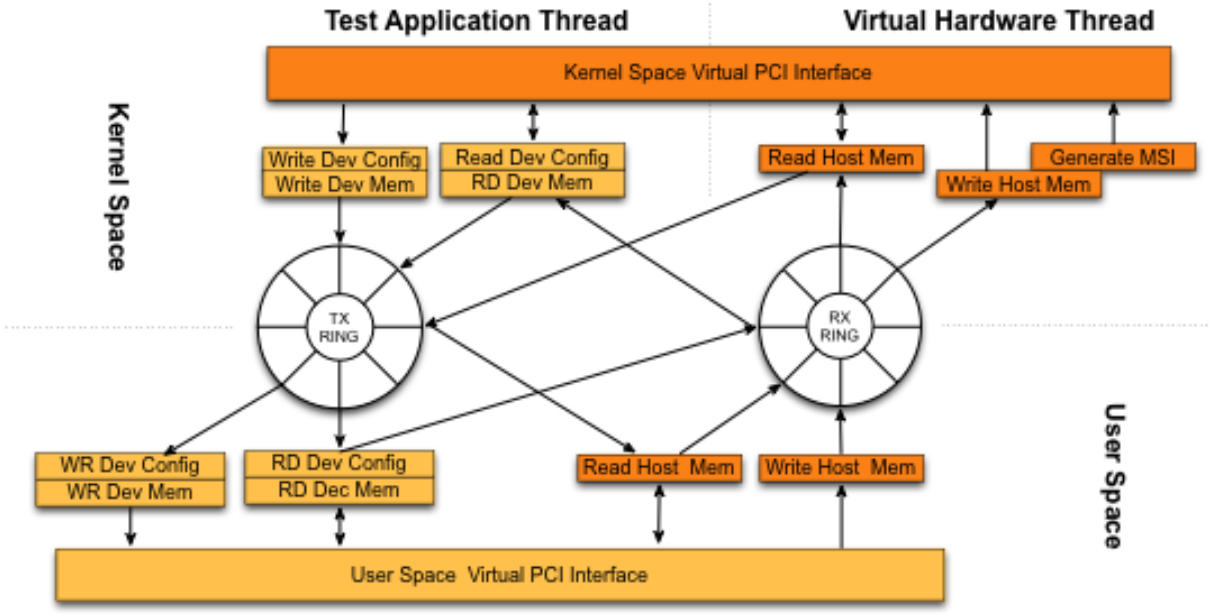

Figure 2 - Detailed view of the uvPCI internal design.

4. EVALUATION

Ultraviolet is a difficult tool too numerically evaluate. As a software emulation of a hardware device, performance numbers have very little meaning. The aim and intent of Ultraviolet was never to high be performance, but instead to provide a framework in which the highly error prone software-hardware interface could be rapidly prototyped, tested and debugged. The key criteria for evaluation are then:

1. *Does/can it work? Can a functional Ultraviolet implementation be put together, especially taking into account the difficulties of inverting the kernel to depend on userspace?*
2. *Is it complete (enough)? Can an ultraviolet implementation be used against a serious production driver? Are there aspects of the framework that are lacking or missing.*
3. *Do drivers port easily? Given virtual hardware driver, how complex is it to move over to using real hardware.*
4. *How complex is the virtual hardware device? Is it a suitable first point for designing and testing useful interface*s?

In order to test these ideas, several virtual devices were written and evaluated. These are described below:

4.1 The 1-in 1-out NIC

High performance network interface adapters operate by directly copying inbound/outbound packets to/from host memory over DMA. Onboard they have a descriptor table that describes the size and location of memory buffers allocated by the host driver to which/from DMA operations must occur. It is the job of the driver to refresh the descriptor table with new memory regions when packets become available or are completed processing. The first device written for Ultraviolet was a network interface adapter with exactly 1 table entry for inbound packets and 1 table entry for outbound packets. This approach was taken to reduce complexity of both the driver and the device. The test was expected to answer the evaluation questions 1 and 4.

The result was a resounding success. The first user space NIC hardware device was written in only 250 lines of C code. It formed a complete and functional user space NIC hardware device. The interface comprised just 4 registers, 1 register for both size and address in both transmit and receive directions forming an effective DMA descriptor table. The device driver was equally simple comprising about 500 lines of C code. An outgoing packet would be directly allocated to the transmit registers. The driver would then wait for a special interrupt number from the card to confirm that the message had been sent. Likewise a single receive buffer was allocated to the receive registers. A special number interrupt number would signal that a packet had arrived. The driver would clear this when ready by resetting size register.

Although simple, this test neatly and fully exercised the design and intent of Ultraviolet. Both a device and a device driver were prototyped, with a trivial but useful device interface protocol. The NIC performed *ping-flood* and *traceroute* operations using standard Unix tools but could not handle a full secure shell (*ssh*) connection. Initially it was thought that this was due to the extremely limited "device resources" available, but later it was found that an ARP misconfiguration in Linux was to blame. Nevertheless, this test did show that a working Ultraviolet hardware device was possible and that the complexity was low enough to make testing and designing useful, partially answering evaluation questions 1 and 4.

4.2 A Simple NetFPGA Hardware Device

A second uvNIC driver was written, this time for a NetFGPA 10G FPGA card, programmed with custom firmware to run a simple register and interrupt generator module. Once prototyped in software, the driver was ported to real hardware and the Verilog hardware description was written and tested. This test was necessarily simple because real hardware development is a long and complex process and was not the aim of the test. This test showed that a minimal but functional driver could be written against a hypothetical specification and run against a real hardware device at a later stage in partial answer to evaluation questions 1 - 4.

4.3 Fully Featured NetFPGA 10G 4 Port NIC

The previous two tests indicated that toy examples of devices were possible to write and run over uvNIC, but did not exercise the framework in any serious way. To more fully answer evaluation question 2, the full production version of NetFPGA 10G driver was ported to uvNIC. This port was interesting for several reasons; Firstly, two minor bugs in the source code were discovered as a result of the port, and, secondly, it showed that very little effort was required to complete the port. It did show, however, that more effort was required to "forward" port a device to uvNIC than to "backwards" port a device from it. This is because a simple search and replace of *"pci_"* with "*uvpci_"* often turned up spurious results where the driver author had named a function or name themselves with the characters "*pci_"*. This is not true of the reverse process, were all instance of "*uvpci_"* could be removed. Despite the extra effort involved, the port of the NetFPGA 10G was again a success. When complete, an 11" Mac BookAir ran an emulated 4 port 10G network card over a basic Intel e1000 NIC using an essentially unmodified driver. By coincidence, the process also formed the first attempt at documenting the internal NetFPGA 10G DMA engine functionality (by reverse engineering) and is unique in the sense that the C code now represents an executable specification of that interface. The NetFPGA port was fully capable of sustaining multiple interactive TCP ssh sessions and supported interactive web browsing to a limited degree[3]. This was a strong answer to evaluation questions 1-4.

4.4 Fully Featured Intel 10G NIC

Work is currently underway to port the Intel 10G IXGBE driver to using uvNIC. The driver compiled against the uvNIC framework with relatively little effort and now starts up and fails. The bulk of the efforts now

[3] It has been found that Google Chrome causes the driver to lock up after a few minutes of operation. The cause for this is not yet known but is being investigated, though is is known that Google Chrome is an extremely aggressive user of the network stack.

lie in reverse engineering the extensive and register and DMA interfaces present in the card. Two chipsets, (82958EB and 82599EB) are targeted at a bug finding test. Substantial effort is involved in building a functional hardware model of these chipsets and useful results are not yet available at the time of writing.

5. RELATED WORK

Userspace device drivers have a long history [8][9] and continues to be employed widely, especially in high performance situations [10][11]. Whilst Ultraviolet shares the basic concept of implementing a part of the device stack in userspace, it is distinct from previous attempts because it implements the hardware in userspace software rather than parts of software stack as is commonly the case. It is crucial to note that performance is not the primary goal of Ultraviolet, rather, the primary goal is rapid prototyping and testing at the software/ hardware interface.

The structure and function of Ultraviolet is highly similar to the virtual devices found in hypervisors and virtual machines (VMs). Both VMware [12] and Xen [13] 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 using Ultraviolet. 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 hardware and device driver.

Ultraviolet is also similar to File System in User Space (FUSE) [14] systems. Like Ultraviolet, FUSE requires that kernel become dependent on userspace applications. In contrast, however, FUSE systems are much simpler than Ultraviolet because no direct emulation of hardware timing, ordering and consistency parameters are required.

The most similar work to Ultraviolet is SymDrive[15]. This approach uses a complex set of symbolic and virtual execution tools used to simulate execution of a device driver in the absence of a device. This leads to similar debugging ability to Ultraviolet. However, it does not have the useful ability to prototype and test a new driver against a speculated/non existent hardware interface as Ultraviolet does.

Another similar approach to device driver correctness involves reusing parts of the device hardware specification to generate a device driver stub. Then, using a custom shim in the operating system [12] to run the stub as the device driver. Whilst this is an elegant solution, the approach requires that hardware designers learn and use a specialised formal language for generating their tests, which is unlikely to see traction amongst device designers. This approach again lacks the facility to prototype and play with interface ideas before committing to the expensive and time consuming task of implementing the hardware and it's accompanying tests.

6. CONCLUSIONS AND NEXT STEPS

Ultraviolet is a simple but unique approach to a real problem. Work so far has indicated that the approach is valid and viable for a small subset of devices on a small subset of available buses. The framework is already being used as a measurement and debugging tool for real device drivers and for porting new device chipsets to old device drivers. The obvious next step is to expand number and types of devices and buses supported and to perform similar tests on these devices. Ultraviolet userspace hardware programs represent fully functional specifications against which device implementations and simulations could potentially be tested before release. Another next step may be to migrate the virtualised hardware model to a cycle accurate simulation to facilitate full, cycle accurate device and driver simulation and testing. Ultraviolet affords designers an unique opportunity to rapidly explore the software-hardware interface of new and existing designs at low cost. It is expected that large suites of Ultraviolet devices could be created to perform automated regression testing on large kernels such as Linux prior to release, ideally improving the quality of devices and the stability of the operating systems that we all use on a daily basis.

7. REFERENCES

[1] Matthew J. Renzelmann, Asim Kadav and Michael M. Swift, 2012. *SymDrive: Testing Drivers without Devices*, OSDI 2012

[2] Archana Ganapathi, Viji Ganapathi, and David Patterson, 2006. *Windows XP Kernel Crash Analysis,* LISA 2006

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler, *An Empirical Study of Operating Systems Errors,* SOSP 2001

[4] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Gernot Heiser, *Dingo: Taming Device Drivers,* EuroSys 2009,

[5] 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*. Microelectronic Systems Education (MSE '07).

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

[7] C. Thacker, A. Nowatzyk, T. Rodeheffer, and F. Yu. *A data center network using FPGAs* (v4.5), April 2011. Unpublished.

[8] 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.

[9] 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.

[10] Ian Pratt, Keir Fraser. 2001. *Arsenic: A user-accessible gigabit ethernet interface.* INFOCOM 2001.

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

[12] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. 2001. V*irtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor.* USENIX ATC 2001

[13] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. 2006.*Optimizing network virtualization in Xen.* USENIX ATC 2006

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

[15] Matthew J. Renzelmann, Asim Kadav and Michael M. Swif, *SymDrive. 2012, Testing Drivers without Devices,* OSDI 2012