A Windows Support Framework for the NetFPGA 2 Platform

Chen Tian[1,2], Danfeng Zhang[1], Guohan Lu[1], Yunfeng Shi[1], Chuanxiong Guo[1], Yongguang Zhang[1]
[1]Mircosoft Research Asia
[2]Huazhong University of Science and Technology
{v-tic, v-daz, lguohan, v-yush, chguo, ygz}@microsoft.com

ABSTRACT

The NetFPGA 2 platform is widely used by the networking research and education communities. But the current software package supports only Linux. This paper describes the development of a Windows Support Framework for the NetFPGA 2 platform. We present the Windows Support Framework design after we briefly introduce the Windows Network Driver Interface Specification (NDIS). We then describe the implementation details such as drivers structure, the packet send/receive procedures, and the user-mode tool kit. Experiments show that our implementation achieves 160Mb/s sending rate and 230Mb/s receiving rate, respectively. We hope the Windows Support Framework brings NetFPGA to those researchers and students who are familiar with the Windows operating system.

1. INTRODUCTION

The NetFPGA 2 platform enables researchers and students to prototype high-performance networking systems using field-programmable gate array (FPGA) hardware [5]. As a line-rate, flexible, and open platform, it is widely accepted by the networking community: over 1,000 NetFPGA systems have been shipped and many innovative networking designs have been implemented [1, 3]. But currently the platform only supports Linux, and the developments of NetFPGA projects are limited to the Linux environments. Given the dominant market share of the Windows operating system, adding support for Windows will help those researchers and students who are familiar with the Windows system.

In this paper, we describe the design and implementation of our Windows Support Framework(WSF) for NetFPGA 2. The WSF is driven by both the requirements of our Windows-based testbed in Microsoft Research Asia and the community benefits. Windows Support Framework has two components:

- *Kernel Drivers.* A suit of kernel mode device drivers that enable the deployment of NetFPGA 2 platform in the Windows operating system.
- *User-Mode Tool Kit.* Common development functions such as registers reading/writing, FPGA bitfile downloading, and packet injecting/intercepting, are implemented as user mode tools.

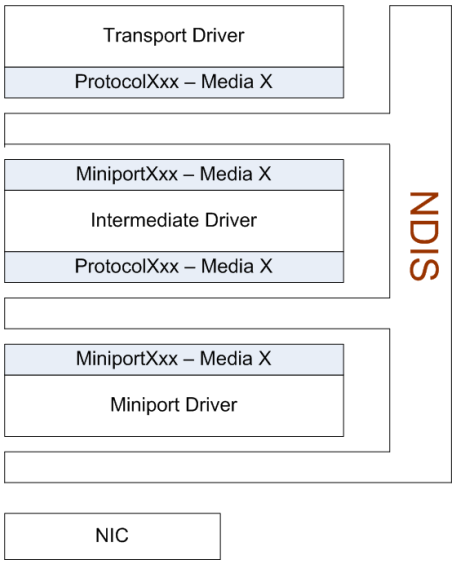

Figure 1: NDIS architecture

The rest of the paper is organized as follows. In Section 2, we first briefly introduce the Windows network driver architecture, and then present the support framework design. We describe the implementation details such as important routines of kernel drivers and the packet send/receive procedures in Section 3. We present experimental results in Section 4. Section 5 concludes the paper and discusses future work.

2. DESIGN

2.1 Windows Network Driver Architecture

The Windows operating system use Network Driver Interface Specification (NDIS) architecture to support network devices and protocols. Based on the OSI seven-

layer networking model, the NDIS library abstracts the network hardware from network drivers. NDIS also specifies the standard interfaces between the layered network drivers, thereby abstracting lower-level drivers that manage hardware for upper-level drivers. To support the majority of Windows users, we design WSF drivers to conform with NDIS version 5.1, which is Windows 2000 backward compatible. Most drivers are written in Kernel-Mode Driver Framework(KMDF) style, which provides object-based interfaces for Windows drivers [4].

As we show in Fig. 1, there are three primary network driver types [4]:

- *Miniport Drivers.* A Network Interface Card(NIC) is normally supported by a miniport driver. An NDIS miniport driver has two basic functions: managing the NIC hardware, including transmiting and receiving data; interfacing with higher-level drivers, such as protocol drivers through the NDIS library. The NDIS library encapsulates all operating system routines, that a miniport driver must call, to a set of functions (*NdisMXxx()* and *NdisXxx()* functions). The miniport driver, in turn, exports a set of entry points (*MiniportXxx()* routines) that NDIS calls for its own purposes or on behalf of higher-level drivers to send down packets.

- *Protocol Drivers.* Transport protocols, e.g. TCP-/IP stack, are implemented as protocol drivers. At its upper edge, a protocol driver usually exports a private interface to its higher-level drivers in the protocol stack. At its lower edge, a protocol driver interfaces with miniport drivers or intermediate network drivers. A protocol driver initializes packets, copies sending data from the application into the packets, and sends the packets to its lower-level drivers by calling *NdisXxx()* functions. It must also exports a set of entry points (*ProtocolXxx()* routines) that NDIS calls for its own purposes or on behalf of lower-level drivers to indicate up received packets.

- *Intermediate Drivers.* Intermediate drivers are layered between miniport drivers and transport protocol drivers. They are used to translate between different network media or map virtual miniports to physical NICs. An intermediate driver exports one or more virtual miniports at its upper edge. To a protocol driver, a virtual miniport that was exported by an intermediate driver appears to be a real NIC; when a protocol driver sends packets to a virtual miniport, the intermediate driver propagates these packets to an underlying miniport driver. At its lower edge, the intermediate driver appears to be a protocol driver to an underlying miniport driver; when the underlying miniport driver indicates received packets, the intermediate driver propagates the packets up to the protocol drivers that are bound to its virtual miniport.

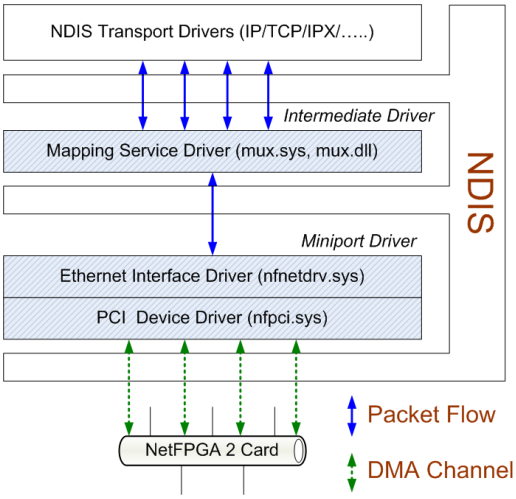

Figure 2: The structure of the kernel mode drivers.

2.2 The NDIS Kernel Drivers

As a PCI board, NetFPGA card has a memory space for PCI configuration information. The information describes the hardware parameters of the devices on the board. The configuration of the NetFPGA reference design contains only one PCI device. All four Ethernet ports share the same interrupt number, and tranmit/receive over their own DMA channels.

Supporting four NetFPGA Ethernet Ports in Linux is relatively simple. During the initialization phase, the Linux driver calls system routine *register_netdev()* four times to register NetFPGA ports as four distinct logical Ethernet devices.

Network driver support in NDIS 5.1 context is more sophisticated, mainly due to Plug-and-Play (PnP) and power management. The initialization process of a network device is managed by the Plug-and-Play(PnP) manager, hence one NetFPGA card can register only one logical Ethernet device. Apparently, one miniport driver alone is not enough to support four ports of NetFPGA in NDIS 5.1 context. We need an additional intermediate driver to map one single NetFPGA PCI card to four virtual Ethernet miniports.

As shown in Fig 2, the NDIS Kernel Driver of NetFPGA has three driver modules. From bottom-up, the PCI Device Driver (PDD) directly interfaces with the hardware and provides I/O services, such as DMA operations and access of registers; the Ethernet Interface Driver (EID) registers the NetFPGA card as a logical

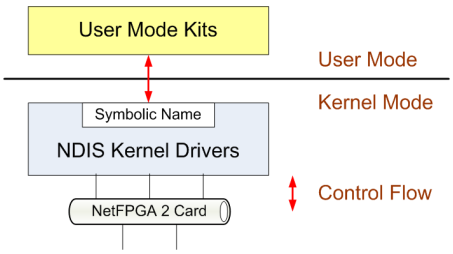

Figure 3: User mode/kernel mode Communication.

Ethernet device in response to PnP manager; these two drivers can be regarded as two submodules of a single NDIS miniport driver. The Mapping Service Driver (MSD) is an intermediate driver, which maps the underlying NetFPGA device to four virtual miniports and exports them to upper layer protocols.

The physical ports and virtual miniports maintain an exact match relationship. Every packet received by PDD is associated with its network port number; EID indicates the packet up together with the port; the MSD then indicates the packet to the corresponding virtual miniports. For the packet sending procedure, the order is reversed.

2.3 User-Mode Tool Kit

Besides kernel mode drivers, developers need to download FPGA bitfiles, or read/write registers for their own purpose. Basically, the main operations of bitfile download are also registers reading and writing. Communications between user mode tools and kernel mode drivers are needed to pass values up and down. As shown in Fig 3, each driver exports an associated symbolic device name: applications can open a driver handle by calling a *CreateFile()* routine; the communications can then be performed by calling *DeviceIoControl()* function and passing the I/O control commands down and reading the reported data back to applications.

To facilitate packet analysis of user derived protocols, packet injecting/intercepting functions are also implemented. All these implementation details will be given in the next section.

3. IMPLEMENTATION

This section gives implementation details. First the important routines of the three drivers are presented; then the complete packet send/receive procedures are illustrated to help understanding the asynchronous interactions among drivers; finally the implementation details of the development tool kit are also presented.

3.1 PCI Device Driver

The PCI Device Driver takes KMDF *PCI9x5x* example of Windows Driver Kit [4] as its template. PDD provides asynchronous hardware access interfaces to its upper layer EID module. The important routines are listed below:

- During device initialization, callback routine *PCIEEvtDeviceAdd()* is called in response to Windows' PnP manager. The routine registers all the callbacks and allocates software resources required by the device; the most important resources are *WriteQueue* and *PendingReadQueue*, which will serve write/read requests later.
- After the PnP manager has assigned hardware resources to the device and after the device has entered its uninitialized working (D0) state, callback routine *PCIEEvtDevicePrepareHardware()* is called to set up the DMA channels and map memory resources, make the device accessible to the driver.
- Each time the device enters its D0 state, callback routine *PCIEEvtDeviceD0Entry()* is called just before the enable of hardware interrupt; the NetFPGA card registers are initialized here.
- When NetFPGA generates a hardware interrupt, the driver's Interrupt Service Routine (ISR) *PCIEEvtInterruptIsr()* quickly save interrupt information, such as the interrupt status register's content, and schedules a Deferred Procedure Call (DPC) to process the saved information later at a lower Interrupt Request Level(IRQL).
- DPC routine *PCIEEvtInterruptDpc()* is scheduled by ISR. This routine finishes the servicing of an I/O operation.
- In response to a write request, callback routine *PCIEEvtProgramWriteDma()* programs the NetFPGA device to perform a DMA transmit transfer operation.

3.2 Ethernet Interface Driver

The Ethernet Interface Driver takes KMDF *ndisedge* example as its template. To its lower edge, it interfaces with PDD by I/O request packets (IRPs) operations; to its upper edge, it acts as a standard Ethernet device. The important routines are listed below:

- The driver's entry routine *DriverEntry()* calls NDIS function *NdisMRegisterMiniport()* to register the miniport driver's entry points with NDIS.
- Callback routine *MPInitialize()* is the entry point of Initialize Handler. Called as part of a system PnP operation, it sets up a NIC for network I/O operations, and allocates resources the driver needs to carry out network I/O operations.

- Callback routine *MPSendPackets()* is the entry point of Send Packets Handler. An upper layer protocol sends packets by calling NDIS function *NdisSendPackets()*. NDIS then calls this routine on behalf of the higher-level driver. This routine prepares write resources and initiates a write request. The port number is associated with the request by *WdfRequestSetInformation()* operations.
- *NICReadRequestCompletion()* is the completion routine for the read request. This routine calls NDIS function *NdisMIndicateReceivePacket()* to indicate the received packet to NDIS. The receive port number is associated with the packet by saving it in the *MiniportReserved* field of the NDIS_PACKET structure.

3.3 Mapping Service Driver

The Mapping Service Driver takes the famous *MUX* intermediate driver as its template. A *MUX* intermediate driver can expose virtual miniports in a one-to-n, n-to-one, or even an m-to-n relationship with underlying physical devices. One challenge is how to configure the protocol binding relationships: only a NetFPGA card is legal to be bound to this intermediate driver and to export four virtual miniports. We achieve this goal by modifying the accompanying installation DLL component.

The important routines are listed below:

- Callback routine *MPInitialize()* is the entry point of virtual miniport Initialize Handler. The MAC addresses of virtual miniports are read from the corresponding registers during the initialization phase.
- Similar to its counterpart of EID, callback routine *MPSendPackets()* is the entry point of Send Packets Handler. The send port number is associated with the packet by saving it in the *MiniportReserved* field of NDIS_PACKET structure.
- Callback routine *PtReceivePacket()* is the entry point of Receive Packet Handler. This routine associates each packet with its corresponding virtual miniport and call NDIS function *NdisMIndicateReceivePacket()* to indicate it up.

3.4 Packet Sending Procedure

Fig 4 gives the life cycle of a sending packet.

1. When an application wants to send data, the upper layer transport protocol prepares packets and calls NDIS function *NdisSendPackets()*; a packet is passed by NDIS to a corresponding virtual miniport interface of MSD.
2. NDIS calls MSD's callback routine *MPSendPackets()* with the packet; this routine associates the corresponding send port with the packet, then call *NdisSendPackets()* to pass the packet down to EID.
3. The NIC write request can be initiated asynchronous when a packet is ready. NDIS calls MSD's callback routine *MPSendPackets()* with the packet. This routine prepares write resources and initiates an asynchronous write request to PDD.
4. Upon receiving a write request, the PDD callback routine *PCIEEvtProgramWriteDma()* acquires a transmit spinlock to obtain DMA control; after that, a new DMA transmit transfer can be started.
5. After the completion of the DMA transmit transfer, the INT_DMA_TX_COMPLETE bit is set in a physical interrupt; the ISR reads the status and schedules a DPC; the DPC routine informs EID of the write request completion and releases the spinlock.
6. The EID's completion routine for the write request *NICWriteRequestCompletion()* is called; it frees write resources and calls NDIS function *NdisMSendComplete()* to inform the upper layer MSD.
7. Consequently the MSD's send completion callback routine *PtSendComplete()* is triggered by NDIS, and it also calls *NdisMSendComplete()* to inform its upper layer transport protocol.
8. On behalf of MSD, NDIS calls the upper layer protocol's callback routine *ProtocolSendComplete()*, the packet send process is finally completed.

3.5 Packet Receiving Procedure

For packet receiving, Fig 5 gives the life cycle of a packet.

1. After initialization, EID posts a sequence of NIC read requests to PDD in advance.
2. When a packet arrives, the INT_PKT_AVAIL bit is set in a physical interrupt; the ISR reads the status and schedules a DPC; the PDD DPC routine dequeues a read request, acquires a receive spinlock to obtain DMA control; after that, a new DMA receive transfer can be started.
3. After the packet is received, the INT_DMA_RX _COMPLETE bit is set in a physical interrupt; the ISR reads the status and schedules a DPC; the PDD DPC routine informs EID of the read request completion and releases the spinlock.
4. The EID's completion routine for the read request *NICReadRequestCompletion()* is called; the routine associates the corresponding receive port with the packet, then calls NDIS function *NdisMIndicateReceivePacket()* to inform its upper layer driver MSD.

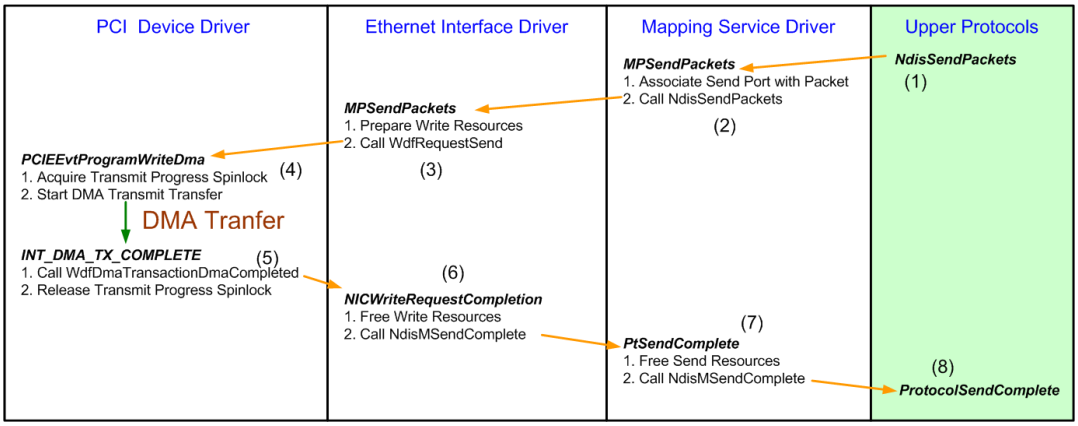

Figure 4: Packet sending procedure.

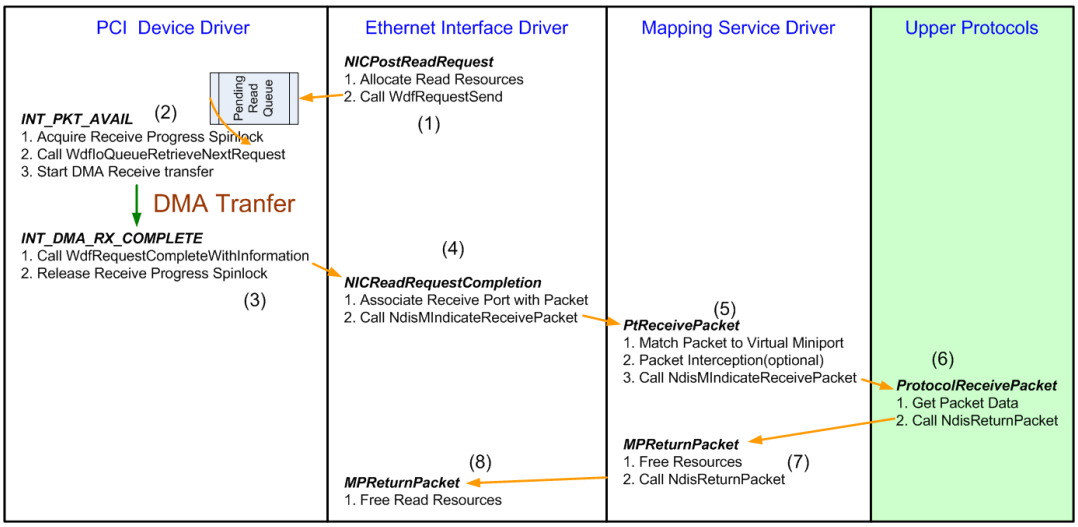

Figure 5: Packet receiving procedure.

5. The MSD's callback routine *PtReceivePacket()* is called by NDIS; the routine matches the packet to its corresponding virtual miniport and also indicates the packet up by NDIS function *NdisMIndicateReceivePacket()*;

6. After the packet data is received, the upper layer protocol calls NDIS function *NdisReturnPacket()*.

7. Consequently the MSD's receive completion callback routine *MPReturnPacket()* is triggered by NDIS, and it also calls *NdisReturnPacket()* to inform its lower layer driver EID.

8. The EID's callback routine *MPReturnPacket()* is called by NDIS, and the packet receive process is finally completed.

3.6 Implementation Details of the Tool Kit

The registers read and write operations are shown in Fig 6(a): an I/O control command is issued to EID first; the EID then builds an internal IRP and recursively calls the lower PDD to complete the request.

The packet injecting/intercepting functions are implemented in the MSD, as shown in Fig 6(b). After reception of a packet from the application level tool, the routine *PtSend()* routine injects the packet into outgoing packet flow path. A boolean value is associated with each packet send request to distinguish the injected packets from normal packets; in step (7) of Fig 4 when callback routine *PtSendComplete()* is called, only the completion of normal packets are required to be reported to upper layer protocols.

The packet interception logic is embedded in step (5)

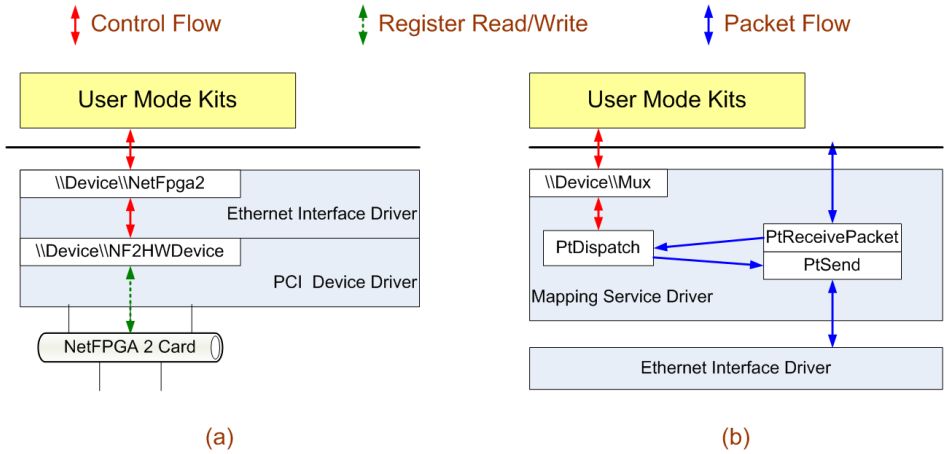

Figure 6: (a) Register Read/Write. (b) Packet Intercept/Inject.

of Fig 5. The interception kit posts a sequence of read IRPs to MSD in advance. In routine *PtReceivePacket()*, every incoming packet is checked first; the decision can be drop, modify, copy up to user mode kit, or continue without intervention.

4. FOR USERS OF WSF

4.1 Package

The package of WSF can be downloaded from our group web site [2]. The source code of both NDIS kernel drivers and user mode tool kits are included. Users who want to write a Windows program that interfaces with the NetFPGA can take the source code as their template.

An installation manual is included [6] in the package to guide the preparation of users and illustrate the use of the WSF from a users perspective. Some usage examples are also given for tool kits in the manual .

4.2 Performance

We build the source code using Microsoft Windows Device Driver Kit (DDK) Release 6001.18001 version. The experiment servers are DELL optiplex 755 with Windows Server 2003 Enterprize Edition with service pack 2. Each server has a 2.33G E6550 Intel Core 2 Duo CPU and 2 GB memory. The *iperf* software [7] is selected to perform the test with the 1500 bytes packets.

The forwarding performance is not affected by operating systems. So we only test the send/receive throughput of NetFPGA equipped host. In our experiment, the two servers connect to a gigabit switch using one NetFPGA port. Both 32-bit/64-bit performance are evaluated. The results are shown in Table. 1.

The host throughput achievable by Windows drivers are lower than that in Linux environment. The reason is that we implement the NetFPGA roles(PCI device/Network device/Port Mapping) to three separate drivers: each send/receive packet must pass through three drivers. A future version(Discussed in Section 5) may compact all functions to a single driver to improve host throughput. However the NetFPGA community is focused on hardware forwarding engine, which may make this throughput performance still acceptable.

Throughput	Send (Mb/s)	Receive (Mb/s)
Win2003 x86	158	234
Win2003 x64	156	233
Linux	186	353

Table 1: Throughput of experiments.

5. CONCLUSION

We have presented the design and implementation of a Windows Support Framework for the NetFPGA 2 Platform. We hope the Windows Support Framework brings NetFPGA to those researchers and students who are familiar with the Windows operating system.

The Windows Support Framework for NetFPGA 2 Platform is still in active development. Our future works may include but not limit to:

- *Migrate to NDIS 6.0.* Many new features are provided in NDIS 6.0 version, and we have a chance to compact all functions to a single driver and at the same time improve performance.
- *Support new NetFPGA hardware.* We plan to port our implementation to support the upcoming 10G NetFPGA once it becomes available.

6. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[2] Data Center Networking at MSRA. http://research.microsoft.com/en-us/projects/msradcn/default.aspx.

[3] Dilip Antony Joseph, Arsalan Tavakoli, Ion Stoica, Dilip Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers, 2008.

[4] MSDN. Microsoft windows driver kit (wdk) documentation, 2008.

[5] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable Router Architecture for Experimental Research. In *PRESTO*, 2008.

[6] Chen Tian. Netfpga windows driver suite installation.

[7] Ajay Tirumala, Les Cottrell, and Tom Dunigan. Measuring end-to-end bandwidth with iperf using web100. In *Proc. of Passive and Active Measurement Workshop*, page 2003, 2003.