IRLSTTTACK 3.0: HIGH-PERFORMANCE WINDOWS SOCKETS

A Thesis

by

YUE ZHUO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2014

Major Subject: Computer Science

ABSTRACT

With the ever-growing volume and speed of Internet traffic, network applications place higher demand on packet I/O rates. Although 1-Gbps and even 10-Gbps Ethernet are widely adopted, achieving wire rate with small packets remains hindered by bottlenecks inside the TCP/IP stack. Improvements have been made for Linux, but there is still limited work in Windows. To bridge this gap, we build a new generation of our network driver IRLstack and show that it can achieve 10 Gbps wire rate (i.e. 14.88 Mpps), for both send and receive, with zero CPU utilization. This compares favorably to the fastest Linux versions, which typically saturate one or more CPU cores and often fail to achieve this rate in both directions.

DEDICATION

To my parents

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Dr. Loguinov for giving me the opportunity to work with him, who is the source of inspiration of this work. He teaches the most impressive and intriguing courses in our department, and always has insightful thoughts about research. The experience of working with him will guide me in my future career. I would also like to thank Dr. Bettati and Dr. Reddy for serving on my committee.

I would like to extend my thanks to my colleagues at Internet Research Lab, to Xiaoyong Li and Yi Cui for encouraging me to enter network area, to Patrick Webster for patient guidance to NDIS development, to Zain Shamsi, who was the first user of this work, to Tanzir Ahmed, who helped me setup the test servers. It is my fortune to work with these wonderful guys.

Finally, I gratefully acknowledge my parents for their support and encouragement throughout my whole life..

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Many Internet studies require high-performance networking. Examples include active probing during Internet-wide scanning [1, 8, 16], wire-rate capture at security devices (e.g., firewalls [13], IDS [14]), large-scale web crawling [7], and software routers [2]. However, the default TCP/IP stack in commodity operating systems is not well-designed for fast I/O, especially when packet size is small. Using Windows as our main target, Table 1.1 shows how poorly Winsock performs with a 1-Gbps NIC. These numbers are similar to those in [17], but with a more-modern and faster CPU. Observe in the table that sending to the same destination maxes out at 51% of the wire rate and saturates all six cores. Producing packets towards *unique* destinations is much worse and at best achieves 3.5%. The latter case also shows major scalability issues when the application runs on multiple cores – *higher CPU utilization produces lower speed!*

1.1 Building TCP/IP stack

The issue of partially (or entirely) bypassing the TCP/IP stack is well-researched in Linux. In Windows, the issue of building a custom stack is more challenging since one has access to the source code of neither the NIC device driver nor the kernel. Network drivers must be written using a set of APIs called *Network Driver Interface Specification* (NDIS), which acts as a wrapper for accessing the hardware. NDIS employs a strict layering architecture where drivers can exchange packets only if they are in adjacent layers. This is illustrated in Figure 1.1. Application data is transmitted through Winsock APIs to the *Ancillary Function Driver* (AFD), which is responsible for buffer management and communication with the requested protocol driver. Under normal conditions, AFD forwards data to the TCP/IP driver `tcpip`.

Socket type	Destination	1 core	2 cores	3 cores	6 cores
Regular UDP	single	337,484	443,674	575,325	758,179
	all unique	53,023	23,517	17,468	18,744
Raw UDP	single	226,768	290,626	393,359	620,192
	all unique	50,292	25,052	20,489	18,517
Raw IP	single	141,546	199,350	262,647	408,350
	all unique	44,704	23,019	17,325	16,082

Table 1.1: Rate in pps while sending 64-byte UDP packets in Winsock at 100% CPU utilization of each core (AMD Phenom II @ 2.8 GHz, Intel PRO/1000 PT adapter, 1.488 Mpps wire speed).

`sys`, which consists of the transport, network, and link layers. Packets created by the protocol driver pass through the underlying filter drivers, which can monitor and modify both incoming and outgoing packets. At the bottom of NDIS are the miniport drivers, which interact with the hardware to configure DMA requests and respond to interrupts.

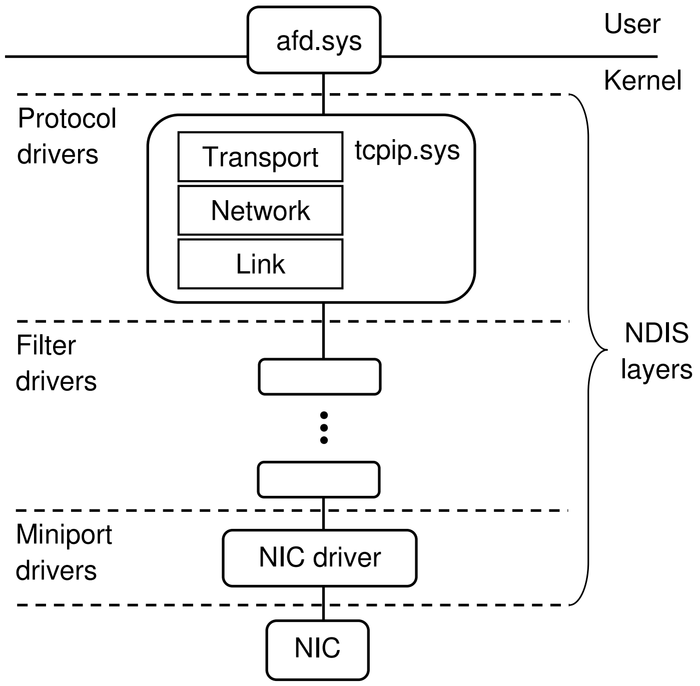

Figure 1.1: Windows network architecture.

2. RELATED WORK

2.1 Packet I/O Engine

Packet I/O Engine [2] is a set of custom device drivers for two 10-Gbps Intel chipsets (i.e., 82598 and 82599), utilizing such optimizations as batch processing, software prefetch, multiple rx/tx queues, and Receive Side Scaling (RSS) for multi-core systems. User applications communicate with these drivers using `ioctl` primitives of the OS.

2.2 PF_RING

PF_RING is a highly optimized Linux network stack developed by the ntop project. Their early work, called *Vanilla* [10], pulls packets from the NIC into a circular buffer via Linux's NAPI. User applications subsequently retrieve data from this ring into their memory space. Follow-up work, known as *Direct NIC Access* (DNA) [11], is a faster driver that maps NIC registers to user space and operates through DMA. On top of this driver, a library is provided for packet distribution across threads and processes. Due to its commercial nature, the user-space library of PF_RING DNA is not open-source [12].

2.3 Netmap

Netmap [15] is a framework similar to the Packet I/O Engine that is built into the kernel. It communicates with clients through custom system calls and supports several Intel, Realtek, and Nvidia adapters. Recent FreeBSD (FreeBSD HEAD and stable/9) already includes netmap in the source tree.

2.4 IRLstack

IRLstack 1.0 [17] is a set of two Windows drivers as shown in Figure 2.1 where packets are first transmitted from user space to a special protocol driver, which then forwards them to a custom filter driver. Data transfers are handled entirely through NDIS, which makes it suitable for any NIC that has an installed miniport driver. While IRLstack 1.0 can saturate a 1 Gbps link in certain cases and achieve 3.5 Gbps with quad-port NICs, its frequent calls to NDIS limit performance and its dual-driver kernel architecture is difficult to maintain/extend. IRLstack 2.0 [18] reduces the number of interrupts generated in the previous version and streamlines the user interface, but does not offer significant performance improvements. Needless to say, neither the 1.0 nor 2.0 driver can handle 10-Gbps rates.

2.5 WinPcap

WinPcap [19] is a popular tool for network capture and transmission in Windows. It includes a filter driver, a low-level interface library, and a high-level system-independent library. In theory, it should be faster than the standard Winsock, but results show [17] that it often performs no better and sometimes even worse.

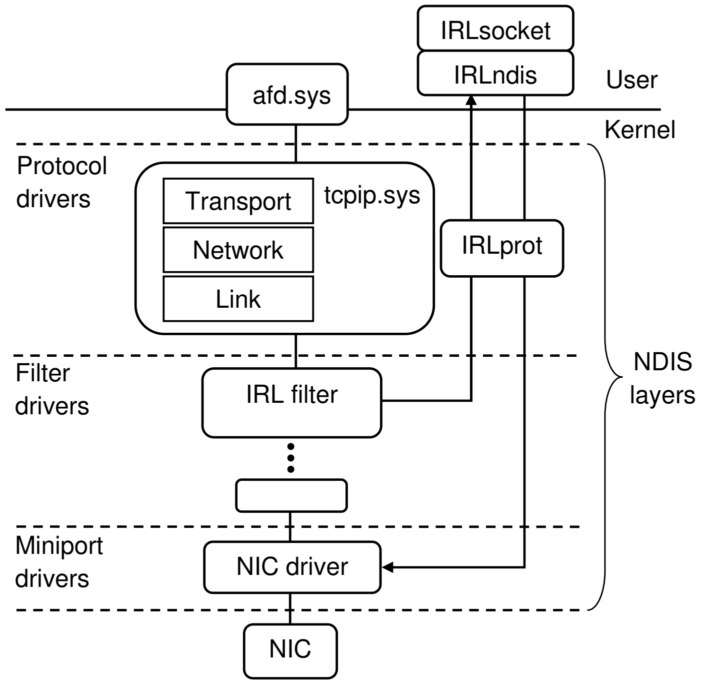

Figure 2.1: IRLstack 2.0 architecture.

3. IRLSTACK 3.0

In the design of network stacks, there is an inherent tradeoff between speed and generality. Compatibility with *all* types of adapters supported by the OS requires usage of the manufacturer's miniport (device) driver and kernel NDIS primitives. In many cases, these two contain enough bottlenecks to prevent wire rate operation even at 1 Gbps. To achieve the highest performance, user applications must communicate directly with the NIC, which limits the generality of the provided solution. Therefore, it is customary to provide two drivers – one that relies on the OS to move packets to/from the device driver and the other that completely bypasses the kernel. This architecture is implemented in PF_RING under the names of Vanilla and DNA, respectively. We keep notation the same for simplicity and now build these two architectures in Windows.

In both methods, some portions of RAM must be shared between kernel and user space. This not only improves performance by eliminating per-packet system calls, avoiding interrupts, and reducing the number of memory copies, but also allows building of network protocols entirely in user space. This simplifies the development process by avoiding extensive kernel debugging. There are two basic methods to share memory. One is to create a buffer in a user application and pass it to the driver as the output buffer parameter via an `ioctl` with overlapped (i.e. asynchronous) structure. The user buffer is then checked for correct access and locked in the memory. The driver keeps the IRP (I/O Request Packets) generated by the ioctl command pending in order to share the output buffer between kernel and user space. Only when the driver no longer needs to share the buffer, it sets the IRP complete. The other is to allocate pages in the driver and map them into the application address space. This scheme is easy to use as it only needs to call several Windows APIs. The driver allocates non-paged memory using the `MmAllocatePagesForMdl` API (`MmAllocatePagesForMdlEx` for Windows Server 2003 Service Pack 1 and later), which returns an memory descriptor list (MDL) describes the allocated memory. Then the driver maps the allocated pages to kernel space via the `MmGetSystemAddressForMdlSafe` API. To share the memory, the driver calls the `MmMapLockedPagesSpecifyCache` API with `UserMode` as its `AccessMode` parameter within the context of the user process and return the user virtual address when the application issues an ioctl. If implemented correctly, both methods are safe when user processes exit accidently. However, for easy sharing of memory regions across multiple user processes, we chose the latter approach.

3.1 IRLstack 3.0 Vanilla

Our first contribution is to redesign IRLstack 1.0 to move data from kernel to user space without using IRPs, which is a framework of system calls for applications to retrieve pending packets from kernel space in Windows. This in turn allows elimination of the protocol driver and leads to simplification of the entire architecture. As shown in Figure 3.1, IRLstack 3.0 only runs a filter driver that shares its circular buffer with a user library we call IRLthread, whose purpose is to demultiplex packets to individual IRLsockets. Since this configuration relies on the miniport driver for both send/receive, it is compatible with all hardware that Windows normally supports.

3.1.1 Receiving packets

In NDIS 6.x, network packets data are represented in `NetBuffer` (NB) and `NetBufferList` (NBL). Every packet is wrapped in NB structure. A NBL may contain a group of NBs with each NB representing one packet, and multiple NBLs

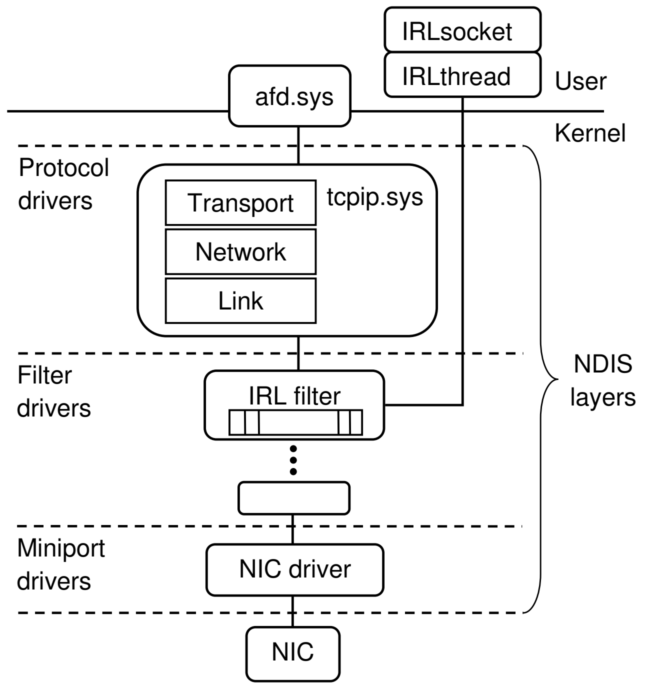

Figure 3.1: IRLstack Generic kernel architecture

can be chained together. In the receive path, each NBL contains exactly one NB. If an incoming packet is not IP (e.g., ARP, IPX, AppleTalk) or is destined to the primary IP address of the host, it is passed through to `tcpip.sys`. Otherwise, it is redirected to IRLthread by removing the corresponding NBL from the chain and copying the data from the NB to the shared circular buffer with a frame header specifying its length. This logic allows IRLstack 3.0 to transparently co-exist with the networking ecosystem of the operating system (e.g., remote desktop, domain authentication, web browsing). All experiments that require IRLstack must utilize secondary IP addresses. For example, our Internet scanning [8] usually runs with 61-123 IPs aliased to the same NIC, in addition to the primary IP.

3.1.2 Sending packets

For departing packets, user applications write data into the shared circular buffer with a special frame header in the front indicating the length of the packet. The filter operates on a timer that allows it to poll the tail of the queue and wrap the available packets into a corresponding NB structure for consumption by NDIS. We utilize NDIS' ability to transmit multiple packets by placing them into a single NBL, where the optimal batch size is close to 150 packets. This number provides a good balance between the frequency of interrupts generated by the miniport and artificially introduced delays in the send path. To avoid CPU-intensive allocation and deallocation of NBs for every packet, we maintain a free list of NBs, which is consulted every time the filter reads a packet from the user. If the free list is empty, a new NB is created. To ensure cached NBs contain enough space for outgoing packets, they are spawned with a pre-allocated data buffer of size larger than the maximum transmission unit (MTU). Since the NBs and NBLs we send are allocated from our own `SendNetBufferPool` and `SendNetBufferListPool`, when the miniport indicates completion of a chain of NBLs, we are able to check their `NdisPoolHandle` to know which NBLs are originated from our filter driver. If a NBL is owned by us, it is removed from the chain and its individual NBs are returned to the free list. Results show that the best performance is achieved with approximately 120 pending NBLs.

IRLstack 3.0 Vanilla depends on the miniport driver's overhead and the various bottlenecks in the NDIS layer. As we show later, these are quite expensive. An extra limitation of this approach is the *single-threaded* operation of the miniport for each NIC device, which prevents performance increase from utilizing multiple cores.

3.2 IRLstack 3.0 DNA

To bypass the miniport, our second contribution is to use the filter to allow the application to modify NIC registers and control DMA transfers. The current version works for popular 10-Gbps adapters based on the Intel 82598, 82599, and x540 chipsets. In Linux, all of the existing work [2, 11, 15] incorporates additional functionality directly into the source code of the Intel IXGBE driver. In Windows, however, this is impossible. Our solution is to leave the Intel miniport responsible for configuring NIC parameters during boot. Our filter driver retrieves the base physical address of NIC registers (by sending the `IoGetDeviceProperty` request to the miniport) and maps it to kernel space. Once this is done, the filter can access any register of the NIC, obtain the necessary pointers from it, referring to the Intel datasheet of each chipset [3, 4, 5], and make the registers available to the application. After the filter maps the register information to the applications, data in both direction is directly exchanged between the users and the NIC, ultimately bypassing the entire NDIS stack.

3.2.1 NIC operation

NICs manage packets through descriptors queues (also called *rings*) shown in Figure 3.2. The physical address of the queue (i.e., base) and its head/tail offsets are stored in NIC registers that are readable/writable by the host. Each entry in the ring has a pointer to the physical address of the packet buffer. In the send path, the NIC monitors the tail register and uses DMA to fetch outgoing packets (first their descriptors, then the corresponding payload). Completion information is written back to the descriptors using DMA and the head register is advanced to indicate availability of the slot to the host. In the receive path, operation is similar, except the role of the head/tail offsets is reversed. Transmit descriptor has legacy

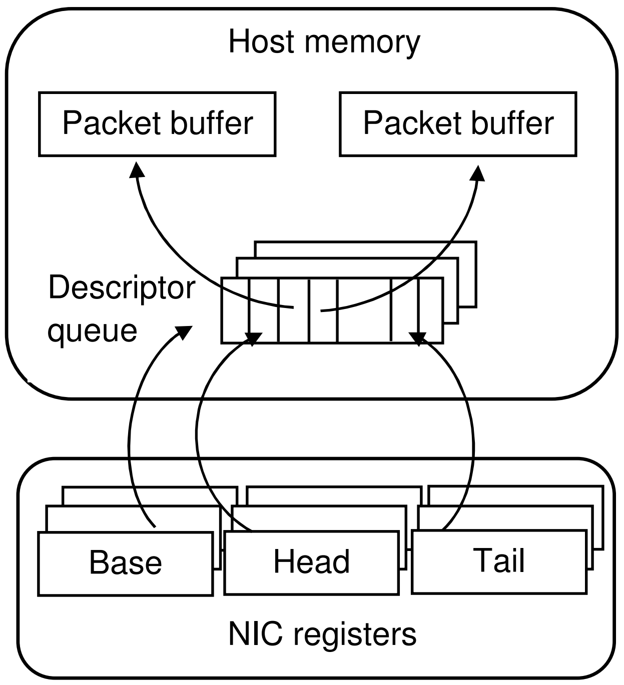

Figure 3.2: NIC operation

and advanced format. We choose to use legacy format since it is more efficient, achieving 10% speedup according to our test.

3.2.2 Setup

Since the miniport does not expect other programs to manipulate descriptor queues, there may be race conditions between IRLstack and the miniport, which often lead to an OS crash. Fortunately, the miniport uses only the first two queues, leaving the other ones available for contention-free modification. Considering that modern NICs support between 32 to 128 TX/RX queues (depending on the chipset), loss of two of these is insignificant in practice. To avoid expensive allocation/deallocation of packet buffer for every user application, we also create a data ring for each descriptor queue. For each descriptor and data ring, the filter driver of IRLstack in Figure 3.3 first allocates buffers using contiguous non-pageable kernel memory via the `MmAllocateContiguousMemorySpecifyCache` API, then creates MDL via the `NdisAllocateMdl` API for the memory and finally maps it to user processes that control the actual I/O. Each data queue has the same number of entries as the corresponding descriptor ring, where each packet is pre-allocated enough space to fit the largest frame. Due to the 1-KB alignment required by Intel NICs, 1500-byte MTUs occupy 2-KB buffers. Finally, these TX/RX queues are enabled when our filter driver writes their information (e.g., physical address, range, etc.) to the corresponding registers. To map each pair of descriptor and data ring to different user processes, the filter stores all of the mapping information with thread handle as identifier in a double linked list. Once a user process exits, the filter gets the notification and unmaps the buffer within its context according to the record in the list. Since the Windows miniport takes a conservative route with various performance features of Intel cards, our filter driver must configure several per-queue features to achieve better performance. We discuss these next.

3.2.3 Performance features

3.2.3.1 Descriptor prefetch and written-back

All 10-Gbps Intel chipsets maintain an internal buffer for TX descriptors of size 64 for the 82598 chipset and 40 for 82599 and x540 chipsets. While the Intel 82598 chipset also has an buffer for RX, the Intel 82599 and x540 chipsets eliminate the on-chip buffer for reception. When this buffer is empty, a *fetch* occurs as soon as any descriptor is available. A *prefetch* occurs when the on-chip buffer has fewer than `TXDCTL.PTHRESH` unprocessed entries and there are at least `TXDCTL.HTHRESH` descriptors available in host memory. To make the best use of the PCIe bandwidth, we set the former to a large number (i.e., half of the internal buffer size) to make prefetch more aggressive and the latter to a small number (i.e., 1) to reduce latency.

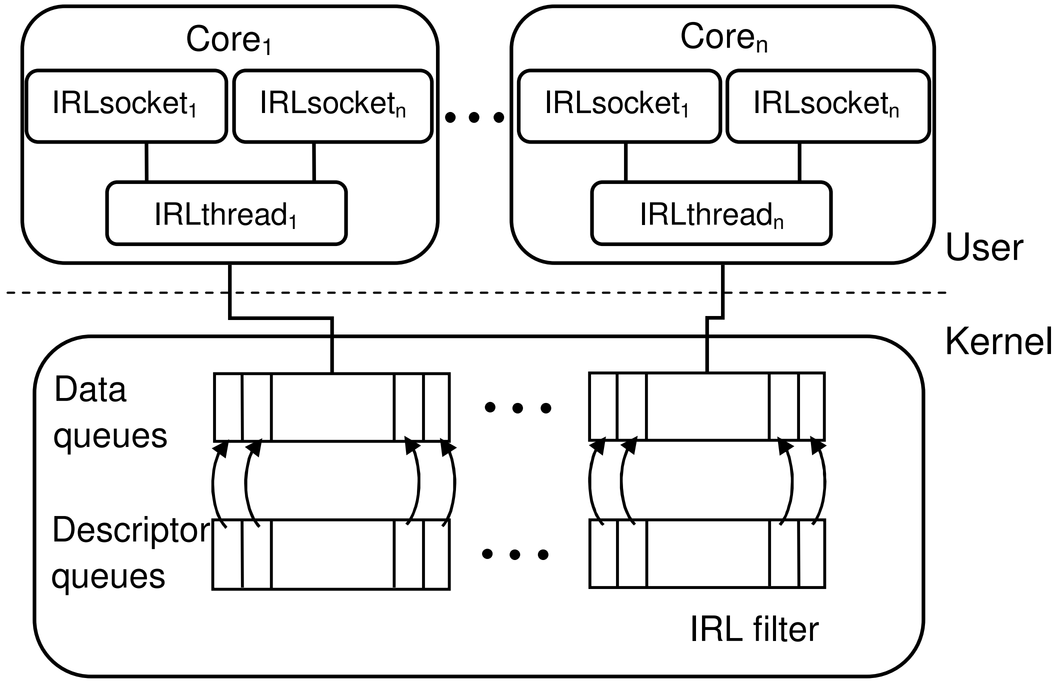

Figure 3.3: IRLstack custom architecture

This improves TX speed by 75%. In addition, the CPU cache line is typically larger than the descriptor size (i.e., 16 bytes). Thus, writing back descriptors for each packet causes expensive partial cache-line updates. This can be avoided by enabling the NIC's *receive descriptor packing mechanism*, which writes them in batches whose size is controlled by register `TXDCTL.WTHRESH`. To maximize memory efficiency and keep the latency reasonable, we set this value to exactly one cache line (i.e., 4).

3.2.3.2 Relaxed ordering

Another crucial feature that the Windows miniport driver fails to enable is *relaxed ordering*, which allows completion of DMA transactions to have no ordering relationship with the reads/writes from the host CPU. More specific, the NIC may write a large number of data blocks, followed by a descriptor write-back indicating the availability of the data. When the user (we bypass the driver) sees the descriptor updated, it begins to process the data. It doesn't matter in what order the data blocks get committed to RAM, but the descriptor must be written after all of the data is in RAM, otherwise, the user may read the partially-updated data buffer after seeing the descriptor get updated. Therefore, the data writes can employ relaxed ordering, but the descriptor must be strictly ordered so that it will not pass the data writes. That is why Intel requires us to not to enable relaxed ordering for RX descriptor write-back. Since the number and size of data transactions are much larger than descriptor updates, the system sees high write-to-memory performance when relaxed ordering is enabled on the data transactions. For the Intel 82599 and x540 chipsets, this feature improves performance by 50% in the receive path. While similar functionality exists for the send path, it does not have much impact in practice.

3.2.3.3 Receive side scaling

Another important mechanism for incoming packets is Receive Side Scaling (RSS) whose purpose is to spread packets into multiple descriptor queues by applying a well-known hash function (i.e., Microsoft RSS) to the packet's connection 4-tuple. The seven least-significant bits of the hash are used as an index into a 128-entry *indirection table*, which specifies RX queues that handle each combination of the bits. The standard use of RSS is to assign each queue to a different CPU in order to increase processing speed. However, there is an additional purpose in IRLstack. To prevent the miniport from handling incoming packets and ensure that IRLstack intercepts all of them, we populate the entire RSS indirection table with our RX queues. If some packets (e.g., those destined to the primary IP) must reach the default Windows TCP/IP stack, IRLthread may return them to the filter for subsequent injection into NDIS.

3.2.4 Reducing overhead

DMA transfers in user space need to check completion bits of descriptors, memcpy data between socket and ring buffers, reset descriptors that have been consumed, and move the head/tail registers. Our experiments show that memcpy does not cost much CPU, but busy-spinning during polling does. To further reduce the overhead, our strategy is to periodically wake up and consume as many descriptors as possible. Writing the tail register for every descriptor is too much expensive. To reduce the cost and still give the NIC enough time to move the head register, we write it every 64 descriptor. In Windows, the granularity of the sleep function is 1 millisecond. Therefore, to achieve the wire rate of 14.88 Mpps, we need to consume 14,880 descriptors per wake-up. Since the maximum NIC-supported queue size is 16,384 packets, wire speed is feasible with just a single TX/RX queue. To reduce scheduling delay jitter, we run user processes at real-time priority. In cases when this is undesirable or the sleep resolution is much coarser, more queues may be needed.

3.3 IRLstack 3.0 in NDIS 6 framework

In this section, we give more details about how to build IRLstack 3.0 in NDIS filter driver. When the filter is loaded, its entry point `DriverEntry` is called for registering the driver and creating a device for users to communicate with. Function handlers including the attach handler `FilterAttach` and the detach handler `FilterDetach` are also assigned here. FilterAttach is called once for every adaptor to create filter's context, where we also add our initializations including allocating NetBufferPool and NetBufferListPool for Vanilla, and mapping NIC registers for DNA. When the filter stops, FilterDetach is called to free pools of Vanilla and TX/RX queues of DNA.

RegisterDevice is called in DriverEntry as well, inside which `DispatchTable` is created to register handlers for different IRP requests (i.e. IRP_CREATE, IRP_CLEANUP, and IRP_DEVICE_CONTROL). We set `DeviceIoControl` function in this table to reply IRP_DEVICE_CONTROL request which happens when users issue ioctl commands with the open device handle. We define several custom commands, through which users can get shared memories and mapped NIC registers. Another function is registered to reply IRP_CLEANUP request issued when a user close the device handle. Upon the request, we unmap the shared memory within the user's context.

While packets transmit and reception of DNA are built in user space, Vanilla keeps them in the filter driver. In the send path, a timer is created when users send ioctl to get the shared circular buffer via the `NdisAllocateTimerObject` API. A send function is assigned in its `TimerCharacteristics` parameter to poll data from the buffer. `FilterSendNetBufferListsComplete` is registered to reuse NBs as the `SendNetBufferListsCompleteHandler` in DriverEntry which is described in section 3.1.2. In the receive path, `FilterReceiveNetBufferLists` is registered to process incoming packets as the `ReceiveNetBufferListsHandler` in DriverEntry, which acts as described in section 3.1.1.

3.4 User interface

3.4.1 Receiving packets

As shown in Figure 3.3, IRLstack creates a number of TX/RX queues, typically one pair for each CPU core. Along the receive path, RSS distributes arriving packets into individual queues, which are then demultiplexed by IRLthread into multiple IRLsocket connection buffers. We reuse the 4-tuple hash provided by the NIC in descriptors to keep CPU utilization low in DNA mode. In Vanilla mode, we have to compute the 4-tuple hash by our own. To share buffers between IRLsocket and IRLthread, we use *named shared memory.* This allows any number of processes to instantiate IRLsocket connections. The name of each buffer consists of the adapter string and the hash, which creates a unique label for each connection that does not need to be explicitly communicated to IRLthread. IRLsocket implements Microsoft RSS hash function in order to compute the hash and creates the named shared memory for its connection. The RSS hash random keys stored in the NIC registers for the hash function are retrieved by IRLthread and shared across every IRLsocket via a global buffer (i.e. shared memory named by the adapter string). The process is similar in Vanilla mode except we share a set of different random keys generated by IRLthread. Since each packet size in the data queue is fixed at 2 KB and access is sequential, IRLthread tries to reduce the cache miss rate by using CPU prefetch on the next packet while memcpying the current one. Finally, since each connection buffer has one producer and one consumer, the receive path is lock-free.

3.4.2 Sending packets

In the send path, IRLsocket directly posts data to the corresponding TX queue using a lock provided by IRLthread. This is accomplished by mapping each TX queue to all IRLsocket processes that run on the same core. To reduce the work of IRLsockets, IRLthread fills up descriptors with necessary information (e.g., insert checksum or not), making each one point to the corresponding physical address in the data ring for every TX queue. What IRLsockets need to do is only memcpy data into the data ring and set the packet length in the descriptor. Since kernel-level mutex/semaphore primitives in Windows are inefficient, we build a custom mutex on top of basic interlocked CPU instructions. This mechanism first tries to acquire the mutex in user mode by busy-spinning one a shared variable for a number of iterations. If this is unsuccessful, it gives up and goes to sleep on a kernel-mode semaphore. To further amortize the cost of competing for the lock, IRLsocket generates data in 64-packet bursts. This batch processing also enables easy adoption of CPU prefetch for TX descriptor and data queues.

4. EXPERIMENTS

4.1 Setup

Our third contribution is to benchmark IRLstack under Windows 2008 R2 and compare it to PF_RING Vanilla/DNA and Packet I/O Engine in Linux 3.0.0, and Netmap in FreeBSD 9.3-RELEASE on the same hardware. Our test machines are both 6-core desktop processors – AMD Phenom II (2.8 GHz) and Intel i7-3930 (4.4 GHz). For 1 Gbps, we use a standalone quad-port Intel Pro/1000 PT and two motherboard NICs from the hosts mentioned above (Realtek 8168C and Intel 82579LM, respectively). For 10 Gbps, we examine a single-port AT2 (Intel 82598EB chipset), dual port x520-T2 (Intel 82599), and dual-port x540-T2 (Intel x540), listed in the order from the oldest to the newest. All the experiments except those for examining performance of IRLsocket use one IRLsocekt for send and receive (i.e. promiscuous mode). To measure pure speed, we run the experiments on directly connected cards in two machines with TX/RX flow control disabled and CPUs running on maximum frequency constantly.

4.2 Results

4.2.1 Vanilla performance

Table 4.1 shows performance of Vanilla versions of IRLstack and PF_RING where cells with wire rate number are highlighted. Note that the latter spawns two processes and its total CPU usage sometimes exceeds one full core (which explains CPU utilization above 100%). In all cases except the sending rate on 82579LM, IRLstack outperforms PF_RING while keeping CPU utilization equal or lower. PF_RING even crashes the OS in the receive path on the Realtek adaptor. More surprisingly, IRL-

Link Speed	NIC	CPU		IRLstack 3.0		PF_RING	
				Mpps	CPU	Mpps	CPU
1 Gbps	PRO/1000 PT	AMD	TX	1.486	100%	1.013	100%
			RX	1.250	42%	1.136	108%
	Realtek 8168C	AMD	TX	0.685	54%	0.405	115%
			RX	1.488	60%	crashed	
	Intel 82579LM	Intel	TX	0.412	84%	0.905	100%
			RX	1.483	36%	1.344	108%
10 Gbps	AT2	Intel	TX	1.350	100%	0.827	100%
			RX	9.023	100%	0.936	106%
	x520	Intel	TX	2.739	100%	0.807	100%
			RX	9.294	100%	1.140	108%
	x540	Intel	TX	3.724	100%	0.983	100%
			RX	9.643	100%	1.124	102%

Table 4.1: Performance of Vanilla drivers (CPU utilization is percent of one core).

stack's receive rate on 10-Gbps adapters is an order of magnitude higher than that of PF_RING.

4.2.2 DNA performance

The rest of the experiments are run on 10-Gbps adapters using DNA. Table 4.2 shows the result on the AMD system. Only Netmap can achieve wire rate on x520 with one core maxed out for receive. It also reaches wire rate for send on AT2 and receive on x540. Except that, none of the other methods can reach wire rate simultaneously in both RX/TX directions. What stands out is that IRLstack always uses zero CPU, achieving wire rate for send on x520 with one or two ports and on x540 with one port. PF_RING can receive at wire rate on x520 and x540 with one core maxed out, which is similar to Netmap, while the Packet I/O Engine never reaches wire rate. Table 4.3 shows the result on the Intel processor, where IRLstack outperforms the PF_RING and Packet I/O Engine on all three adapters, achieves full wire rate in both directions on x540, and keeps 0% CPU utilization. Although

NIC	Ports / cores	Dir	IRLstack 3.0		PF_RING		I/O Engine		Netmap	
			Mpps	CPU	Mpps	CPU	Mpps	CPU	Mpps	CPU
AT2	1	TX	14.80	0%	14.09	45%	14.01	100%	14.88	60%
		RX	13.34	0%	13.94	100%	7.97	26%	13.95	100%
x520	1	TX	14.88	0%	14.59	47%	13.12	100%	14.88	55%
		RX	14.34	0%	14.88	100%	14.64	100%	14.88	100%
	2	TX	29.76	0%	28.86	94%	24.90	200%	29.76	120%
		RX	27.40	0%	29.76	200%	29.14	196%	29.76	200%
x540	1	TX	14.88	0%	14.46	45%	not supported		13.38	45%
		RX	14.45	0%	14.88	100%			14.88	100%
	2	TX	29.04	0%	28.58	90%			27.06	100%
		RX	27.78	0%	29.76	200%			29.76	200%

Table 4.2: AMD performance of DNA drivers (CPU utilization is percent of one core).

Netmap has faster receive rate on x520 with one port, it is not as fast as IRLstack in any other cases.

Our next experiment is to examine scalability of the drivers operating with a single NIC port to multiple cores. As PF_RING DNA does not provide multi-core support and we fails to enable multiple TX/RX queues on Netmap, we only compare our work with the Packet I/O Engine. Table 4.4 shows their performance on the AMD processor. While IRLstack always keeps the CPU at zero utilization, its performance slightly degrades with parallelization. This was expected due to higher lock contention and non-sequential RAM access during memcpy. The Packet I/O Engine in some cases exceeds the speed of IRLstack, but this is accompanied by a significantly higher CPU overhead. Both methods can send at wire rate on x520, but neither can achieve it on receive. Table 4.5 shows the result on the Intel processor. Here, IRLstack scales much better and beats the Packet I/O Engine in almost all cases. It achieves wire rate for send on AT2, and in both directions on x520 using

NIC	Ports / cores	Dir	IRLstack 3.0		PF_RING		I/O Engine		Netmap	
			Mpps	CPU	Mpps	CPU	Mpps	CPU	Mpps	CPU
AT2	1	TX	14.88	0%	14.20	35%	14.20	100%	14.88	30%
		RX	14.12	0%	14.08	100%	5.49	13%	14.09	100%
x520	1	TX	14.88	0%	14.88	35%	14.21	42%	14.88	26%
		RX	14.80	0%	14.40	100%	14.58	42%	14.71	100%
	2	TX	29.76	0%	29.74	70%	26.64	100%	29.76	68%
		RX	29.42	0%	28.92	200%	29.10	100%	29.76	100%
x540	1	TX	14.88	0%	14.34	33%	not supported		13.58	27%
		RX	14.88	0%	14.88	100%			14.88	100%
	2	TX	29.76	0%	28.80	67%			27.18	68%
		RX	29.76	0%	29.76	200%			29.76	200%

Table 4.3: Intel performance of DNA drivers (CPU utilization is percent of one core).

four cores and x540 using any number of cores.

4.2.3 IRLsocket performance

The last experiment is to measure the performance of IRLsocket. We examine it on x540 using the Intel processor. For send, we run multiple IRLsockets that share per-core TX queue and lock. For receive, each IRLsocket representing one connection binds to the RX queue calculated by user-level RSS. The generator in another machine sends packets to different connections in round-robin way. Table 4.6 shows the send performance. We achieve wire rate using zero CPU with up to 400 processes on any number of cores. Table 4.7 shows the receive performance. Here, only one core is no longer suitable for running more than 100 connections. Fortunately, RSS mechanism enables us to make use of multi-core system, which scales quite well. When using three cores, it can reach wire rate for 400 processes with zero CPU.

		AT2 (82598EB chipset)							
		1 core		2 cores		4 cores		6 cores	
		Mpps	CPU	Mpps	CPU	Mpps	CPU	Mpps	CPU
IRLstack 3.0 DNA	TX	14.80	0%	14.68	0%	14.08	0%	13.89	0%
	RX	13.34	0%	13.41	0%	13.46	0%	13.48	0%
I/O Engine	TX	14.01	100%	13.68	200%	13.90	400%	13.76	595%
	RX	7.97	26%	10.06	30%	12.96	45%	12.97	50%
		x520-T2 (82599 chipset)							
IRLstack 3.0 DNA	TX	14.88	0%	14.88	0%	14.88	0%	14.88	0%
	RX	14.34	0%	14.36	0%	14.29	0%	13.67	0%
I/O Engine	TX	13.12	100%	14.47	200%	14.88	400%	14.88	599%
	RX	14.64	100%	13.51	68%	14.13	76%	14.12	75%
		x540-T2 (x540 chipset)							
IRLstack 3.0 DNA	TX	14.88	0%	13.49	0%	13.96	0%	13.99	0%
	RX	14.45	0%	14.42	0%	14.33	0%	14.23	0%

Table 4.4: AMD performance with multiple cores (CPU utilization percent of one core).

		AT2 (82598EB chipset)							
		1 core		2 cores		4 cores		6 cores	
		Mpps	CPU	Mpps	CPU	Mpps	CPU	Mpps	CPU
IRLstack 3.0 DNA	TX	14.88	0%	14.88	0%	14.88	0%	14.88	0%
	RX	14.12	0%	14.08	0%	14.06	0%	14.05	0%
I/O Engine	TX	14.20	100%	14.18	200%	14.18	400%	14.15	600%
	RX	5.49	13%	9.18	24%	13.70	40%	13.69	41%
		x520-T2 (82599 chipset)							
IRLstack 3.0 DNA	TX	14.88	0%	14.88	0%	14.88	0%	14.88	0%
	RX	14.80	0%	14.84	0%	14.88	0%	14.11	0%
I/O Engine	TX	14.21	100%	14.88	200%	14.88	400%	14.88	600%
	RX	14.58	42%	14.77	34%	14.84	45%	14.84	42%
		x540-T2 (x540 chipset)							
IRLstack 3.0 DNA	TX	14.88	0%	14.88	0%	14.88	0%	14.88	0%
	RX	14.88	0%	14.88	0%	14.88	0%	14.88	0%

Table 4.5: Intel performance with multiple cores (CPU utilization percent of one core).

# cores	**# processes**			
	100	**200**	**300**	**400**
1 core	14.88 (0%)	14.88 (0%)	14.88 (0%)	14.88 (0%)
2 cores	14.88 (0%)	14.88 (0%)	14.88 (0%)	14.88 (0%)
3 cores	14.88 (0%)	14.88 (0%)	14.88 (0%)	14.88 (0%)

Table 4.6: IRLsocket send performance on the Intel processor (CPU utilization percent of one core).

# cores	**# processes**			
	100	**200**	**300**	**400**
1 core	14.87 (100%)	8.19 (48%)	8.19 (48%)	8.19 (48%)
2 cores	14.88 (100%)	14.88 (100%)	14.85 (200%)	11.39 (120%)
3 cores	14.88 (0%)	14.88 (0%)	14.88 (0%)	14.88 (0%)

Table 4.7: IRLsocket receive performance on the Intel processor (CPU utilization percent of one core).

5. CONCLUSIONS

We proposed a set of new algorithms for improving Windows networking and demonstrated that IRLstack 3.0 could achieve significantly lower CPU utilization and higher pps throughput compared to the fastest Linux variants, especially while utilizing the Intel i7 processor. We provided a socket-like library IRLsocket to users which scales well in multi-core system. Future work involves implementing a complete TCP stack to support a wider range of applications, and comparing its performance with existing work [6, 9] built on top of Packet I/O Engine and Netmap respectively.

REFERENCES

[1] Z. Durumeric, E. Wustrow and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," *in Proc. USENIX Security*, Aug. 2013, pp. 605–620.

[2] S. Han, K. Jang, K. Park and S. Moon, "PacketShader: A GPU-Accelerated Software Router," *in Proc. ACM SIGCOMM*, Aug. 2010, pp. 195–206.

[3] Intel, "Intel 82598 10 GbE Controller: Datasheet," `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82598-10-gbe-controller-datasheet.pdf`, accessed Oct. 5, 2014.

[4] Intel, "Intel 82599 10 GbE Controller: Datasheet," `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`, accessed Oct. 5, 2014.

[5] Intel, "Intel X540 Controller: Datasheet," `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-x540-datasheet.pdf`, accessed Oct. 5, 2014.

[6] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," *in Proc. USENIX NSDI*, Apr. 2014, pp. 489–502.

[7] H.-T. Lee, D. Leonard, X. Wang and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," *in Proc. WWW*, Apr. 2008, pp. 427–436.

[8] D. Leonard and D. Loguinov, "Demystifying Service Discovery: Implementing an Internet-Wide Scanner," *in Proc. ACM IMC*, Nov. 2010, pp. 109–122.

[9] I. Marinos, R. N. M. Watson and M. Handley, "Network Stack Specialization for Performance," *in Proc. ACM SIGCOMM*, Apr. 2014, pp. 175–186.

[10] Ntop, "PF_RING," `http://www.ntop.org/products/pf_ring/`, accessed Sep. 3, 2014.

[11] Ntop, "PF_RING DNA," `http://www.ntop.org/products/pf_ring/dna/`, accessed Sep. 3, 2014.

[12] Ntop, "PF_RING DNA License," `https://svn.ntop.org/svn/ntop/trunk/PF_RING/drivers/DNA/README.LICENSE`, accessed Sep. 3, 2014.

[13] L. Qiu, G. Varghese and S. Suri, "Fast Firewall Implementations for Software and Hardware-based Routers," *in Proc. IEEE ICNP*, Nov. 2001, pp. 241–250.

[14] P. M. S. del Rio, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli and J. Aracil, "Wire-Speed Statistical Classification of Network Traffic on Commodity Hardware," *in Proc. ACM IMC*, Nov. 2012, pp. 65–71.

[15] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," *in Proc. USENIX ATC*, Jun. 2012, pp. 101–112.

[16] Z. Shamsi, A. Nandwani, D. Leonard and D. Loguinov, "Hershel: Single-Packet OS Fingerprinting," *in Proc. ACM SIGMETRICS*, Jun. 2014, pp. 195–206.

[17] M. Smith and D. Loguinov, "Enabling High-Performance Internet-Wide Measurements on Windows," *in Proc. PAM*, Apr. 2010, pp. 121–130.

[18] P. J. Webster, "Towards More Efficient Delay Measurements on the Internet," Master's thesis, Texas A&M University, Aug. 2013.

[19] WinPcap, "The Windows Packet Capture Library," `http://www.winpcap.org/`, accessed May 11, 2014.