WireCAP: a Novel Packet Capture Engine for Commodity NICs in High-speed Networks

Wenji Wu, Phil DeMar
Fermilab, USA
wenji@fnal.gov; demar@fnal.gov

Abstract

Packet capture is an essential function for many network applications. However, packet drop is a major problem with packet capture in high-speed networks.

This paper presents WireCAP, a novel packet capture engine for commodity network interface cards (NICs) in high-speed networks. WireCAP provides *lossless zero-copy packet capture and delivery services* by exploiting multi-queue NICs and multicore architectures. WireCAP introduces two new mechanisms—*the ring-buffer-pool mechanism* and *the buddy-group-based offloading mechanism*—to address the packet drop problem of packet capture in high-speed network. WireCAP is efficient. It also facilitates the design and operation of a user-space packet-processing application. Experiments have demonstrated that WireCAP achieves better packet capture performance when compared to existing packet capture engines.

In addition, WireCAP implements a packet transmit function that allows captured packets to be forwarded, potentially after the packets are modified or inspected in flight. Therefore, WireCAP can be used to support middlebox-type applications. Thus, at a high level, WireCAP provides a new packet I/O framework for commodity NICs in high-speed networks.

1 Introduction

Packet capture is an essential function for many network applications, including intrusion detection systems (IDS) [1, 2, 3], and packet-based network performance analysis applications [4]. Packets are typically captured from the wire, temporarily stored at a *data capture buffer*, and finally delivered to the applications for processing. Because these operations are performed on a per-packet basis, packet capture is typically computationally and I/O throughput intensive. In high-speed networks (10 Gbps and above), packet capture faces significant performance challenges.

Packet drop is a major problem with packet capture in high-speed networks. There are two types of packet drop: *packet capture drop* and *packet delivery drop*. Packet capture drop is mainly caused by the inabilities of packet capture to keep pace with the incoming packet rate. Consequently, packets may be dropped because they cannot be captured in time. Packet delivery drop is mainly caused by the inability of the application to keep pace with the packet capture rate. Consequently, the data capture buffer overflows, and packet drops occur- even when 100% of the network traffic is captured from the wire. Any type of packet drop will degrade the accuracy and integrity of network monitoring applications [2, 5]. Thus, avoiding packet drops is a fundamental design goal in packet capture tools.

There are two approaches to performing packet capture. The first approach is to use a dedicated packet capture card to perform the function in hardware [6, 7]. This approach requires the least amount of CPU intervention, thus saving the CPU for packet processing. A dedicated packet capture card can ensure that 100% of the network packets are captured and delivered to applications without loss. However, this approach demands custom hardware solutions, which tend to be more costly, relatively inflexible, and not very scalable.

An alternative approach is to use a commodity system with a commodity NIC to perform packet capture. In this approach, the commodity NIC is put into promiscuous mode to intercept network packets. A packet capture engine (a software driver) receives the intercepted packets and provides support to allow user-space applications to access the captured packets. This capture solution depends mainly on the software-based packet capture engine, which is flexible and cost-effective, but requires significant system CPU and memory resources. Therefore, this solution is not suitable for resource-limited systems where resource competition between packet capture and packet processing might lead to drops. However, with recent technological advances in multicore platforms and multi-queue NICs, this approach becomes more appealing due to the availability of ample system CPU resources and I/O throughputs, and paves the way for a new paradigm in packet capturing and processing [8, 9, 10].

The new paradigm typically works as follows. A multi-queue NIC is logically partitioned into n receive queues, with each queue tied to a distinct core of a multicore system (Figure 1). Packets are distributed across the queues using a hardware-based traffic-steering mechanism, such as receive-side scaling (RSS) [11]. A thread (or process) of a packet-processing application runs on each core that has a tied queue. Each thread captures packets via a packet capture engine and handles a portion of the overall traffic. On a multicore system, there are several programming models (e.g., the run-to-completion model and the pipeline model [12]) for a packet-processing application. Here, the application may be of any type. This new paradigm essentially exploits the computing parallelism of multicore systems and the inherent data parallelism of network traffic to accelerate packet capturing and processing. A basic assump-

Operated by Fermi Research Alliance, LLC under Contract No. De-AC02-07CH11359 with the United States Department of Energy.

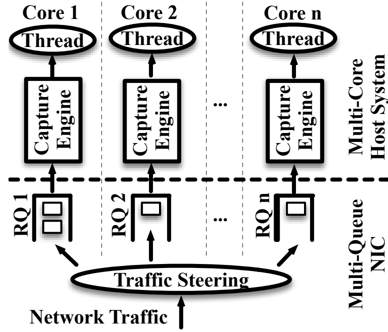

Figure 1 A new packet capturing and processing paradigm

tion associated with this approach is that the hardware-based balancing mechanism is capable of evenly distributing the incoming traffic among cores. Thus, each core would handle *1/n* of the overall traffic and the packet rate at each core would be reduced to *1/n* of the overall packet rate from the network, with a significantly reduced chance of causing a packet drop. However, this assumption is often not the case in practice [8]. Typically, a NIC's traffic-steering mechanism distributes packets to cores based on a per-flow policy that assigns packets of the same flow to the same core. A flow is defined by one or more fields of the IP 5-tuple. Such a traffic-steering mechanism maintains core affinity in network processing [11], helping to preserve application logic (i.e., packets belonging to the same flow must be delivered to the same application). However, this method of traffic steering can lead to a load imbalance condition in which certain cores become overloaded while others remain idle. In the worst-case scenario, a single core will be flooded with all the network traffic at wire speed.

There are two types of load imbalance. The first type is a short-term load imbalance on one or several cores. In this situation, an overloaded core experiences bursts of packets on a short scale. Here, "short" typically refers to time intervals up to 100 – 500 ms [13]. The second type is a long-term load imbalance, which may be due to an uneven distribution of flow groups in the NIC. Our research reveals that (1) load imbalance of either type occurs frequently on multicore systems; and (2) existing packet capture engines (e.g., PF_RING [14], NETMAP [15], and DNA [16]) can suffer significant packet drops when they experience load imbalance of either type in a multicore system, due to one or several of the following limitations: inability to capture packets at wire speed, limited buffering capability, and lacking of an effective offloading mechanism to address long-term long imbalance.

In order to avoid packet drops, load imbalances on multicore systems must be handled properly. There are several approaches for solving this problem. A first approach is to apply a round-robin traffic steering mechanism at the NIC level to distribute the traffic evenly across the queues. However, this approach cannot preserve the application logic (see Section 2.3). A second approach is to use existing packet capture engines (e.g., DNA) and handle load imbalance in the application layer. But an application in user space has little knowledge of low-level layer conditions, and cannot effectively handle load imbalance (see Section 2.3). A third approach is to design a new packet-capture engine that addresses load imbalance at the packet-capture level. A packet-capture engine has full knowledge of low-level layer conditions, placing it in a better position to deal with load imbalance. In addition, this approach simplifies the design of a packet-processing application. In this paper, we focus on the third approach.

This paper presents WireCAP, a novel packet capture engine for commodity NICs in high-speed networks. WireCAP is designed to support the packet capturing and processing model shown in Figure 1. It has several salient features.

(1) WireCAP provides lossless packet capture and delivery services by exploiting multi-queue NICs and multi-core architectures. We have designed two new mechanisms to handle load imbalance in order to avoid packet drops.

For short-term load imbalance, we design and implement a *ring-buffer-pool* mechanism. A ring buffer pool, which consists of chunks of packet buffers, is allocated for each receive queue in kernel. Through *dynamic packet buffer management*, each chunk of packet buffers can be used to receive packets flowing through the network, and temporarily store received packets. A ring buffer pool's capacity is configurable. When a large pool capacity is configured, a ring buffer pool can provide sufficient buffering at the NIC's receive ring level to accommodate short-term bursts of packets.

To handle long-term load imbalance, we apply a *buddy-group-based offloading* mechanism. The basic idea is simple: a busy packet capture engine offloads some of its traffic to less busy or idle queues (cores) where the traffic can be processed by other threads. However, the challenge of our design is how to preserve application logic—traffic belonging to the same flow must be delivered to the same application when multiple applications are running in the system. We introduce a *buddy group* concept: the receive queues accessed by threads (or processes) of a single application can form a *buddy group*. Traffic offloading is only allowed within a *buddy group*.

(2) WireCAP is efficient. WireCAP employs several optimization techniques—including pre-allocated large packet buffers, packet-level batching processing, and zero-copy—to reduce the packet capture and delivery costs. These optimization techniques have been used in the past and are well understood [14—16, 23]. The challenge in designing WireCAP was to understand how to combine these techniques with our *ring-buffer-pool* and *buddy-group-based offloading* mechanisms in an effective solution to achieve high performance.

In high-speed networks, excessive data copying results in poor performance [15, 17, 18, 19]. Zero-copy is widely used in our design. WireCAP can achieve *zero-copy packet capture and delivery* (i.e., a network packet can be captured and delivered to an application with zero-copy), and *zero-copy forwarding (*i.e., a captured packet can be forwarded with zero-copy).

We specifically separate short-term load imbalance from long-term load imbalance and address them appropriately. When only short-term load imbalance occurs, WireCAP not only avoids packet drops but also ensures that packets belonging to the same flow are handled in the same core. This design improves performance.

(3) With WireCAP, the design of a packet-processing application can be simplified. WireCAP addresses load imbalance in the packet-capture engine level to avoid packet drops. Therefore, an application need not implement its own mechanism to handle load imbalance in the application layer.

(4) With WireCAP, the performance of a packet processing application can be improved. WireCAP provides large ring buffer pools in the kernel to accommodate captured packets and supports *zero-copy packet capture and delivery*. Therefore, an application need not copy captured packets from low-level ring buffer pools and store them into its own set of buffers in user space. Instead, the application can use ring buffer pools as its own data buffers, and process the captured packets directly from there. This strategy helps to improve application performance.

(5) WireCAP implements a packet transmit function that allows captured packets to be forwarded, potentially after being analyzed in flight. Thus, WireCAP can be used to support middlebox-type applications.

Ultimately, WireCAP is intended to advance the state-of-the-art for packet capture engines on commodity multicore systems. Our design is unique in the sense that we seek to *address off-the-wire packet capture concerns in conjunction with packet delivery issues to the application*. The innovation in our design is twofold. First, we develop a new NIC ring-buffer-pool management mechanism that dramatically increases packet ingest buffering capabilities. Second, we introduce a load imbalance management mechanism that optimizes captured packet processing for multicore systems. Together, these innovations potentially pave the way for advancement of packet analysis tools to deal with emerging high-speed networks.

We implemented WireCAP in Linux. It consists of a kernel-mode driver and a user-mode library. The optimized kernel-mode driver manages commodity NICs and provides WireCAP's low-level packet capture and transmit services. The user-mode library is Libpcap-compatible [20], which provides a standard interface for low-level network access and allows existing network monitoring applications to use WireCAP without changes. We evaluated WireCAP with several experiments. Experiments have demonstrated that WireCAP achieves better performance compared with existing packet captures engines.

2 Background and Motivation

2.1 Problems with existing packet capture engines

When a commodity system with a commodity NIC is used for packet capture, the NIC is put into promiscuous mode to capture network packets. A packet capture engine manages the NIC and provides packet capture services to user-space applications.

Capturing packets from the wire. Packet capture is a special case of packet reception [21]. Typically, a commodity NIC can be logically partitioned into one or multiple receive queues. For each receive queue, the NIC maintains a ring of receive descriptors, called a receive ring. The number of receive descriptors in a ring is device-dependent. For example, an Intel 82599-based 10 GigE NIC has 8192 receive descriptors [22]. Assuming that the NIC is configured with n receive queues, each receive queue will have, at most, a ring size of $\lfloor 8192/n \rfloor$. A receive descriptor must be initialized and pre-allocated with an empty ring buffer in host memory—in the ready state—to receive a packet. Before packet capture begins, the packet capture engine performs this receive ring initialization. When network packets arrive, the NIC moves packets from the wire to the ring buffers via direct memory access (DMA). Subsequently, applications can access the received packets through the OS services provided by the packet capture engine. Receive descriptors are used circularly and repeatedly. The packet capture engine needs to reinitialize a used receive descriptor and refill it with an empty ring buffer as quickly as possible because incoming packets will be dropped if the receive descriptors in the ready state aren't available.

Existing packet capture engines. The protocol stack of a general purpose OS can provide standard packet capture services through raw sockets (e.g., PF_PACKET). However, because the protocol stack is designed to support a wide range of network applications, it is not optimized for packet capture. Consequently, the performance is inadequate for packet capture in high-speed networks. A number of packet capture engines, such as PF_RING [14], NETMAP [15], and DNA [16], have been developed. These packet capture engines essentially bypass the standard protocol stack and achieve improved performance through techniques such as pre-allocated large packet buffer, packet-level batch processing, and zero-copy [15, 23]. Existing packet capture engines can be classified into two types, depending on how captured packets are delivered from ring buffers to applications:

Type-I Packet Capture Engine, represented by PF_RING, pre-allocates a large packet buffer in kernel space for each receive ring. The large packet buffer consists of fixed-size cells. Each cell corresponds to a ring buffer and is assigned to a receive descriptor in the ring. A receive descriptor and its assigned ring buffer are 1-to-1 mapped to one another. The ring buffers are used circularly and repeatedly; a used receive descriptor is refilled with the same assigned ring buffer. In addition, PF_RING allocates an intermediate data buffer, termed *pf_ring*, within the kernel and uses it as *data capture buffer* for each receive ring. When the network packets arrive, the NIC moves the packets from the wire to the ring buffers via DMA. Subsequently, the packet capture engine copies packets from the ring buffers to *pf_ring* (for example, using NAPI polling in Linux [21]). Finally, a user-space application accesses captured packets from *pf_ring*. To improve performance, *pf_ring* is memory-mapped into the address space of the user-space application to avoid copying.

Type-II Packet Capture Engine, represented by DNA and NETMAP. Like PF_RING, DNA and NETMAP implement a similar pre-allocated large packet buffer scheme for ring buffer operations. DNA and NETMAP expose shadow copies of receive rings to user-space applications. The ring buffers of a receive ring are memory-mapped into the address space of a user-space application. The ring buffers not only are used to receive packets but are also employed as *data capture buffer* to temporarily store the packets. When network packets arrive, the NIC moves packets from the wire to the ring buffers via DMA. A user-space application accesses packets directly from the memory-mapped ring buffers. The advantage of this design is that it avoids the costs of unnecessary data movement.

The problems. A Type-I packet capture engine requires at least one copy to move a packet from the NIC ring into the user space. At high packet rates, excessive data copying results in poor performance. In addition, it may suffer the receive livelock problem in user context [24].

A Type-II packet capture engine has limited buffering capability. For DNA or NETMAP, a received packet is kept in a NIC ring buffer until it is consumed. During this period, the ring buffer and its associated receive descriptor cannot be released and reinitialized. Because a NIC ring has a limited number of receive descriptors, the receive descriptors in the ready state can be rapidly depleted if the received packets are not consumed in a timely manner. Consequently, subsequent packets would be dropped.

Neither type of packet capture engine has an effective offloading mechanism to address the long-term load imbalance problem.

2.2 Experimental proof

We conduct two experiments to support our claims. In the first, we demonstrate that load imbalance occurs frequently in multicore systems. In the second, we demonstrate the deficiencies of existing packet capture engines, including PF_RING, DNA, and NETMAP.

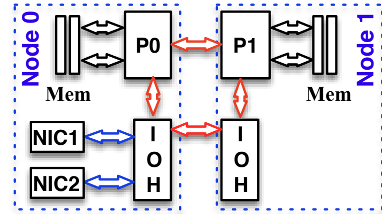

Figure 2 Experiment system

Experiment tools. The first tool is called *queue_profiler*. It is a single-threaded application that captures packets from a specific receive queue and counts the number of packets captured every 10 *ms*.

The second tool is called *pkt_handler* and is single-threaded. It captures and processes packets from a specific queue and executes a repeating *while* loop. In each loop, a packet is captured and applied with a Berkeley Packet Filter (BPF) [25] *x* times before being discarded. By varying *x*, we simulate different packet-processing rates of real applications during monitoring. In our experiments, the BPF filter "131.225.2 and UDP" is used, and *x* is set to 0 and 300, respectively. With x=0, no packet-processing load is actually applied. We use this value to evaluate whether a packet capture engine can capture packets at wire speed with no loss. With x=300, the packet-processing rate of a single 2.4 GHz CPU is 38,844 p/s. The value 300 was selected to emulate a heavy load application such as snort [1], which is capable of sustaining similar packet-processing rates [26].

Experiment configuration. The experiment system is depicted in Figure 2. It consists of two Intel E5-2690 processors and two Intel 82599-based 10 GigE NICs—*NIC1* and *NIC2*. The system I/O bus is PCIE-Gen3. A traffic generator capable of generating traffic at wire speed or replaying captured traffic at the speed exactly as recorded is connected directly to *NIC1*. *NIC2* is not used in this experiment.

The system runs a 64-bit Scientific Linux 6.0 server distribution. For packet capture engines, we use driver *pf_ring_5.5.2* (PF_RING) [14], *ixgbe-3.10.16-DNA* (DNA) [16], and *20131019-netmap* (NETMAP) [15].

Experiment data. To ensure that the evaluations are practical and repeatable, we capture traffic from the Fermilab border router for use as experiment data. The experiment data includes 5 million packets and lasts for approximately 32 seconds.

Experiment 1. The traffic generator replays the captured data at the speed exactly as recorded. *NIC1* is configured with six receive queues (*queue 0 – 5*), with each queue tied

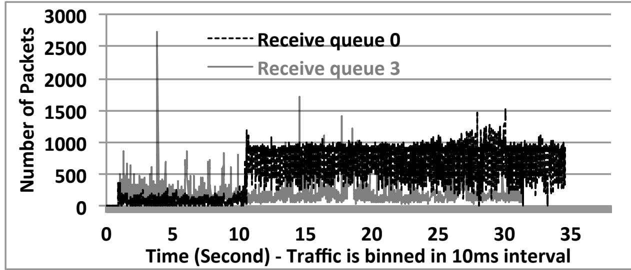

Figure 3 Load imbalance,

to a distinct core (*core 0 – 5*). A separate *queue_profiler* is launched to profile traffic for each queue. We use DNA as the packet capture engine. No packet drops occur in the experiment. Figure 3 presents the time-series of the number of packets received during a short time interval for queue 0 and 3, respectively. The figure shows that a load imbalance can occur routinely in a multicore system because the Intel NIC distributes packets to cores based on a per-flow policy that assigns packets of the same flow to the same core [22]. We observed both short-term load imbalance (short-term bursts of packets) and long-term load imbalance (*queue 0* receives much more traffic than *queue* 3). Because network traffic is bursty by nature and TCP is the dominant transport protocol, a short-term load imbalance occurs more frequently in practice. When we vary the number of receive queues, similar load imbalance patterns are observed.

Experiment 2. The traffic generator replays the same captured data as in experiment 1. Again, *NIC1* is configured with six receive queues (*queue 0 – 5*), and each queue tied to a distinct core (*core 0 – 5*) on which a separate *pkt_handler* is launched to capture and process the traffic for each queue. For *pkt_handler*, *x* is set to 0 and 300, respectively. The CPU speed is set at 2.4 GHz. We vary the packet capture engines in the experiments, using PF_RING (mode 2), DNA, and NETMAP, respectively. Each NIC receive ring is configured with a size of 1,024. For PF_RING, the size of *pf_ring* buffer is set to 10,240. We calculate the *packet capture drop rate* and the *packet delivery drop rate*. Note: because PF_PACKET's performance is too poor compared with these packet capture engines, we do not include PF_PACKET in our experiments. Readers can refer to [9] for PF_PACKET evaluation.

With x=0, each *pkt_handler* does not incur any packet-processing load. In the experiments, we did not observe any packet drops for DNA or NETMAP. However, we observed that PF-RING suffers small packet capture drops at queue 3.

With x=300, pkt_handler emulates a heavy load application. Table 1 lists the packet drop rates for queue 0 and 3, respectively. At queue 0, the incoming packet rate sustains approximately 80,000 p/s from 10s to 35s (Figure 3, Note:

	NETMAP	DNA	PF_RING
Receive Queue 0:			
Packet Capture Drops	46.5%	50.1%	0%
Packet Delivery Drops	0%	0%	56.8%
Receive Queue 3:			
Packet Capture Drops	33.4%	9.3%	0.8%
Packet Delivery Drops	0%	0%	0%

Table 1 Packet Drop Rates

the traffic is binned in 10ms interval), which far exceeds the packet-processing speed of a 2.4 GHz CPU (38,844 p/s). Because the *pkt_handler* at queue 0 is over-flooded by incoming packets, all packet capture engines suffer substantial packet drops. DNA and NETMAP suffer substantial packet capture drops because they use ring buffers as *data capture buffers*, which has a limited buffering capability. PF_RING avoids packet capture drops but suffers substantial packet delivery drops due to both the inability of the application to keep pace with the packet capture rate and the receive livelock problem [24]. In addition, because PF_RING requires at least one copy to move a packet from the NIC ring into the user space, PF_RING incurs higher packet capture costs than DNA and NETMAP. Consequently, PF_RING suffers higher overall packet drops than DNA and NETMAP. Table 1 demonstrates that the existing packet capture engines lack an offloading capability to migrate traffic to less busy or idle cores, where other thread would process it.

At queue 3, the incoming packet rate sustains approximately 20,000 p/s between 1s and 32s (Figure 3). While this rate is less than the packet processing speed of a 2.4 GHz CPU (38,844 p/s), DNA and NETMAP still suffer significant packet capture drops due to the limited buffing capability for dealing with short-term bursts of packets. For example, during the time interval [3.86s 3.97s], 2724 packets are sent to queue 3. Because *pkt_handler* can process only 10ms * 38,844 p/s = 388 packets during the interval and because the ring buffers can temporarily buffer 1024 packets at most, packet capture drops inevitably occur. PF-RING also suffers small packet capture drops.

2.3 How to avoid packet drops that are caused by load imbalance?

The experiments reveal that existing packet-capture engines suffer significant packet drops when they experience load imbalance in a multicore system. To avoid packet drops, we must handle load imbalance. There are several possible approaches.

A first approach is to apply a round-robin traffic steering mechanism at the NIC level to distribute the traffic evenly across the queues. However, this approach cannot preserve the application logic because packets belonging to

the same flow can be delivered to different applications when multiple applications are running in the system.

A second approach is to use existing packet capture engines (e.g., DNA) and to address load imbalance in the application layer. For example, an application thread can read packets from a queue, and store them into its own set of buffers, and then process a fixed number of packets before reading again. When load imbalance occurs, the application thread can move packets in some flows over to other threads if its own buffers were filling up. However, this approach has several limitations:

- An application in user space has little knowledge of low-level layer conditions. If the application thread cannot read data from a queue in time, bursts of network packets would overrun low-level packet buffers and cause packet drops.
- Because existing packet capture engines have a limited buffering capability, this approach must involve copying captured packets into user space. At high packet rates, excessive data copying leads to poor performance [15, 17, 18, 19].
- This approach would make the application complex and difficult to design.

Our approach is to design a new packet-capture engine that addresses load balance in the packet-capture level. We believe a packet capture engine is in a better position to address load imbalance because it has full knowledge of low-level layer conditions. WireCAP is our solution.

3 The WireCAP Design & Implementation

3.1 Design goals

WireCAP is designed to support the packet capturing and processing paradigm as shown in Figure 1. We have several design goals:

- Providing lossless packet-capture services in high-speed networks is our primary goal. WireCAP aims to provide lossless packet capture services in high-speed networks by exploiting multicore architecture and multi-queue NICs. Therefore, WireCAP must handle load imbalance that frequently occurs in multicore systems.
- Providing efficient packet delivery.
- WireCAP must be efficient. In a multicore system, core affinity on network processing can significantly improve performance. Therefore, when short-term load imbalance occurs, WireCAP must not only avoid packet drops but also ensure core affinity on network processing. However, when long-term load imbalance occurs, WireCAP has to perform traffic offloading to avoid packet drops. We believe this is an appropriate tradeoff.
- WireCAP must facilitate the design and operation of a packet-processing application in user space.
- WireCAP must have wide applicability and can be easily adopted.
- WireCAP should implement a transmit function that allows captured packets to be forwarded. Such a function would allow WireCAP to support middlebox-type applications.

3.2 WireCAP design

We begin by describing our *ring-buffer-pool* and *buddy-group-based offloading* mechanisms. Then, we discuss the WireCAP architecture.

3.2.1 The mechanisms

Ring-buffer-pool. Assume each receive queue has a ring of N descriptors. Under WireCAP, each receive ring is divided into *descriptor segments*. A descriptor segment consists of M receive packet descriptors (e.g., 1024), where M is a divisor of N. In kernel space, each receive ring is allocated with R *packet buffer chunks*, termed the *ring buffer pool*. In this case, R is greater than N/M. A *packet buffer chunk* consists of M fixed-size cells, with each cell corresponding to a ring buffer. Both M and R are configurable. Within a pool, a packet buffer chunk is identified by a unique *chunk_id*. Globally, a packet buffer chunk is uniquely identified by a {*nic_id*, *ring_id*, *chunk_id*} tuple. Here, *nic_id* and *ring_id* refer to the NIC and to the receive ring that the packet buffer chunk belongs to.

When an application *opens* a receive queue to capture packets, the ring buffer pool for the receive queue will be mapped into the application's process space. Therefore, a packet buffer chunk has three addresses, *DMA_address*, *kernel_address*, and *process_address*, which are used by the NIC, the kernel, and the application, respectively. These addresses are maintained and translated by the kernel. A cell within a chunk is accessed by its relative address within the chunk.

A packet buffer chunk can exist in one of three states: "free", "attached", and "captured". A "free" chunk is maintained in the kernel, available for (re)use. In the "attached" state, the chunk is attached to a descriptor segment in its receive ring to receive packets. Each cell in the chunk is sequentially tied to the corresponding packet descriptor in the descriptor segment. A "captured" chunk is one that has been filled with received packets and captured into the user space.

A ring-buffer-pool provides operations to allow a user-space application to capture packets. These operations can be accessed through the *ioctl* interface:

Open. Opens a specific receive queue for packet capture. It maps its ring buffer pool into the application's pro-

cess space and attaches each descriptor segment in the receive ring with a "free" packet buffer chunk.

Capture. Captures packets in a specific receive queue. The capture operation is performed in the units of the packet buffer chunk; a single operation can move multiple chunks to the user space. To capture a packet buffer chunk to user space, only its metadata {{*nic_id*, *ring_id*, *chunk_id*}, *process_address*, *pkt_count*} is passed. The chunk itself is not copied. Here, *pkt_count* counts the number of packets in the chunks. When a packet buffer chunk attached to the receive ring is captured to the user space, the corresponding descriptor segment must be attached with a new "free" chunk to receive subsequent packets. Because the NIC moves incoming packets to the empty ring buffers without CPU intervention, a packet buffer chunk cannot be safely moved unless it is *full*. Otherwise, packet drops might occur. Thus, our capture operation works as follows. (1) If no packet is available, the capture operation will be blocked until incoming packets wake it up. (2) Else if full packet buffer chunks are available, the capture operation will return immediately, with one or multiple full chunks moved to the user space. The corresponding descriptor segment will be attached with a new "free" chunk. (3) Else, the capture operation will be blocked with a timeout. The process will continue as stated in (2) if new full packet buffer chunks become available before the timeout expires. If the timeout expires and the incoming packets only partially fill an attached packet buffer chunk, we copy them to a "free" packet buffer chunk, which is moved to the user space instead. This mechanism avoids holding packets in the receive ring for too long.

Recycle. In the user space, once the data in a "*captured*" packet buffer chunk are finally processed, the chunk will be recycled for future use. To recycle a chunk, its metadata are passed to the kernel, which will be strictly validated and verified; the kernel simply changes the chunk's state to "free".

Close. Closes a specific receive queue for packet capture and performs the necessary cleaning tasks.

Through the *capture* and *recycle* operations, each chunk of packet buffers can be used to receive packets flowing through the network, and temporarily store received packets. A ring buffer pool's capacity is configurable. When a large pool capacity is configured, a ring buffer pool can provide sufficient buffering at the NIC's receive ring level to accommodate short-term bursts of packets. Thus, it helps to avoid packet drops.

Buddy-group-based offloading. The basic concept is simple: a busy packet capture engine offloads some of its traffic to less busy or idle queues (cores) where it can be processed by other threads. However, the challenge is how to preserve application logic—traffic belonging to the same flow must be delivered to the same application when multiple applications are running in the system. Therefore, we introduce a *buddy group* concept: the receive queues accessed by a single application can form a *buddy group*. Traffic offloading is only allowed within a *buddy group*. We illustrate the buddy group concept in Figure 4. In the system, each receive queue (RQ1—RQ4) is tied to a distinct core. Application 1's threads are running at core 1 and 2 while application 2's threads are running at core 3 and 4. In this scenario, RQ1 and RQ2 can form a buddy group to implement the offloading mechanism for application 1. Similarly, RQ3 and RQ4 can form a buddy group to implement the offloading mechanism for application 2.

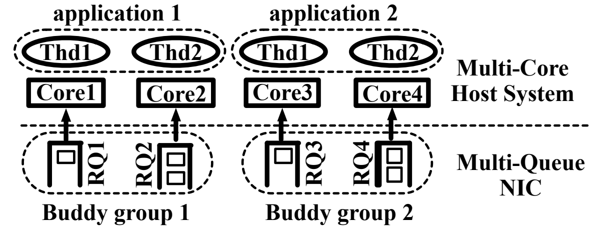

Figure 4 The buddy-group concept

3.2.2 The WireCAP architecture

Figure 5 shows the WireCAP architecture. It consists of a kernel-mode driver and a user-mode library.

- The kernel-mode driver manages commodity NICs and provides low-level packet capture and transmit services. It applies the *ring-buffer-pool* mechanism to handle short-term load imbalance.
- The user-mode library extends and builds upon the services provided by the kernel-mode driver and executes several mechanisms: it provides a Libpcap-compatible interface for low-level network access, and it applies the *buddy-group-based offloading* mechanism to handle long-term load imbalance.

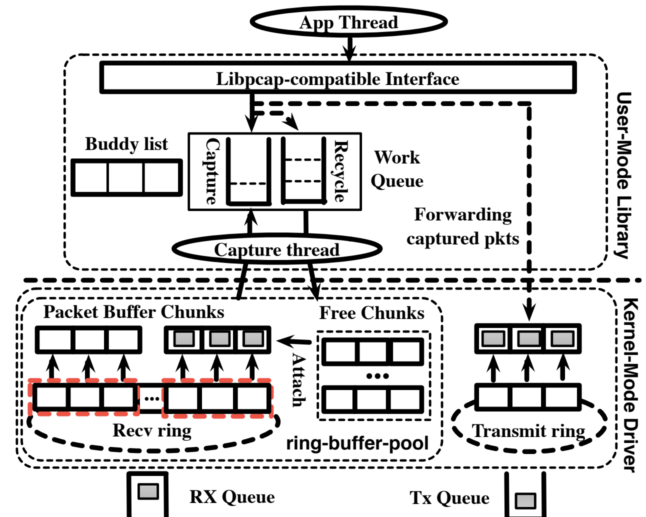

Figure 5 The WireCAP architecture

a. Lossless zero-copy packet capture and delivery

WireCAP captures packets on a per-receive-queue basis. When a user-space application opens a receive queue to capture packets, the kernel-mode driver maps the ring buffer pool associated with the receive queue into the application's process space, and the use-mode library creates and assigns three key entities in the user space for the receive queue:

- *A capture thread* performs the low-level *capture* and *recycle* operations, and implements the offloading mechanism. Typically, a *capture thread* and an application thread do not run in the same core. The system can dedicate one or several cores to run all *capture threads*.
- *A work-queue pair* consists of a capture queue and a recycle queue. A capture queue keeps the metadata of captured packet buffer chunks, and a recycle queue keeps the metadata of packet buffer chunks that are waiting to be recycled.
- *A buddy list* keeps the buddies of a receive queue in a buddy group. It is used to implement the *buddy-group-based offloading* mechanism. The receive queues in a buddy group are buddies. The user-mode library provides functions to allow an application to populate the buddies of a receive queue.

WireCAP captures packets in two modes—the basic mode and the advanced mode.

In the basic mode, WireCAP handles each receive queue independently. Figure 6a illustrates how this process works. (1) For each receive queue, its dedicated *capture thread* executes the low-level *capture* operations to move filled packet buffer chunks into the user space. The packet buffer chunks captured from a particular receive queue are placed into its capture queue in the user space. (2) To ingest packets from a particular receive queue, a packet-processing thread accesses the receive queue's capture queue in the user space through a Libpcap-compatible API such as *pcap_loop()* or *pcap_dispatch()*. Packet buffer chunks in the capture queue are processed one by one; a used packet buffer chunk is placed into the associated recycle queue. (3) A *capture thread* executes the low-level *recycle* operations to recycle used packet buffer chunks from its associated recycle queue.

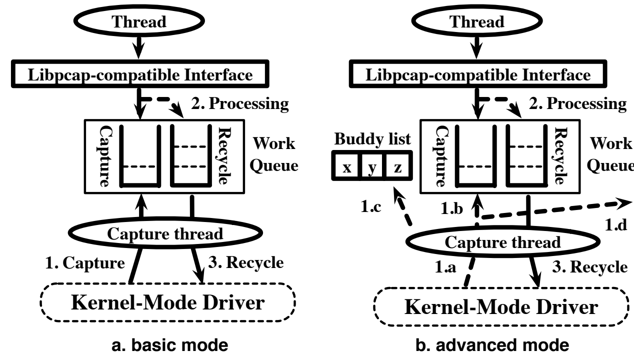

Figure 6 WireCAP packet-capturing operations

A ring buffer pool can temporarily store *R*M* packets. Assuming P_{in} is the maximum incoming packet burst rate at a receive queue and P_p is the processing rate of the corresponding packet-processing thread. WireCAP in the basic mode can handle a maximum burst of $P_{in}*(R*M)/(P_{in}-P_p)$ packets without loss at the receive queue. However, WireCAP in the basic mode cannot handle long-term load imbalance. When one core is over-flooded by incoming packets, the corresponding capture queue in the user space will soon be filled. Because the captured packet buffer chunks cannot be processed and recycled in time, the free packet buffer chunks in the corresponding ring buffer pool become depleted, causing packet capture drops.

In the advanced mode, WireCAP updates the basic mode operation (1) with the *buddy-group-based offloading* mechanism to handle long-term load imbalance. Figure 6b illustrates how this process works: (1.a) For each receive queue, its dedicated *capture thread* executes the low-level *capture* operations to move filled packet buffer chunks into the user space. (1.b) When a *capture thread* moves a chunk into the user space, the thread examines its associated capture queue in the user space. If the queue length does not exceed an *offloading percentage threshold* (*T*), WireCAP's indicator of long-term load imbalance, the thread will place the chunk into its own capture queue. (1.c) When the threshold *T* is exceeded, the thread will query the associated buddy queue list and (1.d) place the chunk into the capture queue of an idle or less busy receive queue. The assumption is that, when a capture queue is empty or shorter, the corresponding core is idle or less busy.

In our design, a ring buffer pool is mapped into an application's process space. Thus, a network packet can be captured and delivered to the application with zero-copy. In addition, WireCAP addresses load imbalance to avoid packet drops. Therefore, WireCAP can achieve *lossless zero-copy packet capture and delivery*.

b. Zero-copy packet forwarding

A multi-queue NIC can be configured with one or multiple transmit queues for outbound packets. For each transmit queue, the NIC maintains a ring of transmit descriptors, called a transmit ring. To transmit a packet from a transmit queue, the packet should be attached to a transmit descriptor in the transmit ring of the queue. The transmit descriptor helps the NIC locate the packet in the system. After that, the NIC transmits the packet.

With WireCAP, an application can use ring buffer pools as its own data buffers and handle captured packets directly from there. Therefore, the application can forward

a captured packet by simply attaching it to a specific transmit queue, potentially after the packet has been analyzed and/or modified. Attaching a packet to a transmit queue only involves metadata operations. The packet itself is not copied.

c. Safety considerations

Two aspects of WireCAP would raise safety concerns: (1) the sharing of memory between the kernel and the user-space applications and (2) the offloading mechanism. However, WireCAP should not cause any safety problems. A misbehaving application will not crash the kernel. WireCAP only maps ring-buffer pools, which do not involve critical kernel memory regions, into the address space of user-space applications. In addition, when a used packet buffer chunk is to be recycled, its metadata will be strictly validated and verified by the kernel. Similarly, a misbehaving application will not crash other applications. The offloading mechanism is implemented across the receive queues of a buddy group belonging to the same application. Different applications do not interfere with one another.

d. Supporting multiple NICs

Because WireCAP operates on a per-receive-queue basis, WireCAP naturally supports multiple NICs. In Section 4, we run experiments to demonstrate that WireCAP supports multiple NICs.

e. Application support

Upon work-queue pairs, WireCAP supports a Libpcap-compatible interface. Packets can be read with APIs such as *pcap_dispatch ()* or *pcap_loop ()*. Both blocking and non-blocking modes can be supported.

3.3 Implementation

WireCAP was developed on Linux. The kernel-mode driver was implemented on the Intel 82599-based 10GigE NIC. We modified the *ixgbe* driver to implement the WireCAP functions, and the modifications involve a few hundred lines of code. We also implemented a user-mode library. The current version supports a simple Libpcap-compatible interface. We plan to release WireCAP for public access soon.

4 Performance evaluations

We evaluate WireCAP using the experiment tools and configuration described in Section 2.2. In addition, we develop a third tool for our experiments. It is a multi-threaded version of *pkt_handler,* called *multi_pkt_handler,* which can spawn one or multiple *pkt_handler* threads that share the same address space.

A simple name convention is used in the following sections. *WireCAP-B-(M, R)* represents WireCAP in the basic mode with a descriptor segment size of *M* and a ring pool size of *R*, while *WireCAP-A-(M, R, T)* represents WireCAP in the advanced mode with a descriptor segment size of *M*, a ring pool size of *R*, and an offloading threshold of *T*.

We compare WireCAP with existing packet capture engines (PF_RING, DNA, and NETMAP). The performance metric is packet drop rate. In the experiments, these packet capture engines suffer different types of packet drops: (1) WireCAP suffers only packet capture drops, which can occur when the free packet buffer chunks in a ring buffer pool are depleted; (2) DNA and NETMAP suffer only packet capture drops; and (3) PF_RING suffers both packet capture drops and packet delivery drops. To make the comparison easier, we only calculate the overall packet drop rate. Each NIC receive ring is configured with a size of 1,024 for all packet capture engines. PR_RING is set to run at mode 2, and the size of *pf_ring* buffer is set to 10,240. The CPU frequency is set to 2.4 GHz.

Packet capture in the basic mode. The traffic generator transmits *P* 64-Byte packets at the wire rate (14.88 million p/s). *P* varies, ranging from 1,000 to 10,000,000. *NIC1* is configured with a single receive queue, tied to a core, on which a *pkt_handler* is launched to capture and process traffic for that queue. For *pkt_handler*, *x* is set to 0 and 300.

With x=0, *pkt_handler* does not incur any packet-processing load. We test WireCAP with various *R* and *M* values. No packet drops are observed (Figure 7), indicating that WireCAP can capture packets at wire speed without loss. No packet drops were observed for NETMAP and DNA. However, PF-RING suffers significant packet drops (both packet delivery drops and packet capture drops).

With x=300, *pkt_handler* emulates a heavy load application. Because the incoming packet rate (14.88 million p/s) far exceeds the packet-processing speed of *pkt_handler* on a 2.4 GHz CPU (38,844 p/s), the maximum *P* that a packet capture engine can handle without packet loss reflects its buffering capability for short-term bursts of packets. As shown in figure 8, WireCAP demonstrates superior buffering capability for short-term bursts of packets. For

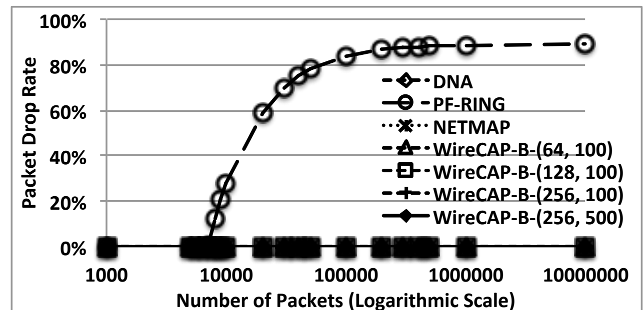

Figure 7 WireCAP packet capture in the basic mode, with no packet processing load (x=0)

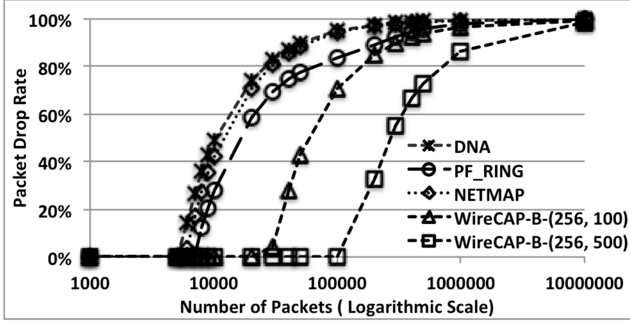

Figure 8 WireCAP packet capture in the basic mode, with a heavy packet-processing load (x=300)

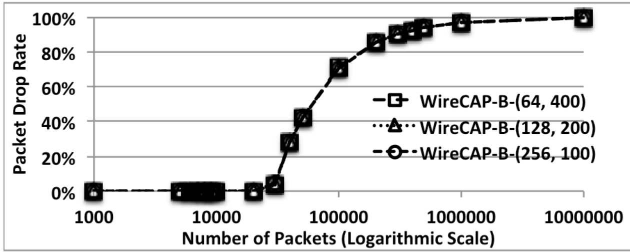

Figure 9 WireCAP packet capture in the basic mode (R and M are varied, R*M is fixed)

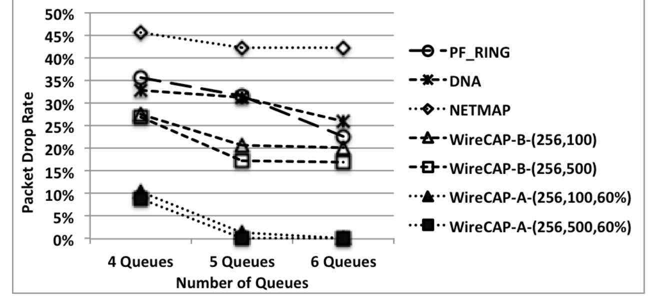

Figure 10 WireCAP packet capture in the advanced mode, with a heavy packet-processing load (x=300)

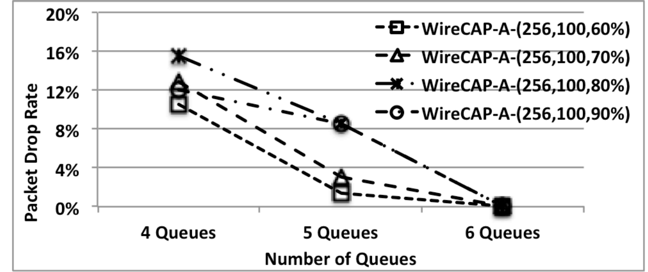

Figure 11 WireCAP packet capture in the advanced mode (R and M are fixed, T is varied)

example, DNA suffers a 15% packet drop at *P=6,000*, while *WireCAP-B-(256, 500)* has no packet drops even at *P=100,000*. WireCAP's resilient buffering capability comes from its unique ring-buffer-pool design and is proportional to the ring-buffer pool capacity *R*M*. *WireCAP-B-(256, 500)* clearly has a higher buffering capability than *WireCAP-B-(256,100)*. At *P=100,000*, *WireCAP-B-(256,500)* has no packet drops, whereas *WireCAP-B-(256, 100)* has a packet drop rate of 71%.

Figure 9 illustrates that WireCAP's buffering capability is proportional to the overall ring buffer capacity *R*M*. The individual *R* or *M* does not affect the overall performance. In the experiment, *R* and *M* are varied, but *R*M* is fixed. The results indicate that *WireCAP-B-(64, 400)*, *WireCAP-B-(128, 200)*, and *WireCAP-B-(256, 100)* have approximately the same packet drop rates at different *P* values.

Packet capture in the advanced mode. The traffic generator replays the captured data at the speed exactly as recorded. *NIC1* is configured with *n* receive queues, with each queue tied to a distinct core. A *multi_pkt_handler* runs at the system and spawns *n pkt_handler* threads. Each thread runs on a core that has a tied receive queue. It captures and processes traffic from its queue. For *pkt_handler*, *x* is set to 300. We vary the packet capture engines in the experiments, using PF_RING, DNA, NETMAP, and WireCAP in the basic mode and WireCAP in the advanced mode, respectively. For WireCAP in the advanced mode, the *n* queues form a single buddy group.

In the basic mode, WireCAP can capture packets at wire speed and effectively handle short-term bursts of packets. Although WireCAP in the basic mode achieves better performance than existing packet capture engines, it still suffers significant packet losses, due to long-term load imbalance (Figure 10). WireCAP in the advanced mode implements the *buddy-group-based offloading* mechanism to address that problem. This mechanism allows the system resources to be better utilized. It is evident that the offloading mechanism achieves a significantly improved performance (Figure 10). For WireCAP in the advanced mode, the offloading mechanism is triggered when the queue length of a capture queue exceeds the offloading percentage threshold (*T*). In general, WireCAP performs better when *T* is set to a relatively lower value (Figure 11).

Packet forwarding. We repeated the above experiments with a small modification to *pkt_handler:* a processed packet is forwarded through *NIC2* (Figure 2) instead of being discarded. *NIC2* is directly connected to a packet receiver. By counting the number of packets that the traffic generator sends and the number of packets the traffic receiver receives, we calculate the packet drop rate. Figure 12 illustrates the experiment results.

The experiments demonstrate that WireCAP's packet forwarding function is capable of supporting middlebox applications. Again, the experiments reveal that the *buddy-group-based offloading* mechanism can achieve a significant improved performance.

We cannot make *multi_pkt_handler* work under NETMAP in this experiment. Under NETMAP, a

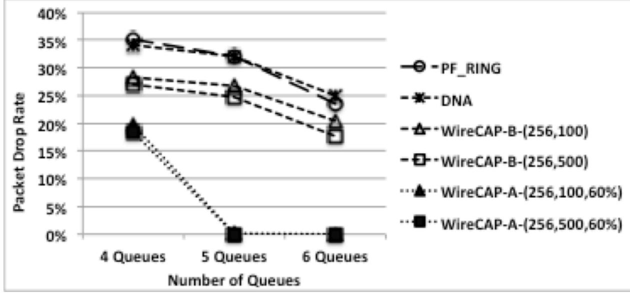

Figure 12 WireCAP packet forwarding

pkt_handler thread cannot synchronize between receiving and transmitting because NETMAP's NIOCTXSYNC (or NIOCRXSYNC) operations do not work on a per-receive-queue (or per-transmit-queue) basis.

WireCAP scalability. We now evaluate and discuss WireCAP's scalability performance. In the experiments, each of *NIC1* and *NIC2* is connected directly to a traffic generator. The generators transmit $1x10^9$ 64-Byte or 100-Byte packets at the wire rate. Each NIC is configured with *n* receive queues, with each queue tied to a distinct core.

- For *NIC1*, a *multi_pkt_handler* is launched, which spawns *n* pkt_handler threads. Each thread runs on a core that has a tied receive queue of *NIC1*. It captures and processes traffic from its queue. For pkt_handler, *x* is set to *0*, with processed packet forwarded on through *NIC2*.
- For *NIC2*, a *multi_pkt_handler* is similarly launched except that captured packets are forwarded on through *NIC1*.

In this experiment, we only compare DNA and WireCAP in the advanced mode because PF_RING's performance is too poor and we cannot make *multi_pkt_handler* work under NETMAP. With WireCAP in the advanced mode, *NIC1's* queues form a buddy group and *NIC2's* queues form another buddy group. Each of *NIC1* and *NIC2* is directly connected to a packet receiver. By counting the number of packets that the traffic generators send and the number of packets that the traffic receivers receive, we calculate the packet drop rate. Figure 13 illustrates the experiment results.

When the generators transmit 100-Byte packets, *NIC1* and *NIC2* in together receive approximately 20 million p/s. We did not observe any packet loss for WireCAP and DNA. The experiment indicates that WireCAP scales well with multiple NICs. Please note: *NIC1* and *NIC2* are installed in a single NUMA node on our experiment system (Figure 2)

When the generators transmit 64-Byte packets, the system needs to handle an approximate rate of 30 million p/s. Under such conditions, the experiment system bus becomes saturated, causing both DNA and WireCAP

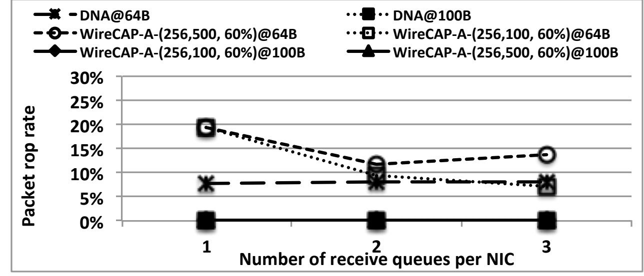

Figure 13 Scalability experiment

to suffer significant packet drops. Compared with DNA, WireCAP requires extra I/O operations and memory accesses to implement its *ring-buffer-pool* and *buddy-group-based offloading* mechanisms. When the system bus is saturated, the I/O operations and memory accesses become costly. As a consequence, WireCAP suffers a higher packet drop rate than DNA. However, WireCAP is designed to use additional system resources to address the packet drop problem. Certainly WireCAP of such a design will lose some scalability performance. We believe this is an appropriate tradeoff.

5 Related work

RSS and Flow Director [11, 22] are advanced NIC technologies that enable the efficient distribution of packets across receive queues in a multi-queue NIC. RSS uses a hash function in the NIC. The NIC computes a hash value for each incoming packet. Based on hash values, the NIC assigns packets of the same flow to a single queue. Flow Director maintains a flow table in the NIC to assign packets across queues. Each flow has an entry in the table. The entry tracks which queue a flow should be assigned to. The flow table is established and updated by traffic in both the forward and reverse directions. Flow Director is typically not used in a packet capture environment because the traffic is unidirectional.

The protocol stack of a general purpose OS can provide standard packet capture services through raw sockets (e.g., PF_PACKET). However, research [9] show that the performance is inadequate for packet capture in high-speed networks.

Several packet I/O engines have been proposed to boost the performance of commodity NICs in packet capture, such as NETMAP [15], DNA [16], PF_RING [14], and the PacketShader I/O engine (PSIOE) [23], These packet capture engines essentially bypass the standard protocol stack and achieve improved performance. Table 2 compares WireCAP and the existing packet-capture engines.

WireCAP	**Goal**: avoiding packet drops. **Deficiency**: requiring additional resources.
DNA **NETMAP**	**Goal**: minimizing packet capture costs. **Deficiency:** limited buffering capability, no offloading mechanism.
PSIOE	**Goal**: maximizing system throughput. **Deficiency**: limited buffering capability; copying in packet capture.
PF_RING	**Goal**: minimizing packet capture costs. **Deficiency:** copying in packet capture; receive livelock problem; no offloading mechanism.

Table 2 WireCAP vs. existing packet-capture engines

PSIOE is similar to PF_RING, except that PSIOE uses a user-space thread, instead of Linux NAPI polling, to copy packets from receive ring buffers to a consecutive user-level buffer (*user buffer*). For PacketShader, the copy operation in packet capture makes little impact on performance and does not consume additional memory bandwidth because the user buffer likely resides in CPU cache [23]. ***However, this result does not have wide applicability.*** If a network application that uses PSIOE to capture packets has a large working set, the *user buffer* is not likely to reside in CPU cache. Under such condition, copying will have a significant impact on performance. PSIOE was specifically designed and optimized for PacketShader. It provides only a limited buffering capability for the incoming packets. PSIOE is not suitable for a heavy-load application.

At a high level, WireCAP provides a new packet I/O framework for commodity NICs in high-speed networks. Therefore, there are some similarities between WireCAP and Intel DPDK [27]. Intel DPDK provides a set of libraries and drivers for fast packet processing on x86 platforms. However, Intel DPDK does not provide a complete solution to avoid packet drops in high-speed networks as WireCAP does.

6 Conclusion & discussion

In this paper, we have described our architectural approach in developing a novel packet capture engine for commodity NICs in high-speed networks. Experiments have demonstrated that WireCAP achieves better packet capture performance than that of the existing packet capture engines through the use of additional resources. WireCAP employs additional computing resources by dedicating a capture thread to perform low-level *ioctl* operations for each receive queue. A modern multicore system can provide sufficient computing resources to support WireCAP operations.

WireCAP utilizes large amounts of kernel space memory to support its ring-buffer-pool mechanism. For *WireCAP-B-(M, R)* or *WireCAP-A-(M, R, T)*, a single pool requires $R*M*2K$ bytes of memory (Note: a cell is two Kbytes in the current implementation). If n receive queues are configured, then $n*R*M*2K$ *bytes* are required. Because WireCAP's buffering capability is proportional to the ring buffer capacity $R*M$, there is a tradeoff between WireCAP's buffering capability and its memory consumption.

Determining an optimal value for T in WireCAP's advanced mode operation also presents tradeoff challenges. Offloading is necessary to prevent packet drops due to a long-term load imbalance. However, redirecting packets to different, less busy capture queues can result in a degraded CPU efficiency caused by a loss of the core affinity on packet processing [11]. Therefore, a simple guideline for configuring T is as follows*:* When avoiding packet drops is critical to the application, T should be set to a relatively lower value (e.g., 50%); otherwise, T should be set to a relatively higher value (e.g., 80%).

WireCAP uses batch processing to reduce packet capture costs. Applying this type of technique may entail side effects, such as latency increases and inaccurate timestamping [28].

Ideally, WireCAP is designed to support the packet capturing and processing paradigm as shown in Figure 1. Within a multi-queue NIC, packets are distributed across receive queues using a hardware-based traffic-steering mechanism. And each receive queue is handled by a thread (or process) of a packet processing application. Because this paradigm uses NIC hardware, instead of CPU, to classify and steer packets across cores, it helps to save CPU for packet capturing and processing. On the other hand, modern NICs are becoming more powerful, and typically feature advanced traffic filtering, classification, and steering mechanisms. We believe this packet capturing and processing paradigm is a promising approach.

However, WireCAP is flexible and robust enough to support other types of packet capturing and processing paradigms:

- Multiple threads (or processes) of a packet-processing application can access a single NIC receive queue, through the queue's corresponding work-queue pair in user space. Certainly, this approach incurs extra synchronization overheads across these threads.
- Upon WireCAP work-queue pairs, a packet-processing application can implement its own traffic steering and classification mechanisms to create packet queues at the application level, in the cases of the NIC hardware-based traffic classification and steering mechanism cannot meet the application requirements; or there are not enough physical queues in the NIC.

In these paradigms, a simple approach is to copy captured packets from WireCAP into the application's own set of

buffers. This approach simplifies WireCAP's *recycle* operations while the benefit of zero-copy delivery will not be available. However, WireCAP still provides lossless packet capture and delivery services.

WireCAP can be configured to switch between supporting different packet capturing and processing paradigms.

References

[1] M. Roesch, Snort: Lightweight Intrusion Detection

[2] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In *ACM ERUOSEC* (2010), pp. 15-21.

[3] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, Kargus: a highly-scalable software-based intrusion detection system. In *ACM CCS* (Oct. 2012), pp. 317-328.

[4] V. Paxson, Automated packet trace analysis of TCP implementations. *ACM SIGCOMM CCR* 27, 4 (1997).

[5] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security* (Aug. 2003), pp. 29-44.

[6] http://www.endace.com/endace-dag-high-speed-packet-capture-cards.html.

[7] http://www.napatech.com/.

[8] F. Fusco, and L. Deri, High speed network traffic analysis with commodity multi-core systems. In *ACM IMC* (2010), pp. 218-224.

[9] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, Comparing and improving current packet capturing solutions based on commodity hardware. In *ACM IMC* (2010), pp. 206-217.

[10] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, On multi–gigabit packet capturing with multi–core commodity hardware. In *PAM* (2012), pp. 64-73.

[11] W. Wu, P. DeMar, and M. Crawford, A transport-friendly NIC for multicore/multiprocessor systems. Parallel and Distributed Systems, IEEE Transactions on, 23(4), 607-615.

[12] C. F. Dcumitrescu, Models for packet processing on multi-core systems. http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/ia-multicore-packet-processing-paper.html.

[13] T. Benson, A. Akella, and D. A. Maltz, Network traffic characteristics of data centers in the wild. In *ACM IMC* (2010), pp. 267-280.

[14] PF_RING. www.ntop.org/products/pf_ring/.

[15] L. Rizzo, netmap: a novel framework for fast packet I/O. In *USENIX ATC* (2012).

[16] DNA. www.ntop.org/products/pf_ring/dna/.

[17] H. K. J. Chu, Zero-copy TCP in Solaris. In Proceedings of the 1996 annual conference on USENIX Annual Technical Conference (pp. 21-21). Usenix Association.

[18] P. Druschel, and G. Banga, Lazy receiver processing (LRP): A network subsystem architecture for server systems. In OSDI (Vol. 96, pp. 261-275).

[19] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier, TCP performance re-visited. In Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on (pp. 70-79). IEEE.

[20] TCPDUMP and LIBPCAP public repository. http://www.tcpdump.org/.

[21] W. Wu, M. Crawford, and M. Bowden, The performance analysis of Linux networking–packet receiving, Computer Communications, 30(5), 1044-1057.

[22] Intel 82599 10GbE Controller Datasheet. http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[23] S. Han, K. Jang, K. Park, and S. Moon, PacketShader: a GPU-accelerated software router. ACM SIGCOMM Computer Communication Review, 40(4), 195-206.

[24] J. Mogul, and K. K. Ramakrishnan, Eliminating receive livelock in an interrupt-driven kernel. In *ACM Transactions on Computer Systems* 15.3 (1997): 217-252.

[25] S. McCanne and V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture. In *Winter USENIX Conference*, 1993.

[26] J. S. White, T. Fitzsimmons, and J. N. Matthews, Quantitative analysis of intrusion detection systems: Snort and Suricata. In SPIE Defense, Security, and Sensing, pp. 875704-875704.

[27] Intel DPDK website. http://dpdk.org.

[28] V. Moreno, P. Santiago del Rio, J. Ramos, J. Garnica, and J. Garcia-Dorado, Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines, *IEEE Communications letters*, 16(11): 1888-1891, 2012.