Improving Server Application Performance via Pure TCP ACK Receive Optimization

Michael Chan
Department of Computer Science
Stanford University
mcfchan@stanford.edu

David R. Cheriton
Department of Computer Science
Stanford University
cheriton@cs.stanford.edu

Abstract

Network stack performance is critical to server scalability and user-perceived application experience. Per-packet overhead is a major bottleneck in scaling network I/O. While much effort is expended on reducing per-packet overhead for data-carrying packets, small control packets such as pure TCP ACKs have received relatively scarce attention. In this paper, we show that ACK receive processing can consume up to 20% cycles in server applications. We propose a simple kernel-level optimization which reduces this overhead through fewer memory allocations and a simplified code path. Using this technique, we demonstrate cycles savings of 15% in a Web application, and 33% throughput improvement in reliable multicast.

1 Introduction

Performance of the endhost networking stack is critical to server scalability and perceived user application experience. Increases in network link speeds raise concerns over CPU utilization in keeping up with wireline data rates. This has led to various techniques such as TCP segmentation offloading (TSO) and generic receive offloading (GRO), which reduce overhead for packets carrying application payload. In contrast, relatively little attention has been given to offloading processing of small control packets such as pure TCP ACKs. (We define a pure ACK as a TCP segment which does not contain any payload, only has the ACK flag set and, if options are present, has only the timestamp option.) While control packets represent a minor portion of network bandwidth, the *number* of such packets received by a server can far outweigh that of packets containing application payload.

In a Web video streaming application, the client sends a small HTTP request to the server. The server responds with megabytes of video data encapsulated in TCP data segments. Software updates, patches and service deployment in datacenters require regular large-scale file distribution to thousands of machines. In return, the data source receives mostly pure ACKs.

Receive-side ACK processing predominantly incurs per-packet overhead, namely the cost of per-packet interactions between the driver and NIC, traversing the network stack and protocol processing. We find that per-packet overhead is significant — 20% CPU cycles of a video-chunk serving workload are expended on processing received packets, 99% of which are pure ACKs.

We propose a *network fastpath* architecture for processing small control packets. The fastpath interface provides an entry point to a significantly simplified networking stack for light-weight protocol processing. In optimizing pure ACK processing, the key insight is that ACKs convey only control metadata to the associated TCP socket, so they need not be delivered as packets. With fastpath processing, pure ACK header values are extracted from received packets and delivered directly to the TCP layer. This contrasts with conventional processing, in which the received packet is encapsulated in a packet buffer[1] and then passed up the stack. Fastpath processing allows packet buffers to be recycled for DMA, reducing memory operations. Moreover, bypassing the bulk of the conventional network stack reduces the number of CPU cycles expended.

We implemented one instance of a fastpath optimization for receive processing of pure ACKs, named TCP-PARO (Pure ACK Receive Optimization), in a recent Linux kernel. We show that it achieves real benefits for server workloads. TCP-PARO lowers application overhead by saving CPU cycles. By reducing per-ACK processing latency, TCP-PARO also improves the throughput achievable by reliable multicast.

[1]For example, *sk_buff* in Linux, commonly referred to as SKB.

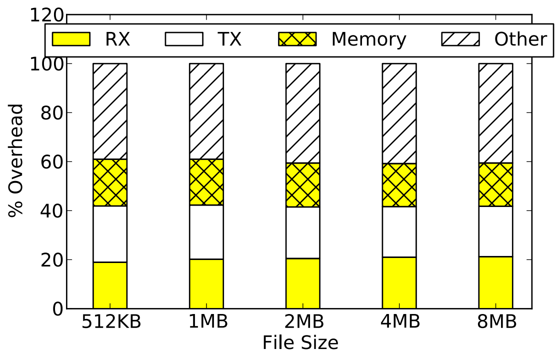

Figure 1: Breakdown of CPU cycles per Web request.

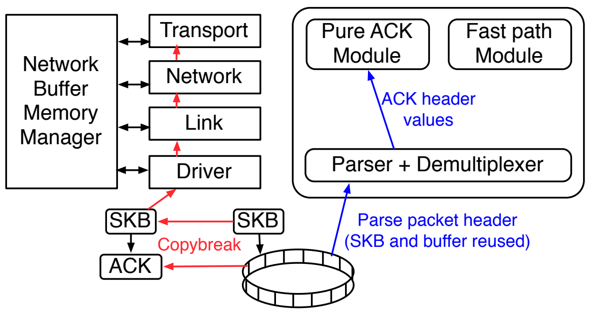

Figure 2: Network fastpath architecture.

2 ACK Processing Overhead Analysis

We quantify network processing overhead with a video chunk serving workload and show that a significant portion of the overhead is due to receive-side network stack processing of pure ACKs.

The tests are run on a testbed with two machines. Each machine is equipped with a quad-core Intel Nehalem processor, 4GB RAM, and a Neterion X3210 10GE adapter. One machine runs the nginx Web server. The 4 NIC hardware DMA engines are bound on 4 different CPU cores, and 4 nginx worker processes are assigned to the cores. This configuration is sufficient to saturate the 10GE link with the generated HTTP traffic. The other machine simulates multiple Web clients using curl-loader. 400 clients are run over 8 CPU threads. Each client requests a random file from a pool of 100 files from the server, receives the file to completion and then repeats. We perform system-wide profiling with oprofile on the server.

Figure 1 presents a breakdown of CPU cycles spent per Web request across various file sizes. The measurements are grouped by system components, where *RX* and *TX* represent receive-side and transmit-side network stack overhead, and *Memory* represents memory operations, including SKB allocations. The figure shows that 20% cycles are spent on network receive processing. Moreover, memory and SKB operations account for 18% of all cycles. The receive processing overhead is significant, considering that the server is mostly sending data, and the bulk of actual application payload is processed by the TX path. However, the receive overhead is comparable to transmit overhead, and is even bigger when serving 4MB and 8MB files. We found that pure ACKs comprised more than 99% of all received packets for all file sizes, thus network receive overhead is indeed dominated by pure ACK processing.

3 Pure ACK Receive Optimization with Network Fastpath

We propose a *network fastpath* architecture which provides efficient packet parsing, packet demultiplexing, and light-weight protocol processing. We describe its design and implementation in Linux, and optimize pure ACK receive processing with this architecture.

3.1 Fastpath Design and Implementation

Figure 2 shows the overall structure of the network fastpath architecture. The fastpath is a light-weight, parallel stack with optimized components for packet parsing, demultiplexing and protocol processing.

The fastpath adopts a modular design. The parser and demultiplexer module reads and parses a packet buffer from the receive-side DMA ring. It serves as a single entry point for network device drivers to deliver packets into the fastpath stack. Based on header fields obtained from the packet, the demultiplexer looks up the fastpath processing module and the socket for which the packet is destined. The fastpath processing module is then invoked on the socket with relevant fields from the received packet.

The fastpath provides a simple, single-function API to device drivers:

```
enum net_fastpath_verdict
net_fastpath_in(struct sk_buff *skb, u32 flags)
```

A verdict is returned to the driver to indicate processing outcomes. If the fastpath is able to process the packet, the *CONSUMED* verdict is returned. Otherwise, the *FALLBACK* verdict is returned. The driver RX routine is patched as follows:

```
if (net_fastpath_in(skb, flags) == CONSUMED)
    goto next_skb;
else
    rx_with_original_stack(skb);
```

Fastpath informs the driver if the packet has been processed. If not, the original stack is the fallback. This allows gradual adoption of light-weight processing for additional packet types without sacrificing protocol support.

Fastpath processing has two main advantages over traditional stack processing. First, the fastpath parses a given SKB, extracts required data and leaves the SKB untouched. Therefore, the SKB and the associated packet buffer can be reused for DMA. This reduces memory allocations. In contrast, existing receive processing clones both the packet buffer and the SKB to ensure isolation of the buffer from concurrent DMA writes by the NIC; Second, the fastpath stack bypasses much of the original stack, and hence reduces cycles expended per packet.

Synchronizing with other kernel threads: Packet receive processing is performed in the soft-interrupt context, which runs concurrently with other kernel threads. A per-socket mutex is held by a concurrent thread accessing the socket. When that thread releases the mutex, it processes received packets in the per-socket backlog. Fastpath similarly introduces a per-packet backlog ring. Parsed header fields are deposited into ring entries. When the mutex is released, the ring entries is flushed. The ring is statically allocated on socket creation. If the ring is full, the earliest entry is overwritten. In our experiments, we found that a ring size of 8 was sufficient. Because each entry is only 32 bytes, the ring adds a modest 256 bytes to socket size.

We note that network taps and Netfilter do not work for pure ACKs when TCP-PARO is used. However, TCP-PARO can also be disabled via *sysfs* should debugging with traditional tools like *tcpdump* be required. We believe this is a reasonable tradeoff between performance and functionality. Moreover, it would be feasible to interface the fastpath stack with existing debugging facilities, though this is beyond the scope of the paper.

3.2 Pure ACK Receive Optimization

We implement TCP-PARO (**P**ure **A**CK **R**eceive **O**ptimization) on top of the fastpath architecture.

Pure ACK parsing and demultiplexing: The parser identifies packets as pure TCP ACKs, extracts the salient header fields and looks up the TCP socket associated with the ACK using the source and destination address/-port pairs. The ACK processing fastpath module is invoked with 5 header fields from the packet — TCP sequence number, ACK number, receiver-advertised window and two timestamp option values. Only pure TCP ACKs are delivered to the processing module.

File size	Copybreak Cycles	PARO Cycles	Savings (%)
512KB	0.80	0.69	14.0
1MB	1.36	1.16	14.7
2MB	2.40	2.04	15.3
4MB	4.60	3.86	16.1
8MB	9.10	7.58	16.7

Table 1: Million cycles per HTTP request.

ACK processing fastpath module: This component quickly processes a pure ACK at the TCP socket level. All link and IP layer processing are omitted, because the ACK has already been parsed and demultiplexed. ACK processing consists of updating the RTT estimate for the connection, removing acknowledged segments from the retransmission queue, performing congestion control and transmitting new segments.

The NIC driver delivers to the fastpath packets that have (1) passed IP checksum test, and (2) have the correct length. In our prototype, a packet is a potential pure ACK if its length is either 54 bytes or 66 bytes, which are lengths of pure ACKs with or without TIMESTAMP option on Ethernet. Some NICs can identify specific packet types, such as pure TCP ACKs, and indicate so in the receive descriptors. The driver can indicate this in a flag when passing the SKB to the fastpath stack, which can skip redundant checks. We added TCP-PARO support to two NIC drivers — e1000e for Intel's PCI-E Gigabit Ethernet controllers and vxge for the Neterion X3120 10GE adapter. Each driver patch consists of 10 lines of code. The pure ACK processing fastpath module is written in 240 lines.

Handling a mixture of pure ACKs and other TCP segments: Whenever a non-pure ACK, such as a SACK, is received, the original path is used for receive processing. Moreover, if TCP-PARO discovers the socket backlog is non-empty, it delegates pure ACKs to the original receive path (by returning the *FALLBACK* verdict to the driver). When the mutex is released, the fastpath backlog ring is flushed, followed by the socket backlog. This scheme ensures all packets for the socket are processed in receive order.

3.3 Performance Evaluation

We repeated the experiments from Section 2 with TCP-PARO. Table 1 shows the number of CPU cycles expended per Web request. *Copybreak* refers to the unoptimized stack which performs SKB and packet buffer cloning (see Section 3.1). With TCP-PARO enabled,

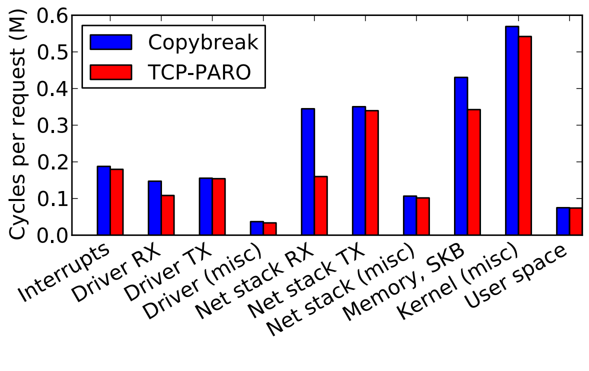

Figure 3: Breakdown of CPU cycles per Web request with and without TCP-PARO. File size is 2MB.

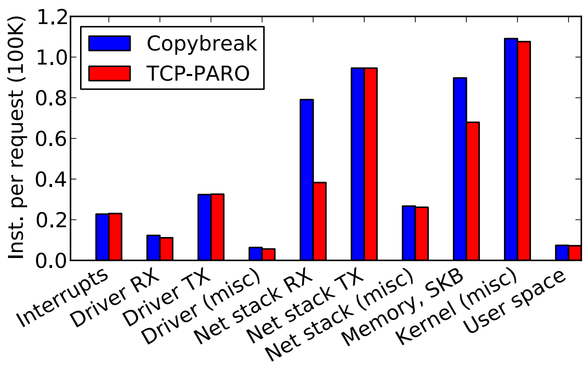

Figure 4: Breakdown of instructions executed per Web request with and without TCP-PARO. File size is 2MB.

CPU cycles are saved consistently across various response sizes. The savings increases with file size. At 8MB, cycles per request is reduced by 16.7%. This is expected because with larger files more data segments are transmitted, eliciting more pure ACKs from the clients.

To understand the sources of cycles savings, we profile cycle expenditure in various system components. Figure 3 shows the profile for 2MB files. TCP-PARO is effective in reducing cycles expended in Driver RX (by 25%), Network RX (by 53%) and memory operations (by 20%). Driver RX includes cycles spent in the driver function for polling packets and setting up fresh RX descriptors for DMA. With TCP-PARO, no SKB is allocated, hence the driver receive routine is lightweight. Network RX includes cycles spent in delivering a packet up the stack and protocol processing. Savings here are due to executing the fastpath processing module, instead of traversing multiple network layers. Our tests were conducted with minimal network stack features, i.e. no network taps or Netfilter modules. Therefore, Network RX for Copybreak represents the minimum cycles expended in receive processing of pure ACKs. The savings in memory operations can be explained by fewer SKB allocations, as SKBs are reused for DMA. Unlike Copybreak, no SKB cloning is necessary for the parsing and ACK header delivery.

Figure 4 presents a similar functional breakdown for number of instructions executed. A major source of saved cycles is from reduced instruction executed. 51% and 24% fewer instructions respectively are executed for network receive processing and memory operations.

4 ACK Optimization for Reliable Multicast

TCP-PARO can be readily integrated with TCP-based reliable multicast protocols, such as TCP-SMO [6]. TCP-SMO is a receiver-driven single-source reliable multicast extension to TCP. The sender maintains a TCP control block (TCB) for each receiver, and aggregates information across all TCBs to produce the multicast TCP state. This state tracks the slowest receiver's earliest unacknowledged number and the minimum congestion and flow control windows. Multicast data segments are ACKed by all receivers. ACK processing at the sender entails updating both the receiver TCB state and the aggregate multicast state.

TCP-PARO integration at the sender is straightforward. The fast path processes ACKs to update the receiver TCBs and then the multicast state. The thread multicasting data is synchronized with the kernel threads performing ACK processing in soft-interrupt. A single pre-allocated ring is used for backlogging pure ACKs in the fast path.

4.1 Performance Evaluation

We study the benefits of integrating TCP-PARO with TCP-SMO. A gigabit Ethernet network of 9 machines connected with a single gigabit switch is used for this study. Each machine is equipped with a Xeon E3-1230v2 processor, 16GB RAM and an Intel 82579LM on-board gigabit NIC. The *e1000e* NIC driver is augmented with TCP-PARO support. One machine acts as the multicast sender and the rest host the receivers.

We ran the sender process and all network processing on a single core. Figure 5 presents a boxplot comparing the total data transfer time with and without TCP-PARO integration into TCP-SMO. Each data point represents 50 experiment trials. TCP-PARO enables near-linear scaling of reliable multicast. The average total transfer time grows from 2.93s to 3.09s (5.6% increase) when number of receivers nearly doubles from 88 to 168. With more receivers, the sender needs to maintain more per-receiver

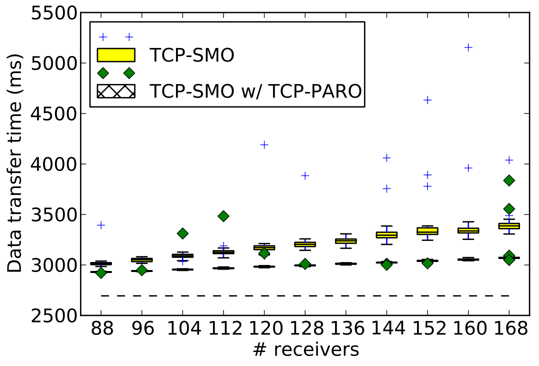

Figure 5: Data transfer time. Sender process and network processing on one core.

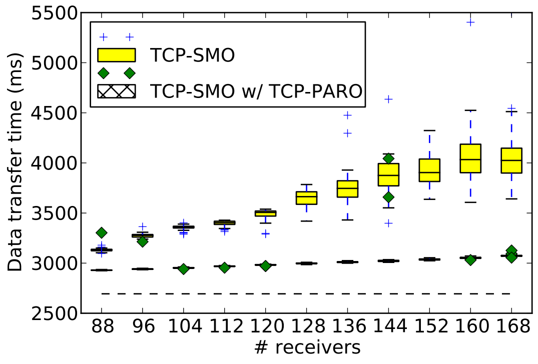

Figure 7: Data transfer time. Sender process on one core. Network processing on multiple cores.

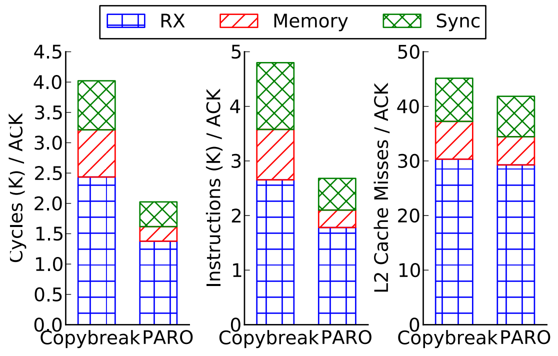

Figure 6: Per-ACK overhead (160 receivers). Sender process and network processing on one core.

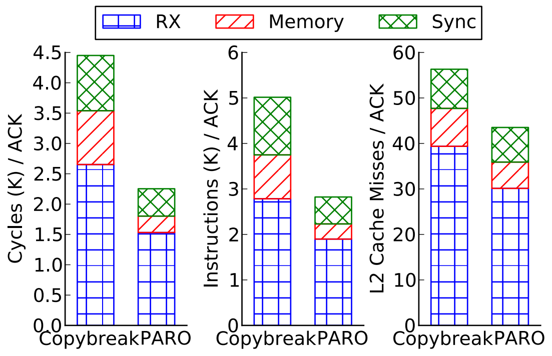

Figure 8: Per-ACK overhead (160 receivers). Sender process on one core. Network processing on multiple cores.

state, and the multicast state updates are more expensive. Moreover, more ACKs are received and processed by the sender. For reference, the black dotted line shows the best possible transfer time on the 1GE network with linear scaling and zero protocol overhead (2.7s). In contrast, without TCP-PARO, the total transfer time grows from 3.02s to 3.4s, a 12.6% increase. Figure 6 breaks down the per-ACK processing overhead into its dominating components. With TCP-PARO, nearly 50% cycles and instructions are saved. The savings in *RX* and *Memory* are due to lightweight fastpath processing and reduced memory allocations. *Sync* includes synchronization operations such as atomic primitives, spinlocks and RCU. TCP-PARO reduces synchronization overhead by 50%, because fastpath execution minimizes lock acquisitions in the original stack which are unnecessary for pure ACK processing. The residual synchronization cost is due to the per-socket mutex. By processing ACKs faster, the sender is able to absorb the burst of ACKs from receivers, thus reducing the transfer time.

We next investigate the potential of parallel network processing on multiple cores to alleviate ACK processing cost. Figure 7 shows a similar boxplot for this setting. The total transfer time with TCP-PARO stays the same, the time penalty growth is similar to the single-core case, and throughput is improved by 33%. In contrast, without optimization, the transfer time grows much more rapidly. The performance difference can be explained by examining per-ACK overhead. In Figure 8, we observe similar cycles savings with TCP-PARO, but L2 cache performance worsens. L2 cache misses increase by about 4% with TCP-PARO, and by 24% with Copybreak. The difference is due to TCP-PARO's reusing SKBs, hence reducing cache miss penalities when cores attempt to deallocate SKBs from other cores' slab caches. Increased cache misses contributed to 12% extra cycles with Copybreak, which in turn translated into 19% increase in the median transfer time and higher variance. On the other hand, the increase in TCP-PARO cycles was modest, hence it did not adversely affect the sender's ability to quickly process ACK bursts. These results show that, with cheap onboard NICs which do not offer multiple DMA engines for parallel processing, ACK optimization can effectively mitigate cache effects in network processing, while preserving the ability to share the NIC among multiple cores. The trend towards smaller computing units in the datacenter suggest that ACK-heavy workloads can expect to benefit from the reduced interactions with memory managers by employing fastpath-based optimizations like TCP-PARO.

The small testbed limited the number of receivers to 168, beyond which the receivers become the bottleneck. Nonetheless, contrary to conventional wisdom, our results indicate that ACK-based reliable multicast can be scaled to non-trivial receiver group sizes, and that the ACK implosion problem can be mitigated fairly well at the end host. We are working on further characterizing the behavior and performance of TCP-SMO with TCP-PARO on 10GE and other environments.

5 Related Work

Per-packet overhead has been identified as a significant overhead in network I/O [3] [7]. Batching optimizations have been proposed to amortize receive-side overhead for data packets [4][8]. Our work focuses on receive processing of small control packets, which are not suitable for batching. Instead, we propose a fastpath architecture which bypasses the original stack and reduces memory allocations.

Alternative packet I/O schemes [9][5] focus on delivering packets from NICs to software. The main motivation of these schemes is to provide a high-performance platform for building software packet processors, such as software routers. However, they do not address packet multiplexing to sockets, buffering, synchronization with multiple threads and protocol processing. They are thus orthogonal to our work.

Partial network offloading techniques such as segmentation offloading and checksum offloading [2], are widely available. Modern NICs implement even more features in the hardware to assist with software stack processing. Examples include filtering packets based on network protocol and performing packet steering to multiple cores [1]. Future NICs can further enhance fastpath performance by implementing packet header parsing in hardware. This could further simplify and improve the performance of the fastpath.

6 Conclusions

In this paper, we investigated optimizing pure TCP ACK receive processing to improve server performance. Pure ACKs consume a small portion of network bandwidth, and have thus received relatively scarce attention in network optimization. However, the rapid increase in network bandwidth, the resultant expectation of higher client-server ratios and the potential benefits of reliable multicast suggest that improving ACK processing efficiency is a real concern in achieving high application performance.

We designed a network fastpath architecture for efficient packet delivery and protocol processing. We implemented TCP-PARO on fastpath, and demostrated 16% cycles savings in an HTTP workload. Moreover, reliable multicast throughput improved by 33% due to reduced ACK processing time.

With the deployment of 40GE and then 100GE, and the desire of lowering server power and space footprint, it is becoming more compelling to improve efficiency of server resource usage. In particular, processing of control packets as a CPU and memory intensive operation is likely to increase in relative cost as link speeds increase and servers handle more clients. Optimizations such as TCP-PARO can be expected to play an important role in reducing the impact of control packet processing on application performance, both on average and in overload cases, as can arise in reliable multicast. We are currently studying other applications of the fastpath architecture, such as SYN flood attack detection, packet accounting and traffic filtering.

References

[1] *Section 7, Intel i350 Gigabit Ethernet Controller Datasheet*. April 2012.

[2] CHASE, J., GALLATIN, A., AND YOCUM, K. End system optimizations for high-speed tcp. *Communications Magazine, IEEE 39*, 4 (apr 2001), 68 –74.

[3] FOONG, A. P., HUFF, T. R., HUM, H. H., PATWARDHAN, J. R., AND REGNIER, G. J. TCP Performance Revisited. ISPASS '03, pp. 70–79.

[4] GROSSMAN, L. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Ottawa Linux Symposium* (2005), pp. 195–200.

[5] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. SIGCOMM '10, pp. 195–206.

[6] LIANG, S., AND CHERITON, D. TCP-SMO: extending TCP to support medium-scale multicast applications. INFOCOM '02, pp. 1356 – 1365.

[7] LIAO, G., ZNU, X., AND BNUYAN, L. A new server I/O architecture for high speed networks. HPCA '11, pp. 255–265.

[8] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. USENIX ATC'08, pp. 85–98.

[9] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. USENIX ATC'12, pp. 101–112.