Master's Thesis (Academic Year 2015)

Improving TCP/IP stack performance by fast packet I/O framework

Keio University
Graduate School of Media and Governance

Kenichi Yasukata

Master's Thesis Academic Year 2015

Improving TCP/IP stack performance by fast packet I/O framework

Summary

Many of server applications are running on computers which are deployed in data centers. There is a requirement that those applications are able to handle huge amount of requests with low-latency. On the other hand, operating systems which are used widely today are not designed for high rate network transaction and they cannot achieve wire rate on 10 Gbps NIC for small message exchanges. Many of researches attack this issue and improve the performance in exchange for less of deployability. This thesis attacks the problems including packet I/O subsystem, synchronization, APIs and other software problems which are not solved by existing researches. This thesis proposes FrankenStack that solves those problems and improves operating systems' TCP/IP stack performance while keeping deployability. The key idea of FrankenStack is combining operating systems' TCP/IP stack and fast packet I/O framework which are designed for different purposes. On request-response workloads, FrankenStack improves throughput by 8.1 to 139.6 % and reduces latency by 7.7 to 58.8 % in 64 to 1024 byte serving message sizes and 1 to 1024 concurrent TCP connections.

Key Words

1. TCP, 2. Network Stack, 3. Operating System, 4. Packet I/O, 5. API

Graduate School of Media and Governance
Keio University

Kenichi Yasukata

Acknowledgment

I would appreciate my supervisor Prof. Hideyuki Tokuda for providing me many chances to study and meet great advisers. I appreciate Dr. Kazunori Takashio and Dr. Jin Nakazawa for advising me continuously from when I was a bachelor student. I would like to express my sincere acknowledgement to Dr. Michio Honda for giving me great opportunities for studying Computer Science.

The work of this thesis has been done at the NetApp GmbH Research Lab in Munich, as the Master thesis in Media and Governance (Cyber Informatics) of Keio University. I would appreciate Dr. Lars Eggert who gave me a chance to work as an internship student. Through the internship, I could learn how European researchers are thinking, making and collaborating.

I would appreciate the members of Hide Tokuda Laboratory. They are always friendly and funny. Thanks to them, I could enjoy my student life in Keio University.

Table of Contents

1 Introduction **1**

1.1 Background 2

1.2 Motivation 2

1.3 Problem Statement 4

1.4 Contributions of this Thesis 5

1.5 Thesis Outline 6

2 Related work **7**

2.1 Systemcall Batching 8

2.2 Packet I/O 9

2.3 Multi-core Scalability 9

2.3.1 `accept( )` Performance 10

2.3.2 File Descriptor Space 10

2.4 Userspace TCP 11

2.5 Specific operating system 11

2.6 Categorization and Analysis 12

2.6.1 API Improvement 12

2.6.2 Userspace Network Stack 13

2.6.3 New operating system 13

2.6.4 Analysis and Design Decision 14
2.7 Summary of this Chapter 15

3 Design **16**
3.1 Problems and Design Space 17
3.1.1 Context Switch and Synchronization 17
3.1.2 Packet I/O Subsystems 18
3.1.3 Application I/O 19
3.1.4 Generality and Flexibility 19
3.1.5 TCP Implementation 21
3.2 Overview of FrankenStack Architecture 22
3.3 FrankenStack I/O 23
3.3.1 netmap 23
3.4 FrankenStack Behavior 26
3.4.1 Getting Packets 26
3.4.2 TCP Receive Processing 28
3.4.3 Application Processing 28
3.4.4 TCP Send Processing 29
3.4.5 Timer Processing 29
3.5 API Design 30
3.5.1 Comatibility to POSIX API 30
3.5.2 Event Notification API 30
3.6 Summary of this Chapter 32

4 Implementation **33**

4.1 Software Components 34
4.1.1 Kernel Modification 34
4.1.2 Packet I/O Framework 34
4.1.3 FrankenStack Kernel Module 34
4.2 Feasibility Challenges 35
4.2.1 Buffer Manipulation 35
4.2.2 Handling Non-Consumable Packet 35
4.2.3 The Case for TCP Transmission Pending 37
4.2.4 Avoiding Sparse Transmission 38
4.3 API 38
4.3.1 Event Notification API 38
4.3.2 Comparison with POSIX API 39
4.4 UNIX Socket Operation 41
4.5 Summary of this Chapter 41

5 Evaluation **42**
5.1 Experiment Setup 43
5.2 Message Sizes and Throughput 44
5.3 Concurrent TCP Connections 45
5.4 TCP Latency 50
5.5 Summary of this Chapter 51

6 Conclusion and Future Work **53**

Chapter 1

Introduction

This chapter describes the background of this research and gives a brief overview of this thesis.

1.1 Background

Cloud computing is a today's computing trend. Powerful computing nodes are deployed in data centers and many of services are running on them. Cloud computing vendors such as Google and Amazon build data centers and supply computing resources to developers. By using those services, developers can allocate and deallocate development platforms and it allows us to create variety of applications.

On the client side, the number of mobile devices is increasing. Smart phones are getting the majority of mobile phones and huge amount of network traffic is generated from them.

Many of inter communications of *server to server* and *server to client* are based on the request-response model. For example, many people use web browsers for fetching web contents and the client devices send HTTP requests to web servers. In the server side, a web server which receives a request makes queries to servers such as storage servers and cache servers for making a response. Those communications are critical for the quality of services and the demand for low-latency and high-throughput communications is increasing.

1.2 Motivation

Request-response is a common and legacy workload of networked applications. While data centers are spreading and many of services are running on them, the importance of such workloads is also increasing. Networked server applications running in data centers need to response to huge amount of requests in low latency. While an important characteristic of request-response

messaging applications is that their message sizes are relatively small (less than MTU of Ethernet which is 1500 bytes in many cases), multi-purpose operating systems which are widely used are not designed for handling huge amount of small packets in a short time. That design mismatch brings big performance problems in the combination with high-performance hardware which is getting cheap and common. For example, 10 Gbps NIC becomes the commodity hardware in data centers, but it is difficult for multi-purpose operating systems to achieve wire rate with small size packets. As a result, design and implementation of operating systems' network stacks become one of the major problems in today's data center network.

There are many of researches attacking these problems. The details of them are described in section 2. In section 2, analyses of those works and categorization of them from the point of view of deployablility are shown. First type is API improvement which requires kernel modifications and API changes for applying it to existing applications. They are using whole of operating systems' network stacks for TCP/IP processing. Second is user space network stack implementation. Minimizing the features of network stacks might improve performance of network processing and it is easy to develop user space network stacks independently to operating systems. For example, netfilter which is provided as default in today's multi-purpose operating systems and it is widely used. However it interferes network transactions and can reduce the performance, and skipping such kind of features will contribute for improving the efficiency of network processing. For applying this type of network stacks, modifications of existing applications are required, but kernel modifications are not necessary. On the other hand, new network

stack implementations do not have well organized development community and they cannot be comparable in terms of stability and capability with operating systems' network stacks which are developed by many of developers for long time. Third is redesigning new operating systems. Since multi-purpose operating systems are designed for flexible and capability of many types of features, they contain many of ineffective parts. New operating system implementation can remove those inefficiency and might be the best option for achieving high-performance processing. However, they do not have compatibility to many of important applications which run on Unix and there is also a problem of the development community which is same as user space network stacks.

1.3 Problem Statement

Today's data centers are constructed with commodity hardwares and multi-purpose operating systems since they are widely used and the qualities of them are reliable. In data centers, the cost for applying new technology should be low and several existing researches do not meet this requirement. Especially, using new or unexperienced operating systems and network stack implementations is risky because they might be fragile and have security problems. On the other hand, network stacks implemented in legacy operating systems are updated frequently for fixing bugs and security issues, and emerging techniques are applied quickly. For these advantages of network stacks in multi-purpose operating systems, improving the performance of them is valuable.

This work focuses on achieving low-latency and high-throughput with operating systems' network stacks. The proposed system reduces the costs of network processing except TCP/IP processing which are not addressed by existing researches yet. The key idea is adopting fast packet I/O framework for existing TCP/IP stack for providing fast data paths for applications. This architecture brings several design problems since TCP/IP stack and packet I/O framework are designed for completely different purposes and this work proposes the solutions for solving the conflicts of them effectively.

1.4 Contributions of this Thesis

This thesis proposes FrankenStack, a new architecture for improving the performance of operating systems' TCP/IP stacks. FrankenStack marries TCP/IP stack to fast packet I/O. This design choice provides following capabilities for applications.

- Zero-copy I/O.
- I/O batching.
- Efficient I/O event handling.
- Reducing scheduling delay between application processing and network stack execution.

Upper 3 capabilities are typical characteristics of packet I/O frameworks and FrankenStack puts those features and operating systems' network stack together. The last capability is deriving from the design choice that FrankenStack executes all network stack processing in systemcall context. This architecture reduces the scheduling delay for switching from network stack pro-

cessing to application processing while network stacks are executed in a kernel thread in multi-purpose operating systems.

FrankenStack is implemented on Linux and evaluate it through benchmarks for identifying how the proposed architecture improves the performance. FrankenStack improves throughput by 8.1 to 139.6 % and reduces latency by 7.7 to 58.8 % in 64 to 1024 byte serving message sizes and 1 to 1024 concurrent TCP connections.

The contributions of this thesis are as follows.

- Design FrankenStack, a new architecture for improving throughput and latency of small data exchanges by extending operating systems' TCP/IP stacks.
- Implement FrankenStack and evaluate its performance through several benchmarks and the costs in network stacks except TCP/IP processing are shown.

1.5 Thesis Outline

This chapter describes the background of this work. Chapter 2 shows the existing works which are attacking the performance problems in the TCP/IP stack implementation and clarify the contribution of this work. Chapter 3 describes the design of the proposed system. Chapter 4 describes the details of the implementation of this system. Chapter 5 describes the experiments. Finally this thesis is concluded in chapter 6.

Chapter 2

Related work

This chapter describes existing approaches for improving the performance of networking transaction and clarify what this thesis focuses on.

Following sections describe the related works with the categorization of them from the point of view of a trade off between performance and deployability.

2.1 Systemcall Batching

Basically applications access the features of operating systems by using systemcall. In networked applications, they call systemcalls for sending and receiving data. Systemcall invokes context switch which changes the memory region and CPU previreges from user space to kernel. Soares and Stumm found out the actual cost of context switch which is not only switching the memory region but also the cashe polution. They have shown the performance reduction caused by frequent systemcall execution. For reducing systemcall related costs, they proposed FlexSC[24] which is a new interface for executing systemcalls. Basic idea of FlexSC is batching. FlexSC makes a systemcall table on the special memory region which is shared between user space and kerenel and it also makes kernel threads for executing systemcalls which are requested by an application via the systemcall table. For executing a systemcall, an application puts a request on the systemcall table. When FlexSC's kernel threads are scheduled by the kernel, they execute the requested systemcalls. Since the requested systemcalls are not executed until the kernel threads are scheduled, systemcall executions are batched. In their paper, they have shown that FlexSC improves the performance of systemcall intensive applications.

2.2 Packet I/O

As described in the previous section, in multi-purpose operating systems, an application has to call systemcalls for asking operating systems to transmit and receive packets. However, since multi-purpose operating systems are not designed for small packet processing, they are ineffective for handling huge amount of small packets. For example, in each `send( )/recv( )` systemcall, the operating systems allocate the memory region for the payload. This won't be a problem when the payload is big (more than 1K Byte), however memory allocation is a heavy workload and frequent memory allocation invoked by frequent packet processing causes big performance degradation.

A new idea is creating a new data path for packet I/O. Packet I/O framework provides zero-copy packet I/O by making the memory region which is shared between user space and kernel, and is DMA mapped to NIC. For reducing the cost of memory allocation, the memory region for payload is preallocated and persistent.

There are several packet I/O framework implementations, netmap[21], DPDK[1], PacketShader I/O Engine[9].

Packet I/O frameworks have been used for packet forwarding purpose, for example high-performance virtual machine inter-connection[22] and middlebox implementation[16].

2.3 Multi-core Scalability

One of the simplest approaches for improving the performance is using multiple CPU cores. However, there are bottlenecks in multi-purpose operating

systems for the scalability of multi-core.

2.3.1 `accept( )` Performance

Request-response is one of the most common workload in today's networked applications. Request-response service is constructed with 3 phases, establishing TCP connection, receiving a request, replying a response.

Pesterev et al. found `accept( )` which is called for TCP connection setup does not scale for multi-core systems since multi-purpose operating systems has a single accept queue. For solving the problem, they proposed *affinity accept*[18]. In affinity accept, operating systems make multiple accept queues which are corresponding to cores for eliminating the contentions for accept queues.

Han et al. also adopting the same idea in MegaPipe[10]. MegaPipe also provides multiple accept queues to improve the multi-core scalability of `accept( )`.

2.3.2 File Descriptor Space

In MegaPipe, they reported the single file descriptor space becomes a problem for multi-core scalability. Since the number of file descriptor is allocated with incremental fashion, all cores need to get a lock of a queue of file descriptor for getting an appropriate number. MegaPipe makes multiple file descriptor spaces and all cores occupy their corresponding spaces.

2.4 Userspace TCP

Since context switch caused by systemcall is a big cost, user space TCP implementation becomes an attractive option. Because its implementation is on user space, it is easier and more flexible to develop new network stacks than in kernel, however packet I/O was the bottleneck in terms of performance of them. Emerged fast packet forwarding techniques described in section 2.2 solved the problem and enable us to develop high-performance user space network stacks. mTCP[12] is an examle of the high-performance user space TCP stack implementation. mTCP runs on fast packet I/O framework and it adopts several promising techniques, I/O batching, unshared file descriptor spaces and unshared accept queues. SandStorm[15] is also an user space network stack implementation. SandStorm is designed for the specific networked applications which are required to have high-capability for high-rate transaction. SandStorm is bypassing kernel for fast I/O by using netmap.

2.5 Specific operating system

The drastic approach is redesigning operating systems for networking transaction. IX[3] and Arrakis[19] are new operating system implementations for high-performance networked applications. They are adopting user space network stack implementations and their packet I/O are bypassing kernel. For accessing NIC, they are using NIC's virtualization support[13] which gives virtual ports to virtual machines directly. In their case, they do not use virtual machines, but assign a virtual port to an application. This design allows applications to access NIC ports directly, and because of the virtual-

ization assist, operating systems can isolate the applications' memory region for safety.

2.6 Categorization and Analysis

	Batching	Pkt I/O	`accept( )`	FD	OS NetStack	Multi-purpose OS
FlexSC	o	x	x	x	o	o
Affinity accept	x	x	o	x	o	o
MegaPipe	o	x	o	o	o	o
mTCP	o	o	o	o	x	o
IX, Arrakis	o	o	o	o	x	x

Table 2.1: Summary of related works

I categorized related works for analysing unsolved problems in network stacks. I discuss them from the point of view of performance and deployability. I show the summary of related works on table 2.6. I do not put packet I/O to any category since they are not the techniques of network stacks. They support the enhancement of network stack performance, and approaches belonging to most of categories can adopt them.

2.6.1 API Improvement

I pick systemcall batching and several optimizations related to multi-core scalability for this category. Performance improvement is limited if I compare with the other categories since they run on multi-purpose operating systems which contains many inefficiency that must be necessary for flexibility. However, the quality of implementations of network stacks are higher than the other options because they have good development communities and they are

developed for long time. We can enjoy new emerging technologies and security updates which are provided by development communities. For deploying these techniques, we need to modify API used in existing applications, but we do not need to reinstall operating system at least. Basically, those techniques can be applied to the other categories and actually they are adopting.

2.6.2 Userspace Network Stack

The second category is user space network stack. Their implementations are independent from operating systems and relatively easy to develop. It is also easy to minimize the features of network stacks and reduce ineffective procedures. Userspace network stack can mitigates the performance reductions coming from systemcall since they do not need to switch their context and go to kernel for the network processing. However, lack of features brings deploying problems for the case where the systems are depending on them.

2.6.3 New operating system

Newly designed operating systems have completely different architectures from widely used operating systems. They eliminate inefficiency in operating systems and network stacks for the performance gain. For applying them, we need to reinstall and replace operating systems and application softwares. Since they are new, they are not experienced well. As a result, deploying reliable systems is difficult because of operating systems' unreliability. It takes many time for softwares to become reliable and it also needs good development communities which are not easy to be established and organized.

2.6.4 Analysis and Design Decision

As many researchers report, multi-purpose operating systems contain ineffectiveness and bigger changes improve their performance much more. On the other hand, there is a tradeoff that the bigger changes result worse backward compatibility and less of features. operating system reimplementation produces the best performance improvement and the worst compatibility to existing systems. Though Userspace network stacks can run on multi-purpose operating systems and have better performance than operating systems' network stacks in many cases, there are many lacks of capabilities. API improvements extend operating systems' network stacks and it is easier to deploy them than the other types of solutions. However, their improvement is limited.

For better deployability and quality of source code, the proposed system is built on existing operating systems' network stacks. When this thesis focuses on the network stacks implemented in operating systems, the competitive option is MegaPipe. MegaPipe adopts other proposed techniques, system-call batching and affinity accept. The big part of MegaPipe's performance enhancement is deriving from improved multi-core scalability and the improvement of single CPU efficiency is limited since systemcall batching only contributes to it. This thesis addresses the single CPU efficiency which is the room for improving TCP/IP stack performance.

2.7 Summary of this Chapter

This chapter shows the related works and clarify unsolved problems. There are 3 important ideas, improving multi-core scalability, redesigning API and reimplementing network stacks. On the other hand, there is the space for improving single CPU efficiency while using existing operating systems' TCP implementations. From the analyses, this thesis proposes a system for enhancing operating systems' TCP stack performance by improving single CPU efficiency. Chapter 3 gives the details of unsolved problems and show the solutions for them.

Chapter 3

Design

This chapter shows design of the proposed system. This chapter describes problems of existing systems and shows the solutions of them as an architecture of the proposed system.

3.1 Problems and Design Space

This section shows the problems that the proposed system attacks. This thesis is addressing following problems.

- Context Switch and Synchronization
- Packet I/O Subsystems
- Application I/O
- operating systems' Generality and Flexibility
- TCP Implementation

For solving those problems, this thesis proposes FrankenStack which is a new architecture for providing operating systems with network services. The following sections describe the details of those problems and show the solutions which are adopted by FrankenStack.

3.1.1 Context Switch and Synchronization

In multi-purpose operating systems, received packets are processed in hardware interruption contexts or kernel threads which are prepared by operating systems. The received packets processed in HW/Soft IRQ context are pushed to appropriate queues which are bound to applications' sockets. An operating system makes an event signaling and wakes up a process for notifying that there is a packet which is destined to an application. The application which receives an event notification wakes up and dequeues the received data and processes it. Until a packet is processed in an application process, there is a context switch from kernel to user space. There are process scheduling

procedures between them. This scheduling delay can increase the latency of request-response server applications.

For reducing this scheduling delay, FrankenStack executes all network processing in systemcall context. After finishing the procedures of systemcall, the processing returns to user space without process scheduling.

3.1.2 Packet I/O Subsystems

Multi-purpose operating systems allocate memory for every data unit. When an application calls a `send( )` systemcall for sending data, an operating system allocates memory in kernel whose size is enough for storing the data. There are packet representation structures like `sk_buff` in Linux and they are also allocated for every data unit. This happens for each `send( )` systemcall and receiver side also has the same kind of memory allocation. Basically, memory allocation is a heavy workload and frequent memory allocation should be avoided for the system's performance. In the case where sending big data (bigger than MTU), this is not a problem because a single memory allocation supplies the space for multiple packets and the memory allocation frequency is not high. However, in high-rate request-response messaging workload, the small memory allocation happens for each reply and brings the performance reduction.

For reducing the cost of memory allocation, FrankenStack preallocates memory region for packets and packet representation structures. Since the size of preallocated memory region is limited, the same region for multiple different packets have to be reused and this is one of the design problems. A design solution for this problem is described in section 4.2.

3.1.3 Application I/O

For sending and receiving data, applications must do it via systemcall. Systemcall does not meet the execution model of I/O batching. Every `send( )` systemcall reaches to NIC's data transmission. For the communication between user space and kernel, memory copies are necessary. Copies are not too much big overhead if the data was on the cache. However, zero-copy helps to improve I/O performance in some cases.

For effective application I/O, FrankenStack gives the control of NIC's I/O scheduling to applications. This design allows applications to batch I/O and control their I/O rate based on their requirements for the latency and the throughput. FrankenStack provides effective data paths to applications. FrankenStack shows NIC's buffer directly to applications and gives the capability of zero-copy I/O. FrankenStack adopts the packet I/O framework for these features. Though improved application I/O enforces the performance, the combination of the packet I/O framework gives us several design problems. The problems and the solutions for them are seen in section 4.2.

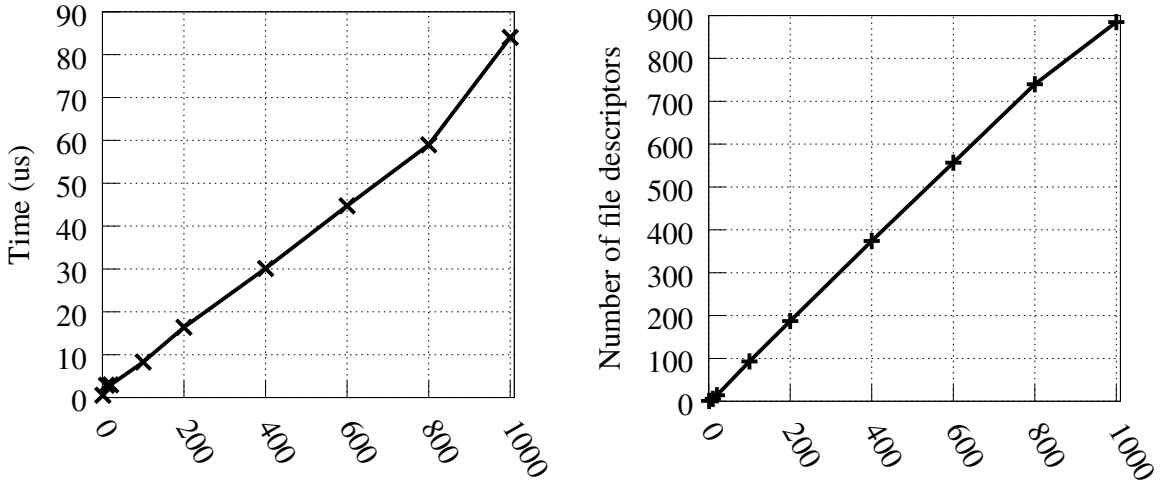
3.1.4 Generality and Flexibility

File descriptor allows us to design and implement flexible and generic applications. File descriptor contains several abstraction layers and these abstractions make applications simple. Applications can access network connections, disk I/O and different processes through file descriptor.

However, there are implementation and procedure overheads which degrades systems' performance for keeping the compatibility to systemcall inter-

faces. Event notification APIs like `select( )` and `poll( )` take many time for preparing event information in callbacks which are bound to operating systems' event notification schemes.

For example, `epoll_wait( )` has linear complexity with the number of file descriptors although it is a Linux extension to POSIX `poll( )` to improve the scalability of this number. In particular, its method to build an array of ready file descriptors has such complexity, which takes several tens of micro seconds depending on the number of file descriptors (figure 3.1(a))). Even worse, concurrent TCP connections increase the average number of file descriptors returned by `epoll_wait( )` (3.1(b)).



(a) Latency to build an array of ready file descriptors

(b) Average number of returned file descriptors

Figure 3.1: `epoll_wait()` complexity versus the number of concurrent TCP connections

[14] also reports long `epoll_wait( )` time when a load is high. I believe it is an unfortunate to preserve semantics in existing software components, such as file structures and synchronization primitives in Linux.

For reducing the costs deriving from the design of operating systems' event

API, FrankenStack provides a new event API which is light-weight and fast. Since FrankenStack has a role for managing the packet I/O layer for the purposes described in this section, FrankenStack can have information of packet I/O events and it is easy to prepare and provide the required information to applications.

3.1.5 TCP Implementation

Recent work has justified TCP stacks in user-space [12, 15] or virtualization execution environments [3] that are implemented almost from scratch or based on a legacy implementation for performance. However, none of them implements modern features, such as latency reduction [2, 5], loss-recovery [6, 17], spurious timeout detection [23], fast connection setup [20] and congestion control [8, 25] algorithms that are necessary to cope with various network conditions. This is not a surprise, because TCP is a complex protocol that supports the Internet; for example, Linux TCP implementation consists of approximately 25K LoC.

This work is motivated to reuse these codes, as well as to follow their active updates. To identify viability of this option, I analysed the latency of TCP/IP "protocol" processing which is only a part of the entire network stack. An observation shows that Linux IPv4 and TCP input packet processing can be reduced to 1–1.5 us (figure 5.5) by improving packet I/O subsystem and APIs, and eliminating context switch overheads. This is not a surprise, because it is known that TCP protocol can be processed very quickly in common cases using a header prediction technique [4].

3.2 Overview of FrankenStack Architecture

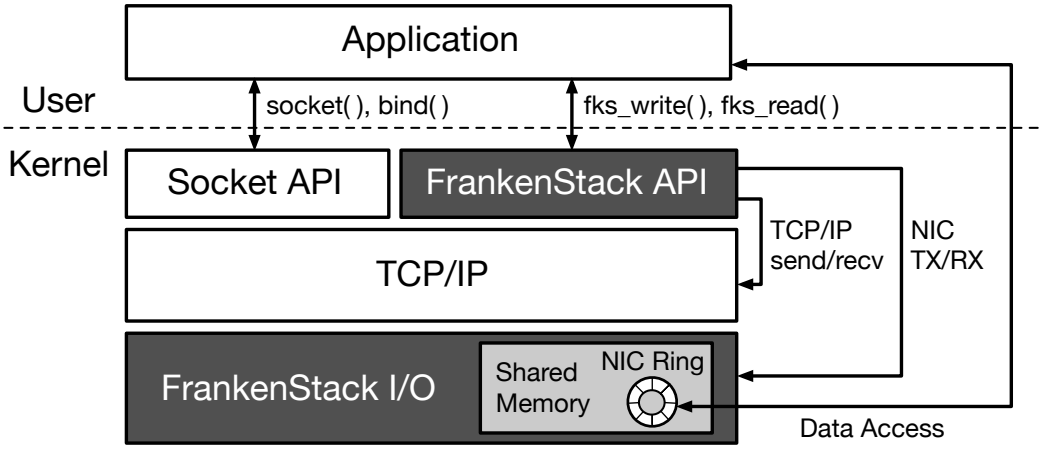


Figure 3.2: FrankenStack architecture: An application actively instruments NIC's I/O and TCP/IP protocol processing via FrankenStack API. The application and TCP/IP directly read and write data on a NIC ring exported by FrankenStack I/O.

Analyses described in section 3.1 bring a set of design principles.

- Context switches between packet I/O, a network stack and an application must be avoided.
- A low cost-per-packet I/O subsystem that uses preallocated, simple packet buffers shared with user space must be needed.
- Low-latency application I/O and event notification must be needed.

The FrankenStack architecture is shown on figure 3.2. For realizing those design principles, FrankenStack adopts packet I/O framework. FrankenStack is constructed with 2 components. First is FrankenStack API which is a set of interfaces for accessing the features of FrankenStack. Second component is FrankenStack I/O which handles packet I/O and is built based on packet I/O framework.

3.3 FrankenStack I/O

Network I/O optimization is a important part of FrankenStack. FrankenStack adopts netmap for improving I/O efficiency. Section 3.3.1 describes the details of netmap.

3.3.1 netmap

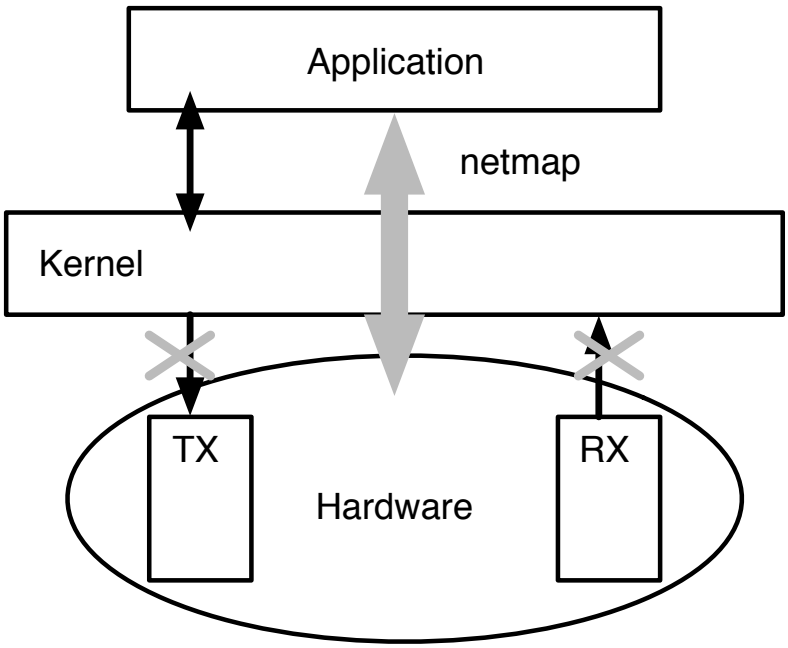


Figure 3.3: netmap architecture

netmap is a fast packet I/O framework which provides fast communication channels for applications. netmap has an abstraction for NIC and it can provides virtual ports which have the same abstraction structure as physical NIC. netmap can achieve line rate of 10Gbps NIC (14.88 Mpps). In FrankenStack, netmap gives fast data paths and enhances packet I/O efficiency.

The architecture overview of netmap is shown on figure 3.3. An important characteristic is that netmap won't crash operating system kernel because their applications run in user space. netmap is an architecture which gives applications fast I/O data paths and efficient synchronization mechanisms. The synchronization is achieved by using systemcall. netmap creates a character device and applications call systemcalls for file descriptors which are bound to the character device. Since netmap's features are executed inside systemcall context, applications cannot execute unexpected behavior for controlling NICs. This architecture allows applications to control NIC safely. The difference from DPDK is that the driver codes are implemented inside the kernel. While DPDK's driver code works in user space, netmap applications need to call a systemcall for accessing NICs.

Effective Packet I/O Subsystem

netmap exposes NIC's buffer to applications directly. Those buffers are preallocated and persistent for reducing the overhead of memory allocation.

netmap API

netmap APIs are implemented by `select( )/poll( )/ioctl( )` systemcalls. Those systemcalls can be used to invoke NIC's data reception and NIC's data transmission. This API design allows systems to control the timing of NIC's tx/rx and applications can batch their transmissions and receptions by their control.

netmap API provides two types of NIC fetch, blocking and non-blocking. `select( )` and `poll( )` yields CPU when there is no received data and the

process is waken up when a new packet arrives for it. `ioctl( )` returns always even if there is no data. In this time, FrankenStack always polls NICs actively by `ioctl( )`.

The following is the pseudo code of a netmap application for explaining how an application reads received data from a NIC.

```
void netmap_read_buf(void)
{
    struct netmap_ring *rx_ring;
    struct netmap_slot *slot;
    int fd, j, k;
    struct nmreq req;

    fd = open("/dev/netmap", O_RDWR);

    bzero(req, 0, sizeof(req));
    strcpy(req.nr_name, "netmap:eth0");
    ioctl(fd, NIOCREGIF, &req);

    while (true) {
        ioctl(fd, NIOCRXSYNC, NULL);

        rx_ring = NETMAP_RXRING(nifp, 0);
        j = rx_ring->cur;
        k = rx_ring->tail;

        while (j != k) {
            slot = rx_ring->slot[j];
            buf = NETMAP_BUF(rx_ring, slot->buf_idx);
            printf("received data %s", buf);
            j = nm_ring_next(rx_ring, j);
        }

        rx_ring->head = rx_ring->cur = j;
    }
}
```

After an application calls `ioctl( )` with NIOCRXSYNC option, the rx ring's information is updated. An application checks the indexes of the rx ring and reads buffers which are placed on indicated slots.

Buffer Swapping

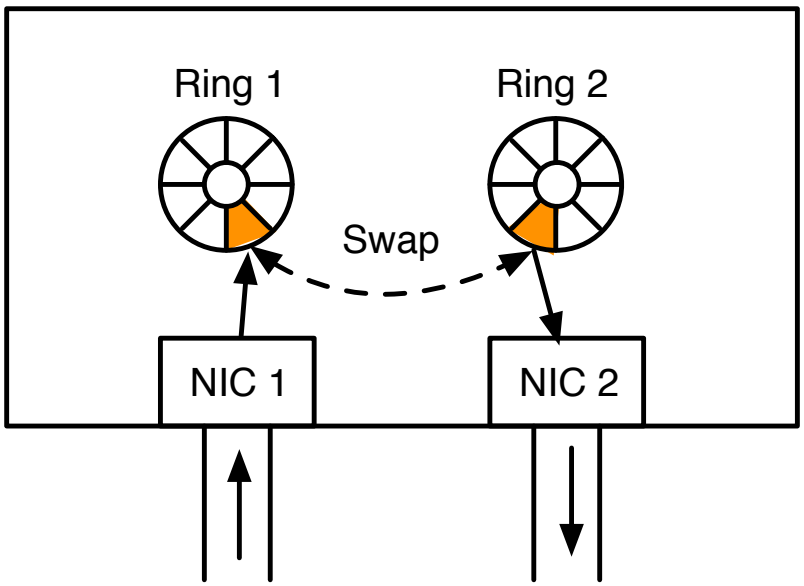


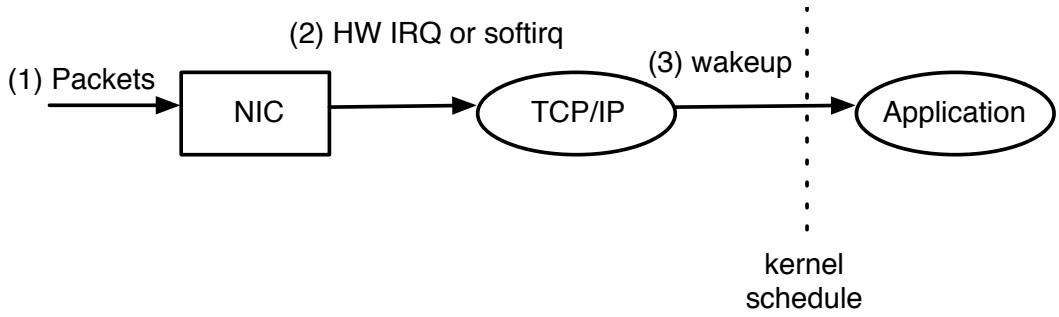
Figure 3.4: netmap buffer swapping example

For achieving zero-copy packet forwarding between different NICs, netmap provides the capability of buffer swapping. Figure 3.4 shows the behavior of it. For swapping buffers, FrankenStack just needs to swap the indexes of NICs' slots. FrankenStack uses this feature for saving un-consumable data.

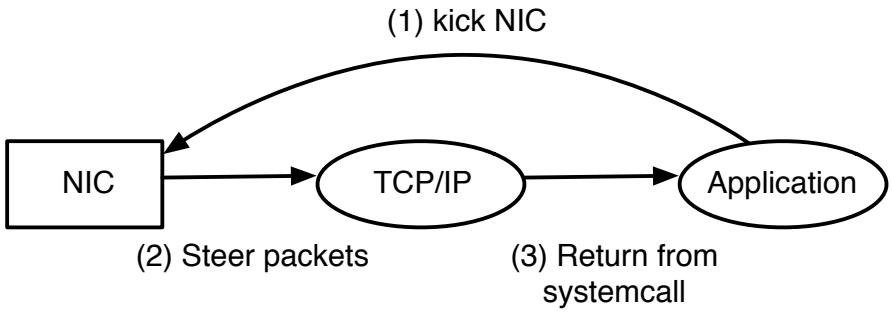
3.4 FrankenStack Behavior

3.4.1 Getting Packets

Figure 3.5(a) and 3.5(b) show the comparison the behavior of packet reception between multi-purpose operating systems' network stack and FrankenStack. An application on top of FrankenStack actively polls a NIC in its own kernel context by calling a FrankenStack systemcall (`fks_read( )`), rather



(a) Multi-purpose operating system packet reception



(b) FrankenStack packet reception

Figure 3.5: Comparison of the packet reception between multi-purpose operating systems and FrankenStack

than running a separate, dedicated kernel thread. This allows us to execute device I/O, network stack and application processing in turn without a context switch, thereby avoiding problems with an expensive context switch and synchronization. FrankenStack instruments a NIC to store packets on preallocated buffers whose memory region has already been mapped into application's virtual address space. This allows the application to access packet data without copy or systemcall after network stack's packet processing. Therefore, this avoids application I/O and packet I/O subsystem problems. There

is an option to use interrupts to detect new packets on a NIC. In this case, NIC's RX interrupt handler just wakes up a userprocess. This option would be useful when he systems want to save CPU cycles in idle periods. Since the focus of this thesis is to reduce latency, I leave exploring this option to future work.

3.4.2 TCP Receive Processing

Returning from each NIC polling with one or more packets received, FrankenStack pushes these packets into a TCP/IP protocol suite. The TCP/IP implementation processes these packets in turn as if these packets are coming from the operating system's packet I/O subsystem. If a packet is an in-order TCP segment, the TCP implementation enqueues it into a socket buffer so that an application can consume its data later. Since FrankenStack is running in application's systemcall context, FrankenStack can skip synchronization procedures that are necessitated by executing a network stack in a software interrupt context .

3.4.3 Application Processing

Immediately after returning from the FrankenStack systemcall or `fks_read( )`, an application consumes data that are made available in the last network stack processing. The application is notified of ready file descriptors in a similar way to `epoll_wait( )` or with an array of them. However, to avoid inefficiency of `epoll_wait( )`, FrankenStack uses the lightweight version. The applications could generate new data, such as "HTTP OK" in response to "HTTP GET". Application I/O to and from the network stack can be done

without `read ()/write( )` systemcalls, because as mentioned earlier, both TX and RX packet buffers have been mapped into application's address space. It should be noted that while standard `write( )` invokes network stack's output processing and device I/O immediately, FrankenStack postpones it until the application explicitly triggers it.

3.4.4 TCP Send Processing

After filling up TX buffers, the application invokes another FrankenStack systemcall (`fks_write( )`) to trigger TCP/IP stack to process these packets and push them out. In this systemcall, the network stack processes these buffers by putting protocol headers, but does not trigger device I/O yet. After all the sending packets have been ready to send, this systemcall kicks device I/O. This allows us to push packets over multiple TCP connections in a single device I/O trigger action.

3.4.5 Timer Processing

To avoid locking a TCP connection structure, FrankenStack replaces all the timer event handlers, such as a retransmission timeout handler, just to post occurrence of their timeouts. FrankenStack executes their original handlers every time an application executes the network stack or calls `fks_read( )` or `fks_write( )`.

Name	Description
int fks_write(int fd, const void *buf, struct nm_desc *nmd, size_t count, int ring_id)	Write data to NIC ring directly
int fks_read(int fd, void *buf, struct nm_desc *nmd, struct nm_desc *exd, size_t count, struct fks_winfo *fwi, int ring_id)	Read data to NIC ring directly
int fks_close(int fd, int ring_id)	Close socket
int fks_reg_session(int fd, int sock)	Register socket with FrankenStack

Table 3.1: Overview of FrankenStack API

3.5 API Design

3.5.1 Comatibility to POSIX API

FrankenStack API is designed for applying FrankenStack to exiting systems easily. Table 3.5 shows the API examples. For the better compatibility to the existing systems, FrankenStack API provides `read( )/write( )` API. Basic parts of `fks_read( )/fks_write( )` are same as POSIX API's `read( )/write( )`. Additional arguments such as *ring_id* and *nm_desc* are needed for netmap specific information including the pointers to netmap's packet buffers.

3.5.2 Event Notification API

I believe an application wants to process TCP connections or file descriptors in turn as with today's event notification API like `epoll_wait( )`. However, scanning all the received packets in the ring for every file descriptor has a scalability problem, particularly when the number of file descriptors is large. Unfortunately, this is the case in transaction workload that handles a large number of concurrent TCP connections. On the other hand, the event

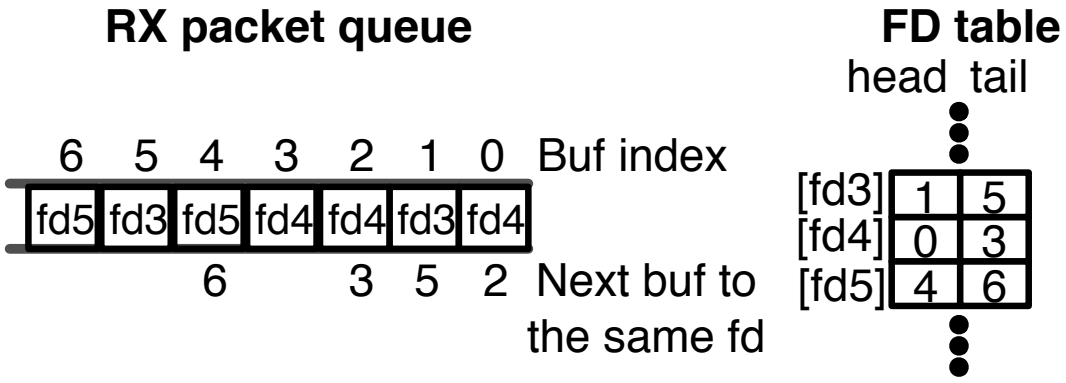


Figure 3.6: Algorithm to build a ready file descriptor list

notification API, or `epoll_wait( )`, has linear complexity when it builds an array of ready file descriptors that is returned to the application.

To avoid these problems, FrankenStack builds such an array during the network stack execution or `fks_read( )`. Figure 3.6 shows the algorithm with an example that packet 0, 2 and 3 go to a file descriptor (fd) 4, packet 1 and 5 go to fd3, and packet 4 and 6 go to fd5. Note that fd is guaranteed to be a unique number in the entire user process. When packets in the receive ring are processed, each packet buffer keeps a next buffer index that goes to the same fd. FrankenStack uses a table (FD table) that is indexed by fds to keep a head and tail packet index for each fd. The tail is used to append a new packet to a fd without having to traverse the ring, and the head is used by an application to find the first packet for each ready fd also without having to traverse the ring. The table size is a product of 32 byte (two 16 byte buffer indexes) and the maximum number of file descriptors (65K in Linux). While it is not so large, FrankenStack can further save memory using a flexible array as file descriptors are allocated from lower integers. FrankenStack also builds an array whose element contains a ready file descriptor (not present in the

figure). By scanning this array, identifying the first packet buffer index for each file descriptor and traversing next buffer index alongside each packet buffer, the application can process ready file descriptors.

3.6 Summary of this Chapter

This chapter shows the details of the addressing problems and overview of solutions. This chapter describes the design of the proposed architecture, FrankenStack.

Chapter 4

Implementation

This chapter describes the implementation of the proposed system. The proposed system brings several feasibility challenges and the solutions for them are also shown in this chapter.

4.1 Software Components

FrankenStack is implemented with a kernel module, packet I/O framework and kernel modification. FrankenStack is implemented based on Linux 3.16. The biggest part of FrankenStack is implemented in the kernel module and the kernel module glues the features of packet I/O framework and the kernel TCP/IP stack.

4.1.1 Kernel Modification

Linux kernel is modified 188 LoC (61 LoC additions and 12 LoC deletions in 11 existing files and two new files) for adding fields in exiting data structures of kernel for keeping the FrankenStack's metadata.

4.1.2 Packet I/O Framework

FrankenStack adopts netmap[21] for packet I/O framework. netmap code is modified with 68 LoC for adding variables in existing structures which are needed for keeping metadata used in FrankenStack.

4.1.3 FrankenStack Kernel Module

FrankenStack kernel module is a 2200 LoC character device that implements FrankenStack systemcalls or `fks_recv( )` and `fks_send( )` to steer packets between Linux TCP/IP implementation and netmap-based packet I/O subsystem.

4.2 Feasibility Challenges

4.2.1 Buffer Manipulation

FrankenStack needs packet buffers used by operating system bypass, user space packet I/O framework. How can we glue such a packet buffer to pass into operating system's network stack? Simply gluing each of packet buffers with operating system's standard one brings overhead associated with allocation and deallocation cost which is known to be expensive. FrankenStack therefore pre-allocates operating system's packet representation structure alongside each of preconfigured, fixed-sized packet buffers. A `free` routine of kernel's memory allocator is modified such that a buffer is not actually freed. As a result, FrankenStack can remove the vast majority of overheads with packet representation and preserve zero-copy and systemcall-batching capability achieved by operating system bypassed user space packet I/O framework.

4.2.2 Handling Non-Consumable Packet

After TCP's input packet processing, a packet buffer cannot be freed when its packet has not been consumed. This is the case when a received TCP segment was out-of-order. To be effective, device drivers only sequentially process packets in their ring buffer so that just two pointers indicate "available" and "occupied" blocks. This means, if an unconsumed packet sits in the middle of the ring, the driver cannot use buffers after this position. To cope with this problem, FrankenStack swaps such a packet buffer with a free packet buffer. The free packet buffer is configured in the same contiguous

memory region such that an application can access it using a unique index or offset after it becomes consumable. After swapping, FrankenStack sets the new buffer brought into the ring structure to DMA. Although buffer swapping comes with some costs as it requires DMA remapping, it could be negligible as far as it is not very frequent. This buffer swapping technique was originally introduced by netmap[21] to achieve zero-copy packet forwarding between different rings or NICs. In the paper of netmap, he reports one swapping on every minimumsized packet forwarding between 10 Gigabit ethernet NICs easily achieves the line rate. The similar, but severer condition would happen in a transmission path. Transmitted TCP packets on a NIC ring cannot be released until they are acknowledged by the receiver. Therefore, it might seem that these in-flight packets quickly exhaust packet buffers in the transmission ring or very frequent buffer swapping is needed. However, this does not happen so frequently. $PacketRate \times RTT$ packets could be sent in one RTT. For example, on a 10 Gbps link whose line rate is 810 Kpps with 1500 byte packets, 162 packets could be sent at the same time when RTT is 200 us, which is relatively long or happens when some intermediate switch queues have been built up. The ring buffer typically has a few thousands of packet slots (e.g., Linux driver for Intel 82599 10 GbE NIC supports up to 4K descriptors), which allows FrankenStack to wait for all the packets acknowledged for several RTTs. Therefore, to avoid buffer swapping as much as possible, FrankenStack does not perform that until it is needed. It should be noted that as explained in the next section, FrankenStack explicitly tells the application the number of available slots and bytes with regard to TCP's window.

4.2.3 The Case for TCP Transmission Pending

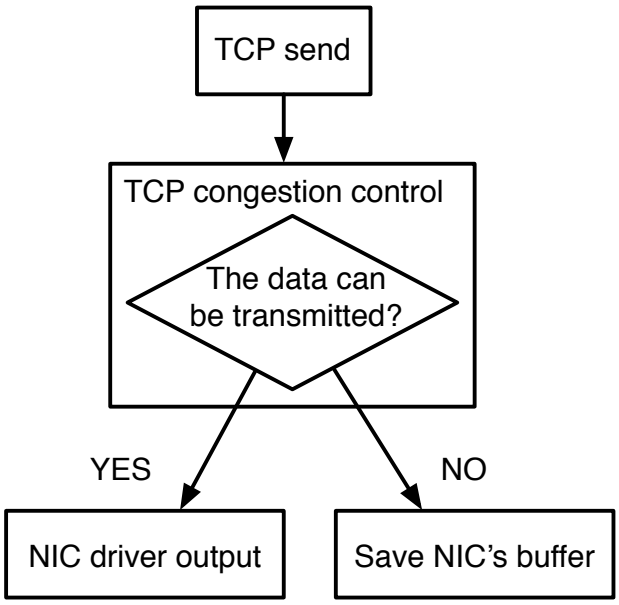


Figure 4.1: Procedure flow for TCP transmission pending

FrankenStack provides a safety data saving technique for the case where an application pushes too much data and exceeds TCP's transmission capacity. Figure 4.1 shows how FrankenStack handles TCP's congestion control. FrankenStack checks the specified data is transmitted or not after a send function returns and if the data is not transmitted, FrankenStack saves the data by using buffer swapping. The reason why FrankenStack use buffer swapping for this purpose is that buffer swapping allow us to keep TCP/IP stack's packet representation structures and they are not needed to be modified. Because those structures are used in many parts in kernel for providing many features, they are very complicated and they should not be touched for simplicity. When the system recovers the capacity of TCP transmission after

it receives several ACKs, TCP tries to transmit the pending packets. Because their packet representation structures are not modified, TCP's transmission works as usual.

4.2.4 Avoiding Sparse Transmission

After a TCP implementation processes an application's data waiting to be sent on the TX ring, FrankenStack flushes these packets by triggering the NIC's I/O. In order for fast packet I/O, FrankenStack has to transmit all the packets from a head to tail position on the TX ring. However, if an application has put an arbitrary amount of data on the ring for multiple TCP connections, packet transmission could have to be sparse due to lack of the window size on some TCP connections, causing excessing buffer swapping. FrankenStack therefore notifies an application of perconnection available bytes (the smaller of advertised receiver window and congestion window) and total number of available slots.

4.3 API

4.3.1 Event Notification API

Variables are added in the netmap's structure (`netmap_ring`) for storing the number of events and file descriptors of the events. This is the same fashion as `epoll` and it is easy to apply to applications which adopt `epoll` based event notification. The variables indicating the status of events are updated in the kernel space. Since FrankenStack has the control of NIC, it can prepare required event information quickly.

4.3.2 Comparison with POSIX API

For comparison, the pseudo codes for implementing simple HTTP servers with FrankenStack API and POSIX API are shown below.

FrankenStack micro HTTP server

```
void franken_http_server(void)
{
    struct netmap_ring *rx_ring;
    struct fks_winfo fwi;
    int i, fd, nfd;
    char buf[256];
    char *httpmsg;
    ssize_t len;

    rx_ring = NETMAP_RXRING(na->nifp, ring_id);
    nfd = rx_ring->nevt;

    for (i = 0; i < nfd; i++) {

        fd = rx_ring->evts[i];

        bzero(&fwi, sizeof(struct fks_winfo));

        len = fks_read(fd, buf, na, exna,
                       sizeof(buf), &fwi, ring_id);

        if (len < 0) {
            fprintf(stderr, "read failed");
            continue;
        }

        len = http_parse_and_create_response(httpmsg, buf, len);

        if (httpmsg) {
                fks_write(fd, httpmsg, na, len, ring_id);
        } else {
                fks_close(fd, ring_id);
        }

    }

    rx_ring->head = rx_ring->cur = rx_ring->tail;
```

```
}
```

POSIX micro HTTP server

```
void posix_http_server(int epfd, struct epoll_event *evts)
{
    int i, fd, nfd;
    char buf[256];
    char *httpmsg;
    ssize_t len;

    nfd = epoll_wait(epfd, evts, SESSION_MAX, -1);

    for (i = 0; i < nfd; i++) {

        fd = evts[i].data.fd;

        len = read(fd, buf, sizeof(buf));

        if (len < 0){
                fprintf(stderr, "read failed");
                continue;
        }

        len = http_parse_and_create_response(httpmsg, buf, len);

        if (httpmsg) {
                write(fd, httpmsg, len);
        } else {
                close(fd);
        }

    }
}
```

Those pseudo codes indicate that the required modification for applying FrankenStack is not big. The basic components of a server application is event API and network I/O. FrankenStack replaces the event API with FrankenStack's effective API which keeps high readability since its way of description is close to `epoll`. The APIs for network I/O are also easy to understand

because they are based on file descriptor. We can apply FrankenStack by replacing POSIX API.

4.4 UNIX Socket Operation

FrankenStack relies on operating systems' socket API for setting up TCP connections. FrankenStack does not have features for making sockets, but just uses created sockets by operating systems' API. This means FrankenStack is completely compatible to file descriptor and it is easy to apply existing applications which are handling TCP connections based on file descriptor.

FrankenStack works just for specified sockets and packets for unspecified sockets are steered to the normal operating system's TCP/IP stack. By this design, FrankenStack achieves transparency for non-FrankenStack enabled applications. FrankenStack does not occupy NICs for FrankenStack's purpose but shares with the other running applications.

For enabling FrankenStack on a specific socket, an application registers it by using FrankenStack API. After the registration, FrankenStack can identify which received packets should be processed by it.

4.5 Summary of this Chapter

This chapter describes the implementation details of FrankenStack. The feasibility challenges which are brought by FrankenStack's design choices are shown and the solutions for those design challenges are proposed.

Chapter 5

Evaluation

This chapter describes the evaluation of the proposed system. Experiments are purposed to measure how FrankenStack improves the throughput and latency. In this time, HTTP workloads are adopted for the benchmarks. Through the experiments, characteristics and benefits of the prosed system are shown.

5.1 Experiment Setup

Experiments are executed for the FrankenStack prototype to analyze its latency and throughput as well as to validate the design decisions. This thesis focuses on transaction workloads on a server that serves request-response traffic with small packets on a large number of concurrent TCP connections.

Two machines are used for the experiments. They are connected via a single 10 Gigabit Ethernet link, which are called *server* machine and *client* machine. Both machines are equipped with Intel Xeon E5-2650 (2.00 GHz) CPU, 128 GB RAM and an Intel 82599ES chipset 10 Gigabit Ethernet card.

The server runs either Linux network stack or FrankenStack. On top of them, Linux or FrankenStack version of a simple HTTP server runs, respectively. The Linux version runs on epoll event loop and processes events on file descriptors in turn. Every processed event includes `read( )`ing the client's request, matching the first four-byte string to check "HTTP GET", copying a pre-generated HTTP response into a buffer and `write( )`ing this buffer into the file descriptor. The FrankenStack version runs on `fks_read( )` event loop and bypasses all of the `epoll_wait( )`, `read( )` and `write( )` systemcalls.

On the client machine the Linux network stack is used and run an existing HTTP benchmark tool wrk[7]. wrk initiates a given number of TCP connections, then continually sends an HTTP GET request to receive an HTTP OK response over them. On each TCP connection it does not send a next request until it receives a response for the last one. The client reuses TCP connections or never re-establishes TCP connections, because this thesis focuses on the small message exchange problem.

In all the experiments the server machine uses only one CPU core, because this is the first experiment and I am interested in how FrankenStack solves the problems described in chapter 3. The client machine uses all the 32 CPU cores as well as RSS to efficiently utilize these cores. Unless otherwise stated, all considerable hardware/software offloading options for default TCP/IP stack are enabled. For FrankenStack case, all of hardware offloading features are disabled because FrankenStack conflicts with them.

5.2 Message Sizes and Throughput

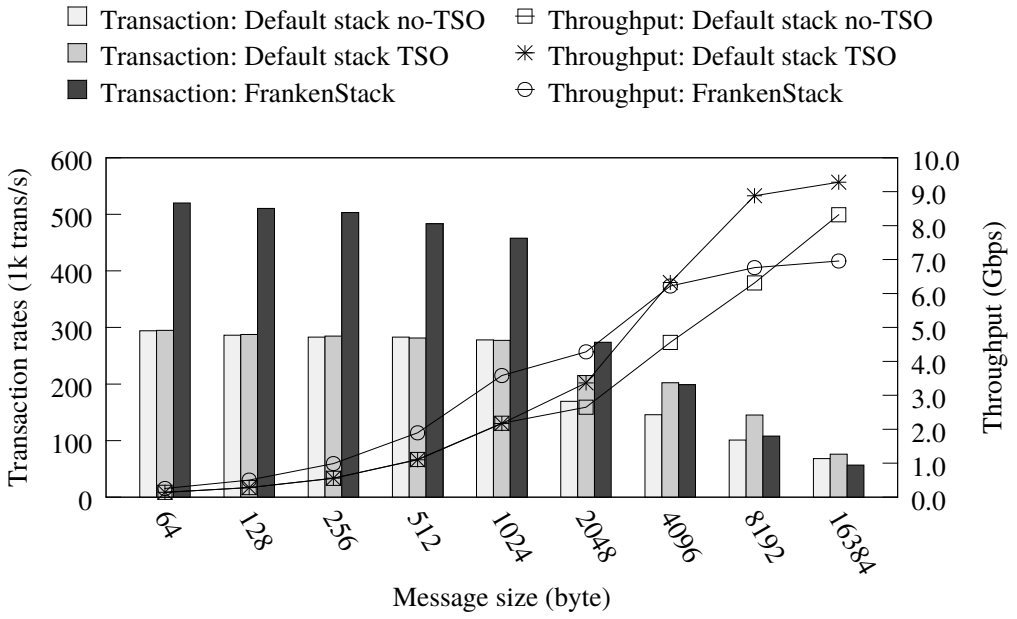


Figure 5.1: Transaction rates and throughput on different message sizes

Figure 5.1 shows throughput and transaction rates on different response message sizes. Response messages that are 2048 bytes or larger consists of multiple packets to fit into 1500 byte link MTU. FrankenStack is compared

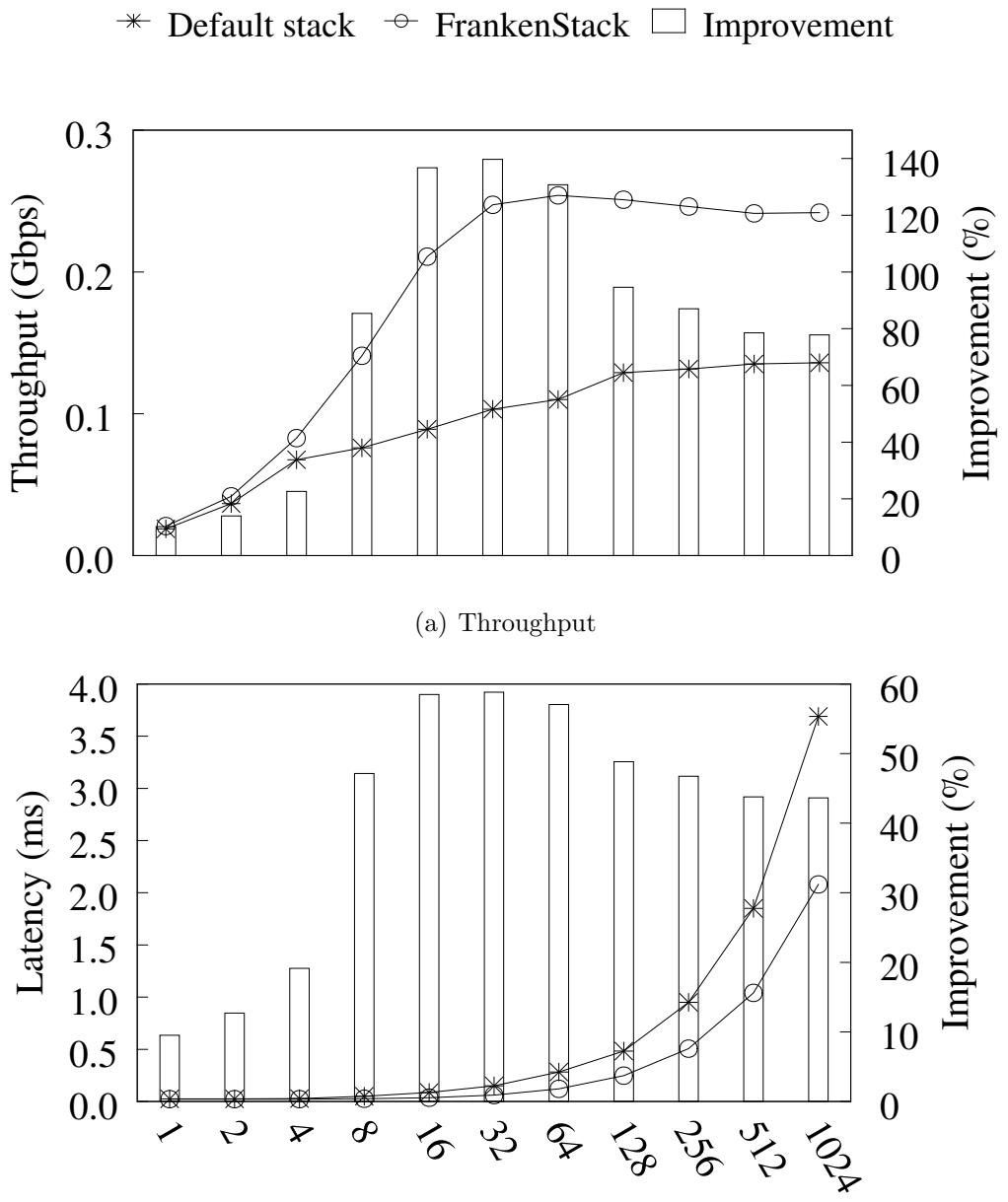
with two versions of Linux that enables and disables TSO. Note that TSO is effective only when a response message includes multiple packets. This experiment purposes to measure the achievable throughput for different message sizes; the benchmark therefore increases the number of concurrent TCP connections from 1 to 2048, and figure 5.1 plots the best numbers for each message size and network stack. Analyses of the relationship between concurrent TCP connections, throughput and latency are shown later. FrankenStack achieves 27.2 % to 77.5 % higher throughput than Linux with TSO when the response message consists of one or two packets or its size is 2048 bytes or smaller. FrankenStack competes with Linux with TSO for 4096 byte message size, and its throughput is lower than Linux with TSO when the message size is 8192 or larger. FrankenStack's throughput is slightly lower than Linux even without TSO. This is because of Generic Segmentation Offload (GSO) which is a software version of TSO but still reduces the number of IPv4 and TCP protocol processing executions. In future work I plan to support GSO or TSO in FrankenStack.

5.3 Concurrent TCP Connections

The throughput and latency of FrankenStack are measured with regard to the number of concurrent TCP connections or transactions. Figures 5.2, 5.3 and 5.4 show throughput and latency with three message sizes. FrankenStack improves throughput by 8.1 to 139.6 % and reduces latency by 7.7 to 58.8 % in 64 to 1024 byte serving message sizes and 1 to 1024 concurrent TCP connections.

From these experiment results, I see that the performance gains are deriving from the event API improvements. In Linux, the increased number of concurrent TCP connections increases latency exponentially. This is because it increases the number of file descriptors returned by `epoll_wait( )`. Since FrankenStack eliminates the complexity with `epoll_wait( )` by the algorithm in section 3.5.2 and reduces `read( )/write( )` systemcall overhead, the latency reductions by higher margin with an increased number of the concurrent TCP connections are seen.

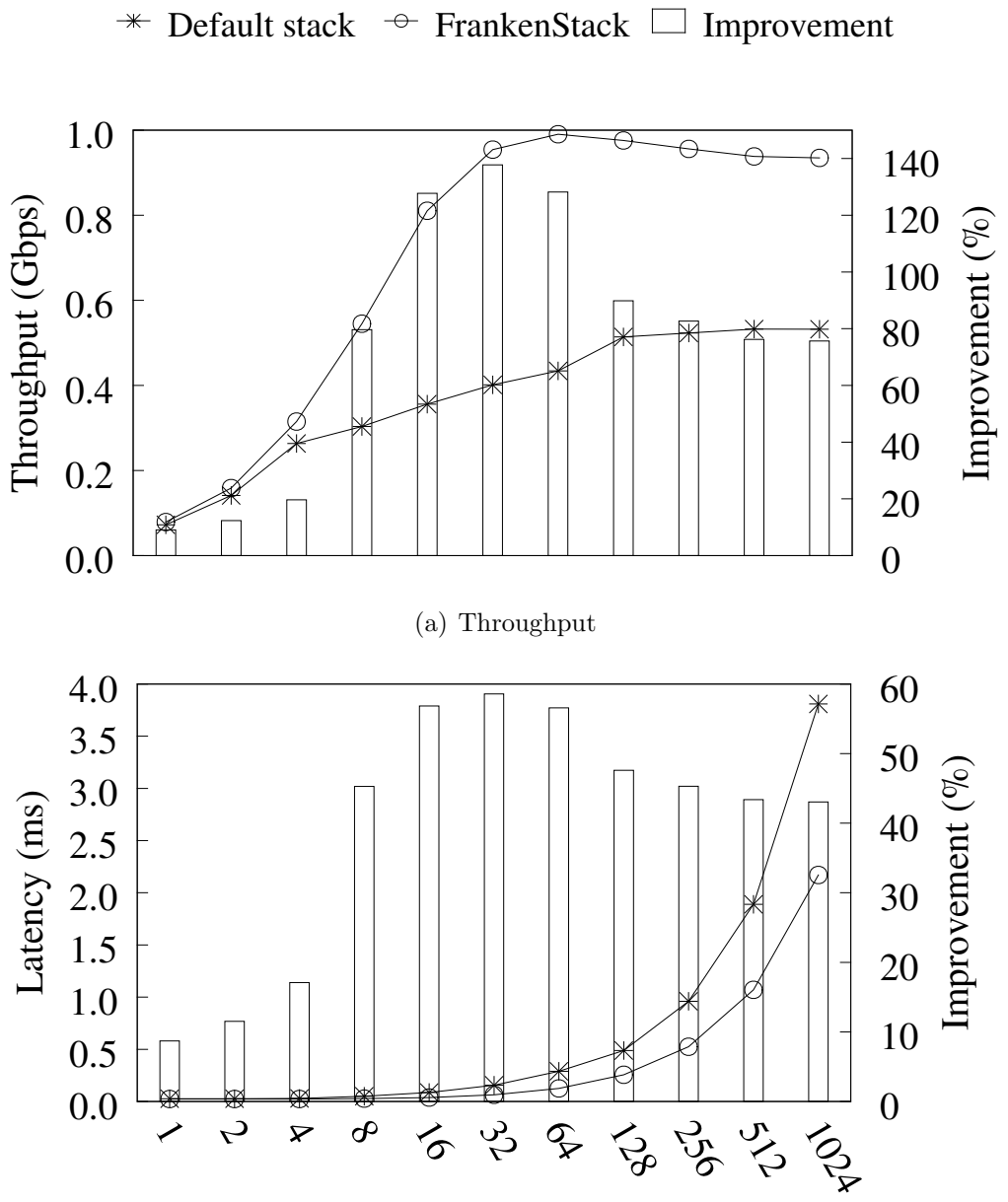
The observations show the reduced number of TCP connections processed by a single event notification, which is 62 and 235 for 128 and 1024 concurrent TCP connections, respectively (not present in the plots). This contrasts to Linux with `epoll` where almost all the concurrent TCP connections are processed on every event notification.



(a) Throughput

(b) Latency

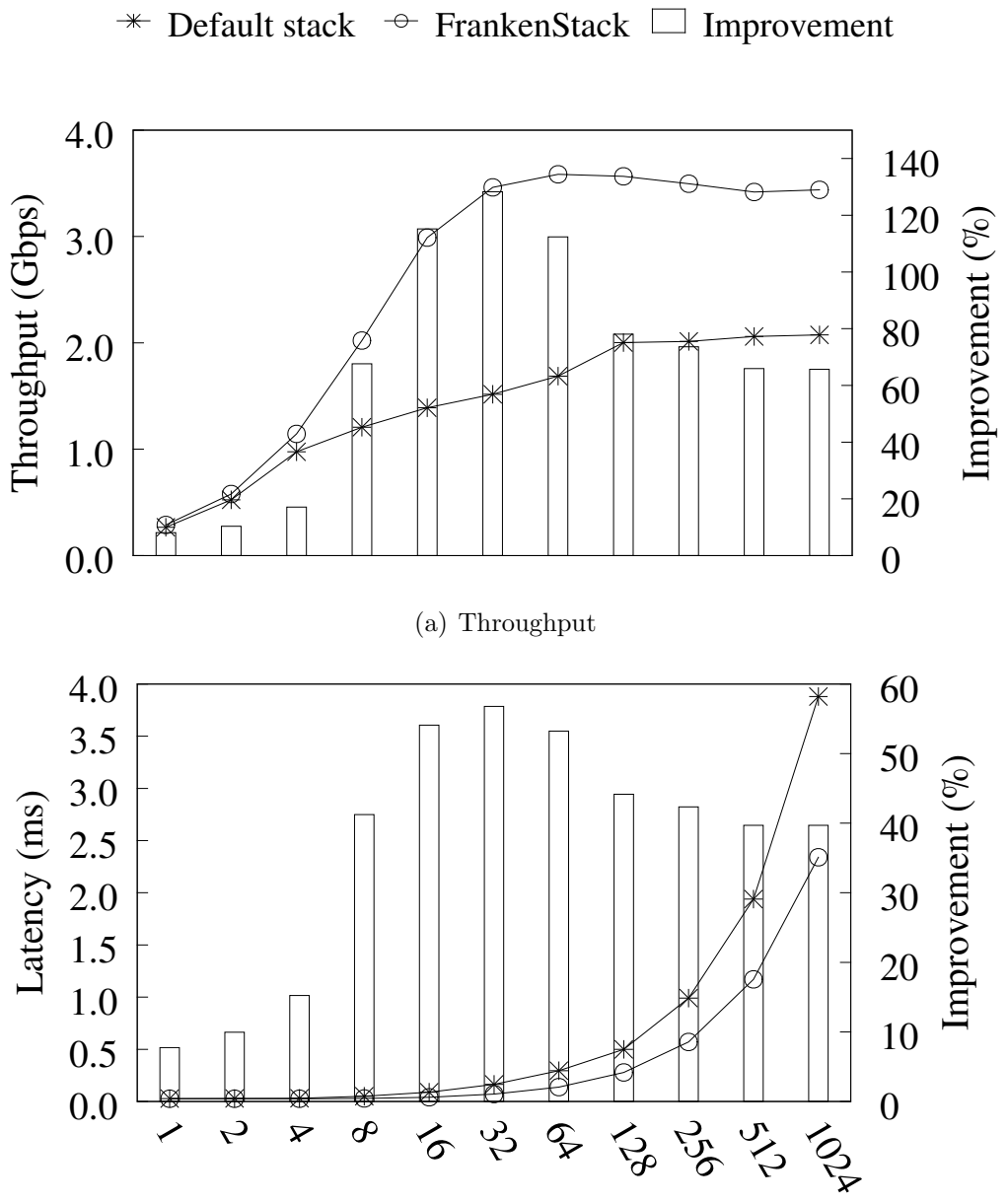
Figure 5.2: Throughput and latency with the number of concurrent TCP connections (horizontal axis) for 64 byte messages



(a) Throughput

(b) Latency

Figure 5.3: Throughput and latency with the number of concurrent TCP connections (horizontal axis) for 256 byte messages



(a) Throughput

(b) Latency

Figure 5.4: Throughput and latency with the number of concurrent TCP connections (horizontal axis) for 1024 byte messages

5.4 TCP Latency

Finally, this section analyzes latency in TCP protocol processing to see the least possible latency with FrankenStack architecture that leaves Linux TCP implementation. The elapsed time from the beginning of TCP/IPv4 input packet processing to the end of that is observed. Figure 5.5 shows the results. The Linux network stack and FrankenStack take 1.3-1.9 us and 1.0-1.5 us at 50th percentile, respectively. These latency reductions are effects of eliminated synchronization from a network stack to an application. Out of these latencies, 0.4-0.7 us (Linux) or 0.3-0.5 us (FrankenStack) are spent by the IPv4 implementation. Overall, latency in the Linux TCP implementation is quite low, and it is further reduced to 1 us or less by FrankenStack architecture.

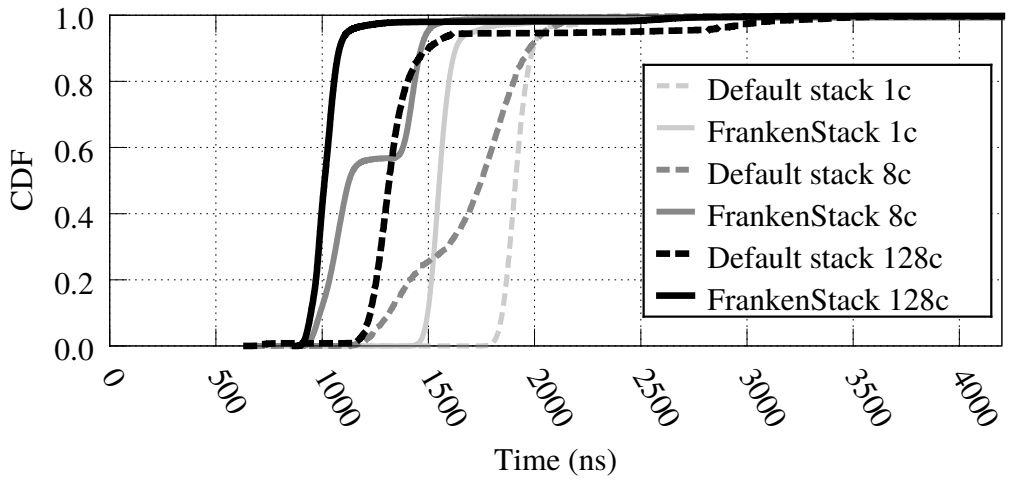


Figure 5.5: Elapsed time in IPv4 and TCP input packet processing (`napi_gro_receive( )`) for 1, 8 and 128 concurrent TCP connections: FrankenStack takes shorter because it avoids synchronization procedures.

5.5 Summary of this Chapter

This chapter shows the details of experiments. For measuring the improvements of the throughput and latency of FrankenStack in request-response server applications, experiments are executed for HTTP server workloads. Regarding the throughput, experiment results show that FrankenStack achieves 27.2 % to 77.5 % higher throughput than Linux in the case where the message sizes are less than 2048 bytes. FrankenStack produces the better performance than Linux if the message size is less than 2048. If the message size is bigger than 4096 bytes, FrankenStack could not produce performance improvements. I see this is because of the procedures of TCP checksum and packet fragmen-

tation which can be done by hardware assist in Linux TCP stack. Regarding the latency, experiment results show that FrankenStack reduces the latency by 7.7 to 58.8 % in 64 to 1024 byte serving message sizes and 1 to 1024 concurrent TCP connections. Especially the measurement results for elapsed time in TCP receive processing show that FrankenStack reduces its packet processing time. This is because FrankenStack reduces the synchronization costs including process wake up and the preparation of event information.

Chapter 6

Conclusion and Future Work

This chapter describes future work and concludes this thesis.

Request-response is an important workload in today's data center networks. Especially small message exchanges can be seen widely, for example HTTP and memcached protocol. On the other hand, multi-purpose operating systems cannot achieve high performance with small packet sizes. Several researches rebuild the implementation of TCP stack itself for the performance. However, being motivated by the fact that operating system's TCP implementation is the most state-of-the-art in terms of features and it is the most actively maintained, this thesis addressed problems with operating system's network stack while preserving its TCP implementation. This thesis presented FrankenStack and improved throughput and latency in transaction workloads.

There are still questions, such as scalability with the increased number of CPU cores and how to run a large number of applications concurrently. One issue is protection which is an important role of operating systems. FrankenStack allows an application to see all the NIC's buffers and this might bring the security problems. Second is the resource sharing schema, this is also a feature which should be provided by operating systems. A considerable solution for them is adopting MultiStack[11] which runs multiple different protocol stacks on virtual ports which are provided by VALE[22]. Since each virtual port is assigned to each application process and packet buffers are copied to each process's memory region, the kernel does not need to show NIC's buffers to any application. Because packet steering is executed by VALE, the resource sharing can be achieved by it.

For identifying the viability of FrankenStack, more experiments in different situations are needed, for example, the validation of multi-core scalability

and benchmarks for existing applications.

I will address these problems and validate the system in future work.

Bibliography

[1] Data Plane Development Kit. `http://dpdk.org/`.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[4] David Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, 1989.

[5] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. *draft-dukkipati-tcpm-tcp-loss-probe-01*, Aug 2013.

[6] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for tcp. In *Proceedings of the 2011 ACM*

SIGCOMM Conference on Internet Measurement Conference, IMC '11, pages 155–170, New York, NY, USA, 2011. ACM.

[7] Github. Modern HTTP benchmarking tool. `https://github.com/wg/wrk`.

[8] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[9] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[10] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, Berkeley, CA, USA, 2012. USENIX Association.

[11] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. Rekindling network protocol innovation with user-level stacks. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, April 2014.

[12] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Imple-*

mentation, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association.

[13] Patrick Kutch. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. Intel application note.

[14] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM.

[15] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 9:1–9:7, New York, NY, USA, 2013. ACM.

[16] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX Association.

[17] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining tcp congestion control. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 281–291, New York, NY, USA, 1996. ACM.

[18] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Pro-*

ceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, pages 337–350, New York, NY, USA, 2012. ACM.

[19] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[20] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. Tcp fast open. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 21:1–21:12, New York, NY, USA, 2011. ACM.

[21] Luigi Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[22] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[23] Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen. F-rto: An enhanced recovery algorithm for tcp retransmission timeouts. *SIGCOMM Comput. Commun. Rev.*, 33(2):51–63, April 2003.

[24] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[25] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM*, pages 1–12, 2006.