I am broadly interested in computer systems and networking. My current work focuses on *data center systems*. Data center systems provide critical infrastructure for cloud computing. The particular need for high efficiency and high reliability places a number of challenges on the entire system stack, including both the physical infrastructure (e.g., server hardware, data center networks) and the system software (e.g., hypervisor, operating system). For example, a cloud network has to deliver terabits per second of traffic reliably and efficiently in spite of frequent failures (e.g., packet loss, link/switch failures) in the underlying data center network. An operating system has to provide low-overhead virtualization support for tens to hundreds of containers and virtual machines simultaneously. My research addresses these challenges by building useful systems and tools to *improve the reliability and efficiency* of data center systems. To this end, I have provided a wide spectrum of solutions from data center network failure debugging to operating system support for efficient container virtualization.

The key theme of my research is *usability*. I want my research to be usable and deployable to make a real-world impact. My methodology consists of three parts: (1) understand and identify *real* problems including existing systems' inefficiency or failures; (2) build *practical* solutions to solve these problems; (3) evaluate my solutions by *deploying* them in realistic settings. This methodology drives me to produce usable and deployable solutions. Some parts of my research in physical infrastructure have been deployed in Microsoft data centers worldwide for monitoring and debugging their networks. My recent research on operating system support for efficient container virtualization has won recognition by the UW CSE industrial affiliates for commercialization potential.[1]

In the future, I am planning to broaden my scope in the area of systems. This includes continuing solving problems in data center systems in various aspects (e.g., efficiency, reliability, security). Also, I want to apply my knowledge in operating systems and networking to other emerging fields (e.g., Internet of things, mobile systems, machine learning systems).

Thesis Work

Data center systems are the foundations of the cloud. They provide the core services that enable multiple cloud customers to share the computer resources. Data center systems consist of two parts: (1) a physical infrastructure, including server hardware and data center networks, (2) system software to virtualize the physical infrastructure, including hypervisors and operating systems.

In both parts, the primary metrics of interest are reliability and efficiency. Reliability is important because companies rely on the cloud to run critical businesses. Even the slightest downtime results in lost productivity and revenue. Efficiency means cloud platforms can pro-

[1] Madrona Prize: https://www.cs.washington.edu/industrial_affiliates/madrona

vide services at a low cost, benefiting a large number of companies and organizations. Unfortunately, achieving high degrees of reliability and efficiency is challenging. On the reliability side, various types of failures happen in data center systems (e.g., packet losses, server crashes), threatening the availability of the cloud. Existing tools fall short in preventing, localizing, debugging, and mitigating them. On the efficiency side, our current software system stack and network architecture incur significant overhead and excessive cost.

My thesis answers two key questions in data center system design: (1) how to design a network architecture that is both low cost and reliable? (2) how to design a safe and extensible operating system that supports efficient virtualization?

Network Architecture

One key infrastructure for data centers is the network. Today's data center connects thousands of racks of servers using tens of thousands networking devices (e.g., switches, links). Failures (e.g., device failure, packet loss) happen frequently because of the sheer number of devices and a large amount of traffic these devices carry. To tolerate these failures, we need to understand their characteristics and built automated tools to identify and mitigate them. At the same time, we want the network architecture to incur a low cost.

Mitigating packet corruption errors. Packet losses in data center networks hurt application performance and can lead to millions of dollars lost revenue. Even 0.01% of packet loss rate can cause user-facing network traffic to slow down by 50%, significantly hurting the user experience. Researchers have explored a wide range of approaches to minimize packet loss, such as congestion control, load balancing, and traffic engineering. All of these approaches focus only on congestion loss—packet loss when the network is overloaded. Another source, packet corruption, has received little attention.

Working with Microsoft, we studied 350K links across 15 production data centers to show that packet corruption is significant and its characteristics differ markedly from congestion losses. Corruption impacts fewer links but imposes higher loss rates. Also, the corruption rate is stable over time, weakly correlated with link utilization. This observation implies that reducing the load on network links (e.g., congestion control, load balancing) does not reduce the packet corruption rate.

Based on these characteristics, we designed and implemented CorrOpt [2], a system that automatically detects and mitigates packet corruption. CorrOpt minimizes corruption losses by intelligently disabling corrupting links, while ensuring the network has enough capacity for its current workload. CorrOpt also recommends specific actions (e.g., replacing equipment, cleaning cable connectors) to repair disabled links. Our evaluation shows CorrOpt can reduce corruption loss by three to six orders of magnitude. A version of CorrOpt's link moni-

toring and recommendation engine is deployed in Microsoft data centers, increasing the repair accuracy by 60%.

Low-cost data center network architecture. The cost of the network is an important metric for data center network design. Costs include all the main contributors, such as switches, links, and power. Several new data center network architectures have been proposed to reduce the cost of networking, including topology design, congestion control, and load balancing. However, the physical layer (i.e., optical fiber) is often ignored.

Using the same infrastructure, we monitored optical links across Microsoft's production data centers for more than 10 months. We found a remarkably conservative state of affairs: 99.9% of links have an incoming optical signal quality that is higher than the minimum threshold. Even the median is 6 times higher. This over-engineering is expensive because transceivers (i.e., devices that convert signals between electrical and optical domains) account for 48-72% of the total data center network cost depending on link length distribution.

Motivated by this observation, we propose lower the network cost via using transceivers beyond their specified limit. Our benchmark shows that the reach of multiple commodity transceivers can be stretched by 1.6-4 times of their specification. However, 1-5% of data center network paths will suffer significant packet loss.

We designed and implemented RAIL [3], a system that ensures applications only use network paths that meet their reliability requirement. RAIL generates multiple virtual topologies on the same physical network topology. Each topology guarantees a maximum packet loss rate. Applications bind to different topologies based on their needs. Our evaluation shows that RAIL reduces data center network cost by 10% for 10 Gbps and 44% for 40 Gbps network.

Operating System

Operating systems are critical system software for data center systems as they provide virtualization and isolation for cloud customers. Today, the cloud is moving from using virtual machines to using containers because containers are more lightweight. Thus efficient operating system support for container virtualization becomes important. At the same time, as developers frequently need new operating system features to serve emerging needs, how to modify operating system in a safe way becomes critical.

Operating system support for efficient container virtualization. Containers have become the mainstream method for hosting large-scale distributed applications. Containers are attractive to many users because containers are both lightweight and portable. Container virtualization is more lightweight than virtual machine because OS virtualizes its resource at the system call level rather than emulating the raw hardware (for virtual machines). The container

overlay network is a key component to support distributed containerized applications. It allows containerized distributed applications to have their own network configurations, fully decoupled from that of the host network.

Unfortunately, today's container overlay network imposes significant overhead in terms of throughput, latency, and CPU utilization. The key problem is that the existing container overlay network implementation uses packet encapsulation for network virtualization. This means each packet has to traverse OS network stack twice and also a virtual switch on both the sender side and the receiver side. This design resembles overlay network architecture for virtual machines. Because the virtual machine has unilateral control of its network stack, the hypervisor has to send/receive packets without the context of network connections. However, the OS kernel has full knowledge of network connections that are created inside containers.

We designed and implemented Slim [4]. Slim virtualizes Linux container networking at connection setup time by manipulating the connection metadata. Slim achieves lower overhead because packets only go through the network stack once. Slim is compatible with existing applications. Our evaluations show that Slim improves an in-memory key-value store by 66% and reduces its latency by 42%. Slim reduces CPU utilization of the in-memory key-value store by 54%. Slim also reduces CPU utilization of a web server by 28-40%, a database server by 25%, and a stream processing framework by 11%. Slim has won recognition by the UW CSE industrial affiliates for potential commercialization impact.

Safe extensibility of operating systems. OS kernels are rapidly changing to adapt to emerging hardware and use cases. The Linux kernel currently supports loadable kernel extensions. These extensions to deliver key functionalities in the cloud. For example, containers (i.e., Docker) depend on Open vSwitch kernel module for network virtualization, AppArmor to provide security, and OverlayFS to provide file systems for stackable container images.

Because these kernel extensions run in the same privilege level as the rest of the kernel, bugs in them can cause significant problems. Bugs (e.g., NULL pointer dereference, use-after-free, out-of-bound access, deadlock, race condition) cause the kernel thread of the extension to crash, potentially leaving the kernel in a bad state. Existing solutions to sandbox extensions require intrusive changes which prevent adoption.

Together with two other graduate students I am mentoring, we are designing and implementing Stone, a framework for developers to write safe kernel extensions. Developers write Linux kernel extensions in Rust, a memory-safe and race condition-free programming language. Stone framework help developers reason about the interaction between the kernel extension and the rest of the kernel. We have ported several existing kernel extensions into Stone. Our evaluation shows that the ported kernel extensions avoid bugs found in their Linux counterparts. We also show that the safe kernel extensions have the same performance as their less safe counterparts.

Other Work

I have collaborated with my colleagues on additional research directions, including how best to add capacity to a data center network, debugging of network failures, and verification of middleboxes.

Incremental network capacity. The demand for network bandwidth has been increasing rapidly in data centers. Operators face challenges to add capacity to their existing network to accommodate that demand with minimal changes to the existing infrastructure. This problem is particularly important at the edge of the network (i.e., switch to server network links).

The common approach taken is to make use of multiple network interfaces on a physical machine. By connecting multiple (e.g., 2x) network interfaces to switches, the network capacity of the server increases (e.g., by 2x). However, several questions are still not answered: (1) which switch should the additional network interface card connect to? (2) how should routing work when a server has multiple network interfaces? (3) how to load balance across multiple network interfaces when network interfaces are connected to different switches?

We answer these questions with Subways [1], a co-design of topology, routing, and load-balancing. Subways cross-wires servers to the top-of-rack switches in the neighboring rack. This switch is then wired to a different cluster. We use adaptive load balancing to distribute load across different clusters. The evaluation shows that Subways improves MapReduce shuffle by 3.1x and throughput of Memcached by 2.5x, relative to an equivalent-bandwidth network that keeps wirings local to a rack.

Diagnosability of gray failures in data center networks. As shown in CorrOpt [2], gray failures are common—persistent failures that allow a fraction of packets to go through. Sometimes these can be detected by switch counters, but often the switch may not know it is the cause of the problem. Further, load balancing in the network can mean that the end host which observes its connection performing badly, may have no idea of the root cause. The next connection will be randomly assigned a different route, and so it will appear as a glitch that simply never gets fixed.

We designed and implemented Volur [6], a data center network architecture that gives control over routing to the end host so that it can detect if there is a problem in the network, even for gray failures. Volur does this by changing how load balancer functions in the network: the network behavior needs to be predictable, and the end host decides how to distribute network load. Our evaluation shows that Volur can route around to failures fast (within a fraction of a second) and report to network operators about gray failures accurately.

Middlebox verification. Middleboxes (e.g., network address translators, load balancers, firewalls) play a critical role in providing performance and security for modern networks. However, building bug-free middleboxes is still an unsolved problem. Critical bugs have been routinely found in today's middleboxes, causing system failures and exposing security vulnerabilities.

We designed and implemented Gravel [5], a framework for developers to design, implement, and verify middlebox implementations in a push-button fashion. Gravel provides developers a domain-specific programming language to express high-level properties of the middleboxes and allows automated verification in a low-level C implementation of the middlebox. The key technique of Gravel is to reduce verification effort by leveraging the data flow decomposition of a middlebox's logic. Our evaluation shows that middleboxes built atop Gravel achieve similar performance as their counterparts today and can avoid bugs similar to those found in their counterparts.

Future Work

In the future, I want to keep exploring new ideas in data center systems. There are still many remaining challenges in providing a secure, efficient, and reliable cloud. I will also seek opportunities to apply my knowledge of operating systems and networking to other related fields.

Application-level security properties. Architecture platforms for secure application programming is emerging (e.g., Intel SGX, ARM TrustZone). These platforms are attractive because they allow cloud users to run applications whose content is not visible to cloud providers even if they compromise the operating system. In this way, applications that contain sensitive data (e.g., private medical records) can also enjoy the elasticity and availability of the cloud.

One caveat of using such a framework is that even when the application is protected, the application still needs the untrusted operating system to provide access to resources (e.g., file systems, network). This means developers have to think carefully about a malicious operating system's behavior. I would like to build a framework to reason about the security of applications atop such secure architectures, extending the architecture-level security guarantee to provide application-specific security goals. The framework should allow developers to write application-specific security goals (e.g., privacy, integrity) and automatically check whether the implementation of the application meets these goals.

Automated compiler and run-time for heterogeneous network hardware. Networking hardware has become increasingly heterogeneous. Many vendors manufacture their own types of switches and network processors. When a middlebox (e.g, firewall) is deployed, developers

have to write a new program for a particular hardware. Later on, the developers also have to tune the program for better performance or for deployment in a distributed setting.

What we actually want is a compiler and a run-time that automatically deals with these issues. Developers should specify high-level intent of the middlebox (e.g., firewall), and the compiler should generate efficient low-level code on different kinds of hardware backends. More interestingly, we can build a run-time that automatically decides which device a functionality should be placed on in a distributed setting. The run-time should adapt to the workload and scale up/down accordingly.

Containers with kernel extension support. Linux has wide support for extensibility: it can be configured with various extensions to customize its behavior. Containers have become the de facto method for hosting applications. Compared to virtual machines, containers have one major drawback: containers cannot inject their own kernel extensions. The reason is security: a kernel extension injected into a container OS has physical machine privilege and thus can see the entire memory address space. This means if kernel extensions are allowed, the container must be trusted by every other container on the same hardware.

To allow safe kernel extension in the container context, we need to rethink the existing mechanism for container isolation. Today, containers depend on kernel mechanisms (e.g., namespace, cgroup) for isolation. These approaches are not directly applicable to kernel extensions. Instead, we need a verification-oriented approach to isolate different kernel extensions.

Understanding timing channels in modern operating systems. Timing channel attack (e.g., Meltdown, Spectre) is a particularly critical problem for the cloud. The cloud packs multiple users' applications onto the same machine, and timing channel attack means one user is able to steal data from or deny service to other users on the same machine by observing/changing the pattern of use of shared resources such as OS shared buffers. With the advent of containers, we need to carefully understand timing channels in modern operating systems.

Timing channels can exist anywhere where resources are shared across different containers. For example, if both containers use the same network interface packet buffer, one container is able to tell whether the other container is sending traffic by monitoring its own latency on network-related system calls. I want to carefully understand whether timing channels exist in each operating system's component for container virtualization. Further, I want to build a container isolation mechanism so that timing channels do not cross container boundaries.

Real-time operating system kernel for the Internet of things. The Internet of things is emerging quickly. Many smart cameras, voice assistants, thermostats have already been deployed in homes. In the future, smart edge devices can have a broader impact on how people live, how people transport, how companies manufacture, and how agriculture works.

Today, more than 70% of IoT devices run Linux and very few run real-time operating systems. I believe this is because developing atop Linux is much easier as Linux provides a rich set of features, such as a well-tuned networking stack. However, a drawback is that Linux is not a good fit for real-time applications because it does not provide real-time guarantees. Even today's Android phones (based on Linux) suffer from responsiveness issues. To solve this problem, we need to rethink an approach to allow real-time responsiveness and also preserves the rich set of features in modern operating systems.

References

[1] Vincent Liu, **Danyang Zhuo**, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A Case for Redundant, Inexpensive Data Center Edge Links. In *CoNEXT*, 2015.

[2] **Danyang Zhuo**, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and Mitigating Packet Corruption in Data Center Networks. In *SIGCOMM*, 2017.

[3] **Danyang Zhuo**, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks. In *NSDI*, 2017.

[4] **Danyang Zhuo**, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *NSDI*, 2019.

[5] Kaiyuan Zhang, **Danyang Zhuo**, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Design and Verification of Software Middleboxes using Gravel. Technical report, University of Washington, Paul G. Allen School of Computer Science and Engineering, 2018.

[6] Qiao Zhang, **Danyang Zhuo**, Vincent Liu, Petr Lapukhov, Simon Peter, Arvind Krishnamurthy, and Thomas E. Anderson. Volur: Concurrent Edge/Core Route Control in Data Center Networks. *Arxiv*, 2018.