Statements of Research and Teaching Interests *Yang Zhou*

I am a systems researcher spanning the areas of networking, operating systems, and distributed systems, focusing on datacenter environments. A datacenter centralizes hundreds of thousands of machines with high-speed networks, enables computations over huge amounts of data, and hosts popular services (e.g., Google search, Netflix streaming, ChatGPT) that impact billions of people's lives.

To handle massive-scale data and computations, datacenter applications run across multiple machines over networks. Ideally, the underlying datacenter infrastructure should be efficient to maintain steady cloud revenues while meeting high user expectations, and be evolvable to handle the increasingly diverse and performance-hungry applications as well as heterogeneous hardware. However, application-level goals are having a growing **mismatch** with the goals of host networking stacks (involving NICs, kernels, transport layers, and threading) that play a core role in connecting machines, causing severe efficiency and evolvability problems. For example, the most widely used kernel networking stack prioritizes security and isolation with separated kernel and user contexts, incurring prohibitive CPU overheads; meanwhile, emerging in-memory applications demand ultra-low latency and high throughput, preferring coalescing different contexts but losing isolation. Even though the networking stacks keep evolving, e.g., the modern kernel-bypass RDMA stacks, applications tend to be network-unaware and abuse (or even deplete) network resources. This fundamental mismatch causes less efficient use of datacenter resources and hinders the scaling-out of diverse applications.

My research has focused on bridging the mismatch by **codesigning** low-level host networking stacks and high-level datacenter applications from a systems perspective. My codesign aims to realize high efficiency and agile evolvability for datacenter infrastructure, and it innovates in two directions: (1) customizing networking stacks based on application needs, and (2) redesigning applications to be network-aware and network-efficient. They have borne fruit for many important datacenter applications, including existing ones (e.g., consensus, distributed transactions) and emerging ones (e.g., far memory over networks, microsecond-scale RPCs). My Electrode [1], Dint [2], and Mew [3] safely inject Paxos, transactions, and RPC load balancing logics into the kernel networking stack respectively via eBPF. This not only achieves remarkable performance improvements (by avoiding kernel overheads) but also allows customizing and evolving the kernel stack based on application needs. My Carbink [4] enables network-aware fault tolerance for far memory with high network and memory efficiency, making it practically usable in datacenters with failures being the norm. Specific to evolvability, my PCAT [5] helps Facebook design an evolvable telemetry system to handle frequent changes in production networks. This focus on **full-stack optimizations** defines my niche as a systems researcher.

CPU efficient distributed protocols with evolvable kernel networking via eBPF. In-memory distributed protocols such as consensus and distributed transactions are important building blocks for datacenter applications. They require intensive network IOs, while the widely-used kernel networking stack gives low IO performance due to high per-IO CPU overhead. Such mismatch has fostered a popular belief that kernel-bypass is the necessary key to high performance for these protocols. However, kernel-bypass is not a panacea: it essentially trades security, isolation, protection, maintainability, and debuggability for performance; it also burns one or more CPU cores for busy-polling even at low loads, which is usually hard to adopt in public cloud deployments due to per-core pricing. As such, I revisit the above popular belief and ask: is the current kernel networking stack really ill-suited for CPU-efficient distributed protocols, especially given many kernel advancements over decades?

With further investigation, I find that the networking stack traversing dominates the kernel overhead. This motivates me to aggressively tailor unnecessary components of the stack for specific distributed protocols, trading slight genericity loss for performance boosts. For example, the reliable transport along with complex queue disciplines, which incurs costly `sk_buff` maintenance and packet copies, could be cut; this is because (1) distributed protocols themselves can recover from packet loss with application-level timeouts, and (2) packet loss happens rarely within today's well-engineered datacenter networks. To do so, I leverage eBPF to *safely* offload protocol-specific request processing logic into the early stages of the kernel stack; this avoids going through the full stack and user space, removing most of the kernel overheads. However, offloading complex distributed protocols into the kernel is challenging, because eBPF has a constrained programming model for kernel safety and liveness. To address this challenge, I strategically partition the distributed protocols to fit frequent critical paths into the kernel for high performance while complex rare paths into the user space for full functionalities. My Electrode [1] project implements such partitioning for the Multi-Paxos protocol, while my Dint [2] implements two transaction protocols (version-based and lock-based), both atop unmodified Linux kernels and achieving remarkable performance boosts. For instance, Dint achieves up to 23$\times$ higher throughput than kernel networking stacks, and 2.6$\times$ higher than a recent DPDK-based kernel-bypass stack Caladan (as the eBPF offloads directly work on raw ethernet packets, bypassing any socket connections). My eBPF offloading work has sparked interest in both industry (e.g., Meta, Intel) and academia (e.g., University of Washington, University of Michigan, NYU).

Network and memory efficient fault-tolerant far memory. In a datacenter, matching a particular application to just enough memory and CPUs is hard. A commodity server tightly couples memory and compute, hosting a fixed number of CPUs and RAM modules that are unlikely to exactly match the computational requirements of any particular application.

Disaggregated datacenter memory is a promising solution. It pairs a CPU with an arbitrary set of possibly-remote RAM modules, with a fast network interconnect keeping access latencies to far memory small. Much of the prior work in this space has a common limitation: a lack of fault tolerance. Unfortunately, in a datacenter containing hundreds of thousands of machines, faults are pervasive. Without fault tolerance, the failure rate of an application using far memory will be much higher than that only uses local memory; the reason is that the use of far memory increases the set of machines whose failure can impact an application.

Achieving both network and memory efficient fault-tolerant far memory is challenging. Conventional memory-efficient fault tolerance scheme applies erasure coding, and stripes a single memory page across multiple remote nodes with RMA-based swapping. For brevity, I use *span* to denote "memory page". Such stripping incurs excessive IO pressure on the networking stack, as each stripe of a span will require an RMA. In Carbink, I tailor the excessive network IOs by eschewing the span-granularity erasure coding, and instead erasure code at the spanset granularity. A spanset consists of multiple spans and their parity data with the same size, and gets swapped out together in a batch. Such batching effectively reduces the number of RMAs per span swap-out, and only requires a single RMA per swap-in, significantly improving network efficiency. However, spanset-granularity erasure coding inevitably incurs memory fragmentation. This is because each span lives in exactly one place (either local memory or far memory), and swapping a span inside a spanset from far memory to local memory creates dead space (and thus fragmentation) in far memory. To address this problem, I design a pauseless defragmentation mechanism running off the swapping critical path, asynchronously reclaiming dead space for later swap-outs in the background. Carbink is implemented and evaluated atop Google's datacenter infrastructure. Compared to a state-of-the-art fault-tolerant design that uses span-granularity erasure coding, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher far memory usage (due to asynchronous memory defragmentation). Carbink also results in a joint patent with Google.

CPU efficient load balancing for microsecond-scale RPCs. My Mew [3] project studied how to efficiently load balance microsecond-scale RPCs among multiple servers, under the emerging microservice architectures. Conventional Power-of-Two load balancing probes servers' load too often (i.e., probing before each RPC) and hurts application throughput. On the other hand, probing too infrequently will result in stale estimates of load, causing suboptimal load balancing and violated SLOs. To break this dilemma, Mew [3] tailors unnecessary load probings to just fulfill the staleness requirement that does not degrade tail latency; it performs probing statistically following an optimal probing frequency, obtained by running a gradient descent algorithm on the probing frequency vs. tail latency space. To efficiently implement load probing, especially for kernel networking stacks with high overhead, Mew leverages eBPF to directly return load signal values in the kernel, without going through the full kernel stack or user space.

Evolvable and memory efficient network telemetry. Modern datacenter networks are complex; thus operators must rely on network telemetry systems for monitoring, alerting, troubleshooting, etc. However, changes happen frequently in production networks (e.g., modifications to monitoring intent, advances of device APIs), impacting the reliability of network telemetry systems. To handle various changes, I helped Facebook develop their evolvable network telemetry system PCAT [5]. PCAT proposes to use a change cube abstraction to systematically track changes, and an intent-based layering design to confine and track changes. Network telemetry also requires high efficiency for memory. My Cold Filter [6], Elastic Sketch [7], and more [8, 9] design memory-efficient probabilistic data structures that can be updated at line rate, have low memory footprints, and high accuracy. Elastic Sketch has been cited over 400 times by follow-up work across many academic research groups (e.g., CMU, Princeton, University of Pennsylvania, Technion, KTH).

Secure hardware architecture for SmartNICs. Cloud providers are deploying various SmartNICs with wimpy-yet-power-efficient RISC cores to offload simple network functions such as network virtualization and traffic scheduling. Unfortunately, vast cloud tenants are barred from the efficiency benefits of SmartNICs, because they are not allowed to run their own customized functions on SmartNICs. The root cause is that modern SmartNICs provide little isolation between the network functions belonging to different tenants; these NICs also do not protect network functions from the datacenter-provided management OS running on the NIC. My S-NIC [10] project proposes minimal changes to SmartNIC hardware, so that datacenters can provide offloaded functions with strong isolation, while preserving most of the total-cost-of-ownership benefits with minimal performance degradations.

Future Research

Building on my past experiences in networking, memory management, OS kernels, and datacenter applications, I am excited to apply my full-stack optimization approach with cross-layer codesign to the following problems.

Resource efficient machine learning. Machine learning (ML) workloads such as the training and inference of Large Language Models (LLMs) are extremely resource-hungry, requiring expensive accelerators like GPUs. I intend to take a full-stack approach to improve the resource efficiency of ML workloads, covering GPU memory efficiency and compute efficiency. One direction is applying far memory techniques to LLM training and inference by swapping to

CPU memory. For performance, I plan to codesign far memory swapping with the memory access patterns of LLM weights and key-value cache, e.g., different access frequencies for different weights due to the attention mechanism in LLMs. Another direction is developing a unified GPU memory abstraction that allows easily accessing remote GPU memory over high-speed networks such as NVLink; this kind of GPU memory pooling would help reduce memory stranding and fragmentation caused by dynamic memory allocations in ML workloads. For performance, I plan to codesign such memory pooling with ML workload characteristics, e.g., allowing relaxed consistency. Finally, I am interested in fine-grained GPU kernel scheduling at the microsecond scale possibly with preemption; the goal is to efficiently multiplex GPU compute resources among multiple jobs without losing performance.

Datacenter-scale distributed runtime. A long-term goal of my research is to build a datacenter-scale distributed runtime to not only simplify application development but also increase the whole datacenter efficiency and evolvability. This distributed runtime sits between applications and datacenter resources: (1) for applications, it provides generic and stable interfaces to use compute, memory, storage, and accelerators, and customizable fault tolerance and recovery schemes based on application needs; (2) for resources, it eschews the conventional reservation-based provisioning strategy, and instead provisions resources in a best-effort manner to achieve high resource efficiency.

Today's datacenters have already provisioned network resources in a best-effort manner, and I plan to expand this strategy to cover more resources like compute, memory, storage, and accelerators. For these new best-effort resources, many networking techniques like congestion control can be applied to enable efficient fair sharing. However, unlike the network resources that are delay-tolerable for applications, other resources especially the memory are not (think of out-of-memory errors). To address this challenge, I intend to leverage techniques like far memory to create a delay-tolerable memory abstraction, at the cost of lower resource utility than normal memory.

Teaching

I greatly enjoy the rewards of teaching and mentoring students. For me, the rewards consist of two parts: (1) the pride and fulfillment when I help students carry out their studies smoothly and when my students grow into independent researchers, and (2) the interesting future research directions inspired or confirmed during teaching and mentoring.

Driven by these rewards, I have taught as a teaching assistant for a system course (Harvard CS145 Networking at Scale) and as a small-group "supervisor" for an algorithm course (Algorithm Design and Analysis at Peking University), and mentored four undergraduates and five junior PhD students in their research. My teaching philosophy has been on three aspects: (1) *Building safe and inclusive environments.* This is important to make students with different prior knowledge feel they are welcome to ask both simple questions and challenging ones. (2) *Focusing on hands-on experiences.* I believe the best way to learn computer systems is through reading, running, debugging, and hacking well-written codebases in a hands-on manner. (3) *Promoting critical thinking on the pros and cons of techniques.* I would like to encourage students to critically think about new techniques around us, such as the emerging LLMs.

Based on my research background, I am qualified to teach undergraduate courses of computer networks, operating systems, distributed systems, and algorithms and data structures, and two advanced graduate courses:

- *Data center networking:* I will discuss how modern data centers design and build high-performance network fabrics including topology, routing, congestion control, fault tolerance, load balancing, etc.
- *Dataplane operating systems:* I will discuss how the OS evolves to keep up with the fast hardware in data centers, including user-space networking, efficient threading, light-weight isolation, etc.

References

[1] **Yang Zhou**, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of USENIX NSDI*, 2023.

[2] **Yang Zhou**, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel Distributed Transactions with eBPF. Under submission.

[3] **Yang Zhou**, Hassan Wassel, James Mickens, Minlan Yu, and Amin Vahdat. Mew: Efficient Inter-Server Load Balancing for Microsecond-Scale RPCs. Under submission.

[4] **Yang Zhou**, Hassan Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David Culler, Hank Levy, and Amin Vahdat. Carbink: Fault-Tolerant Far Memory. In *Proceedings of USENIX OSDI*, 2022.

[5] **Yang Zhou**, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong. Evolvable Network Telemetry at Facebook. In *Proceedings of USENIX NSDI*, 2022.

[6] **Yang Zhou**, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold Filter: A Meta-Framework for Faster and More Accurate Stream Processing. In *Proceedings of ACM SIGMOD*, 2018.

[7] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, **Yang Zhou**, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of ACM SIGCOMM*, 2018.

[8] Tong Yang, **Yang Zhou**, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams. *Proceedings of the VLDB Endowment*, 2017.

[9] **Yang Zhou**, Omid Alipourfard, Minlan Yu, and Tong Yang. Accelerating Network Measurement in Software. *ACM SIGCOMM Computer Communication Review*, 2018.

[10] **Yang Zhou**, Mark Wilkening, James Mickens, and Minlan Yu. SmartNIC Security Isolation in the Cloud with S-NIC. Under submission.