Computer networks are integral to society, whether in the form of cellular links, Wi-Fi links, or datacenter networks. A critical component of these networks is the infrastructure of base stations, access points, and routers that connects computers together. Traditionally, networks emphasized an architecture with smart computers and dumb infrastructure, where routers focused on distributed routing protocols and packet forwarding. Over time, however, routers have taken on an ever-changing list of requirements including access control, load balancing, packet scheduling, congestion control, and measurement. To address these important needs, router programmability has become as important as performance. However, despite efforts over the past two decades, programmable routers remain 10–100x slower than fixed-function routers. They are rarely, if ever, used in production networks.

My research focuses on the design and implementation of high-speed programmable routers with no compromise in performance. I address the programmability-performance tension by focusing on important classes of router tasks requiring programmability. I then design both the bespoke programmable hardware that provides high performance and the software infrastructure of languages and compilers to program this hardware. Because my research straddles both router hardware and software, I seek to combine my expertise in computer networks and software systems with the “borrowed” expertise of collaborators in hardware design, languages, and compilers.

My projects illustrate this multidisciplinary approach.

1. **Domino** provides both a low-level instruction set and a high-level programming model for packet-processing algorithms. For the first time, Domino shows how several load balancing, congestion control, and queue management algorithms can be written in a high-level imperative language and compiled to run on a high-speed programmable router, with no loss in performance relative to the fastest fixed-function routers. Domino’s programming model has been adopted by P4, an industry effort towards a standard language for programming network devices.

2. A **Push In First Out queue (PIFO)** is the first programming abstraction for packet scheduling (i.e., deciding which packet from a router’s buffer to transmit next) that can be implemented efficiently in hardware. The PIFO abstraction is simple: a priority queue of packets with a small program to assign each packet its priority. Yet, by flexibly programming a packet’s priority assignment, a network operator can use the PIFO abstraction to program a variety of previously proposed scheduling algorithms.

3. **Performance Queries** are a new abstraction to measure network performance on routers. We are developing both a SQL-like language for querying network performance (e.g., average queueing latency or average packet drop rate on a particular switch) and efficient router hardware to execute these queries at the router’s line rate.

**Programmable Routers**

**Domino.** Domino shows how to program data-plane algorithms on high-speed routers. Data-plane algorithms modify packets and persistent router state as packets transit a router; they include algorithms for in-network congestion control, active queue management, load balancing, network security, and network measurement. High-speed data-plane programming poses two challenges: (1) what hardware instructions are required to support programmable state modification at the router’s line rate and (2) what is the right programming model?

In addressing each challenge, Domino makes two new contributions.

1. **Atoms** capture a router’s instruction set. They specify atomic units of packet processing provided by the router hardware, e.g., an atomic counter or an atomic test-and-set. Atoms are atomic in that all state updated by an atom is visible to the next packet arriving at the atom in the next clock cycle.

2. **Packet transactions** provide a programming model. A packet transaction is an atomic and isolated block of code capturing an algorithm’s logic. It provides the semantics that any visible state is equivalent to a serial execution of packet transactions in the order of packet arrival—akin to an infinitely fast single-threaded router. Packet transactions are expressive and capture many important data-plane
algorithms. Further, their serial semantics shield programmers from the hardware’s parallelism, instead enlisting a compiler to transform a serial packet transaction to a parallel atom pipeline.

P4 [2], an emerging packet-processing language, has adopted packet transactions. P4 programmers can now use an \texttt{atomic} annotation around a block of statements to specify that the block must execute atomically. I am currently working with the P4 design group to implement algorithms from Domino’s compiler in the P4 reference compiler.

**Push In First Out Queues.** Packet scheduling is an important determinant of network performance. The choice of scheduling algorithm is tied to a network’s overarching goals. For instance, an algorithm that divides link capacity fairly is ideal in a multi-tenant setting, while the shortest remaining processing time algorithm is ideal for a single tenant who desires low flow completion time. Today’s routers provide a fixed set of scheduling algorithms and do not allow an operator to \textit{program} scheduling to suit their needs.

Routers lack programmable scheduling because there is no single abstraction to express many scheduling algorithms. \textit{Push In First Out Queues (PIFOs)} provide such an abstraction. They exploit the fact that in many practical schedulers, the relative order of packets that are already buffered does not change in response to new packet arrivals. Put differently, when a packet arrives, it can be pushed into the right location based on a packet priority (push in), but packets can always be dequeued from the head (first out). A single PIFO expresses many schedulers, e.g., token bucket shaping, weighted fair queueing, and strict priority scheduling. Further, PIFOs can be combined to express hierarchical schedulers. PIFOs are feasible in hardware: a hardware design for a programmable 5-level hierarchical scheduler costs less than 4% additional chip area.

**Performance Queries.** An ongoing project provides a query language and supporting router hardware for network performance measurement queries. As examples, an operator could ask for flows with a high degree of packet reordering or a moving average of packet latencies for each flow. Our query language is modeled after SQL, but allows order-dependent aggregation (e.g., an exponentially weighted moving average over packet latencies), while SQL only supports order-independent aggregates like counts and averages. The router hardware to support these queries is a programmable key-value store in the router’s ASIC. The keys represent flows, while the values store and programmatically update per-flow state on every packet.

**Other Projects**

Besides working on programmable routers, I have taken on several projects in networking, described below.

**Sprout** [5] is a congestion-control protocol designed for low-latency video conferencing on cellular networks. These networks are characterized by variable link rates and very low loss rates due to link-layer reliability. Hence, if a transmitter doesn’t match its sending rate to the current link rate, it can build up second-long standing queues. Sprout tackles this problem by modeling the cellular link’s capacity as a random walk. It then integrates the link capacity forward in time to determine a forecast of the number of bytes that it can send, bounding the risk of the latency exceeding 100 ms to 5%. Sprout outperformed existing protocols by between 2 and 4× on throughput and 7 and 9× on delay. It won the Internet Research Task Force’s Applied Networking Research Prize in 2014.

We empirically investigated the \textit{learnability} of congestion-control algorithms [6] by using the Remy protocol-design tool [7] to automatically synthesize congestion-control algorithms given an objective function, a model of the network, and its workload. We used Remy to learn a congestion-control algorithm given a set of training networks, but tested it on a set of testing networks, which differed from the training network in one or more parameters. We found that it was important to accurately model some network features during training, such as the degree of multiplexing, to perform well in the testing network. On the other hand, features like the network topology didn’t have to be modeled accurately.

We developed \textbf{Mahimahi} [8], a suite of tools to record HTTP resources during a page load and then replay the page load over an emulated network. We used Mahimahi to reproducibly measure the page-load times of several Web protocols including HTTP, SPDY, and QUIC. Mahimahi is open source, available as a Debian package, and used in Stanford’s Network Application Studio (CS 344G) and MIT’s graduate...
networking course. Informed by our Mahimahi measurements, we built Cumulus [8], a cloud-based browser that improves page-load times relative to existing Web protocols by between 19% and 80%.

Future Directions

Programmable Networks

An ideal programmable network would be as programmable as CPUs, GPUs, and DSPs today. The network should present no barrier to innovation: programming a router should be as routine as using the sockets API on an end host. Realizing this vision requires us to solve several problems at the intersection of networking, programming languages, and hardware design.

**Designing Router Instruction Sets.** Can we formalize and automate instruction set design? How do we pack in the “maximum programmability” within an atom’s single clock cycle timing budget? At the same time, can we minimize the area of the atom’s digital circuit? What set of algorithms can this atom support? Is this atom programmable, i.e., will it generalize to unanticipated use cases?

**The Middle Plane for Networking.** There will always be algorithms that cannot run in the data plane because they cannot sustain packet processing at the router’s line rate. Today, such algorithms can only run on the much slower control plane, leading to a performance cliff once an algorithm crosses a threshold of complexity. Can we develop a programmable “middle plane” that sits between the control and data planes? This middle plane would provide greater programmability than the data plane but at lower performance. It could be used to run algorithms either on a sample of packets across all ports or at full line rate on a few ports. For instance, an operator might require programmable measurement on a sample of all packets or line-rate scheduling support only on the congested ports.

Hardware Architectures for Networked Systems

Moving beyond routers, I believe that we must rethink hardware architectures for entire networked systems consisting of both end hosts and routers. Two technology trends are motivating: (1) memory disaggregation, where memory in clusters is moving out of servers and into a global cluster-wide memory pool and (2) the slowing down of transistor scaling.

**Memory Disaggregation.** An emerging architecture for datacenters is a cluster with separate units of memory, storage, and compute interconnected over a common network fabric. This architecture presents the entire cluster as one large computer, allowing the cluster’s memory to be treated as a large pool that can be shared dynamically across its computers. In a disaggregated world, memory access—traditionally an architect’s problem—becomes a networking problem. This will require us to rethink the design of routers and network interface cards to support these new low-latency remote memory accesses.

**The End of Moore’s Law.** The end of transistor scaling has lead to application-specific hardware accelerators for both energy and performance reasons. Without the automatic benefits of increasing clock frequency or core counts, application designers will have to eke out as much performance as they can from a limited transistor budget. As a concrete example, high-speed networking is increasingly straining CPUs on end hosts and middleboxes, requiring FPGAs [9] and scale-out clusters [10] to handle the increased load. Can we profile repeatedly used and relatively mature networking motifs that target x86 chips today and measure their “x86 tax” relative to a pure silicon implementation? Such motifs (e.g., SSL encryption and data compression) can be hardened as accelerators in silicon, providing energy and performance benefits over FPGAs or scale-out clusters.
Network Performance Verification

Objective-driven protocol design as embodied in Sprout and Cumulus has shown how knowing an application’s objective (e.g., bounded risk of delay or page-load time) can inspire better protocols. Can we flip this around? Given a precise specification for a network protocol, its workload, and topology, can we formally verify that it satisfies a service level agreement such as guaranteeing that the 95th percentile RPC latency is under a specific target? While considerable work has gone into formal verification of boolean properties such as reachability, verification of quantitative properties like latency is still wide open.

References


