McNettle: High Performance Centralized Network Control in a High-Level Language

Yale

Andreas Voellmy Junchang Wang
Yale University University of Science and Technology of China

Motivation

A centralized SDN controller provides many benefits, but may become a *scaling bottleneck*.

A data center controller may need to provide control for [Tavakoli]: 100s to 1000s of switches, 2 million virtual machines (hosts), and 20 million new TCP sessions per second, but existing sequential controllers cannot meet this performance.

Scaling controller through parallel and concurrent programming may substantially complicate controller programming.

McNettle: Scalable Functional Network Control on Multicores

Hardware: Single-node multicore NUMA servers with modern multiqueue NICs.

Software: *McNettle API* allows users to write control algorithms in Haskell, a high-level functional, garbage-collected programming language. *McNettle Engine* executes on single-node multicore NUMA servers, automatically scaling performance through 40 cores.

Programming in McNettle

Each switch is controlled by a user-specified *Switch Controller* accepting messages from the switch and generating messages to switches.

Highly consistent global state supported through shared memory:

- Nonblocking data structures for high performance.
- Software Transactional Memory (STM) for complex shared state manipulation.

Multi-switch state setup supported by allowing one switch controller to setup state in another switch.

Example 1: Fine-grained Flow Management with Global State

```
main = do server ← startServer params
          tbl ← newHostLocTable
          forever (do (features, switch) ← acceptSwitch server
                      startSwitch switch (switchProg tbl))

switchProg tbl msg =
  case msg of
    PacketIn pkt →
      do mport ← lookupAndLearn tbl pkt
         case mport of
           Nothing → forward pkt flood
           Just port → forwardWithExactRule pkt (phyPort port)
    _ → return ()
```

Example 2: Bandwidth Reservations with STM

```
atomically (reservePath capTable amt path)
```

where

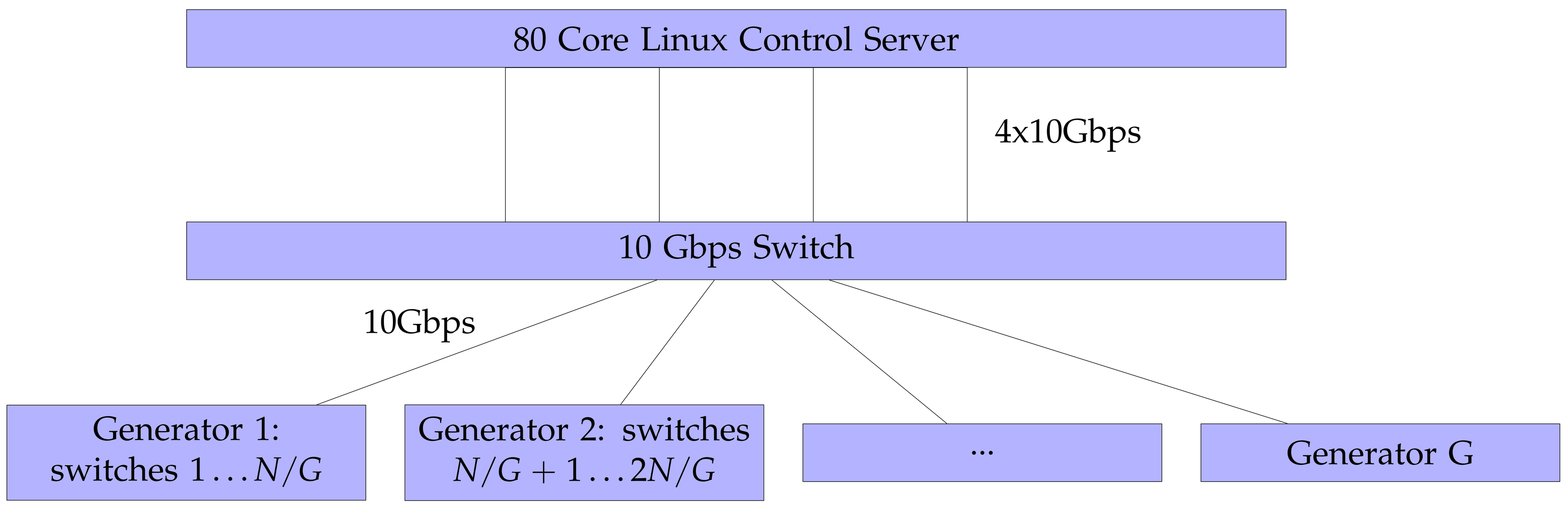
```
reservePath capTable amt path
  = forM path (λlink → reserveLink capTable amt link)
reserveLink capTable amt link =
  do current ← linkCapacity capTable link
     let remaining = current − amt
     when (remaining < 0) retry
     updateCapacity capTable link remaining
```

Example 3: Parallel Dijkstra Routing

Compute routing for each switch in parallel:

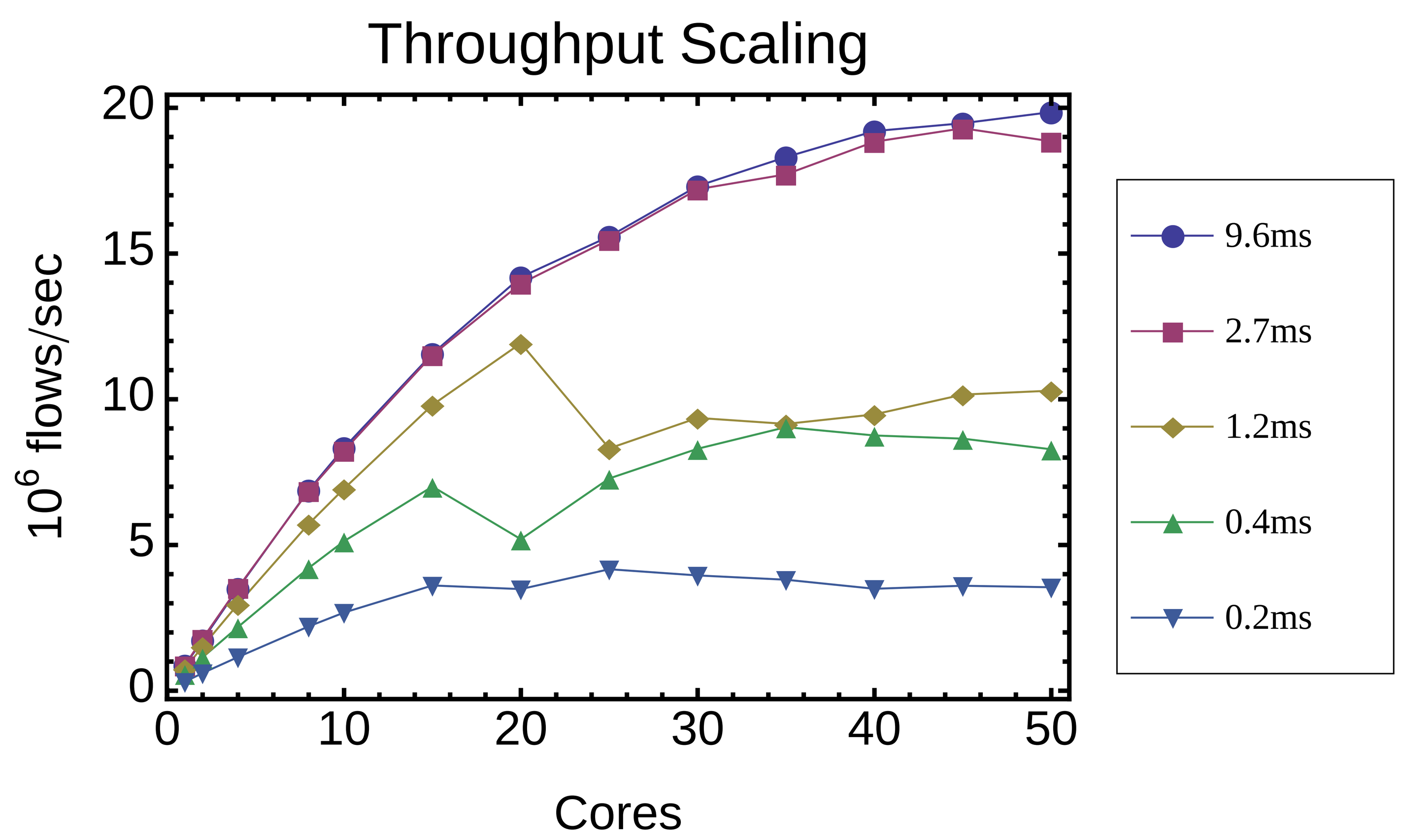
```
parMap (dijkstraRouting topology) switches
```

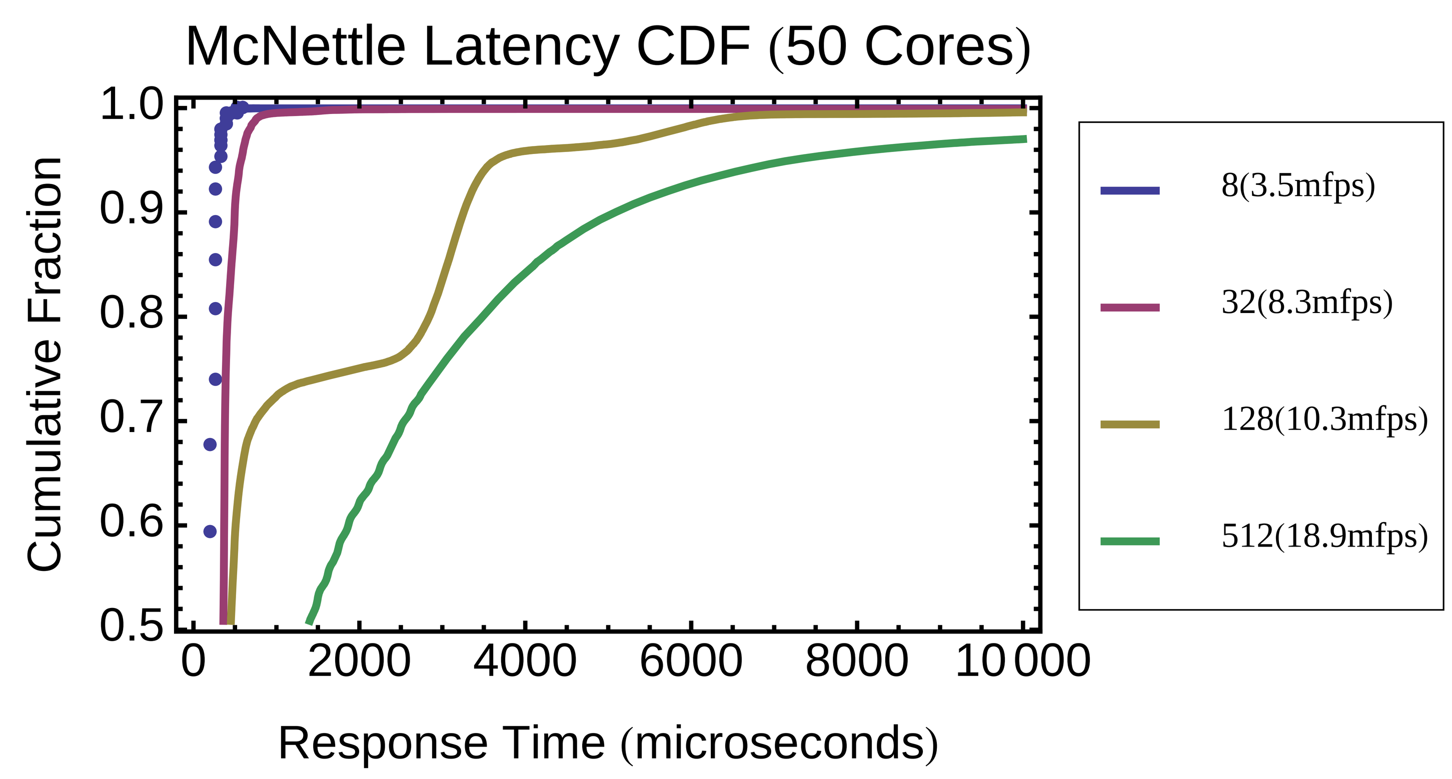
Evaluation Setup

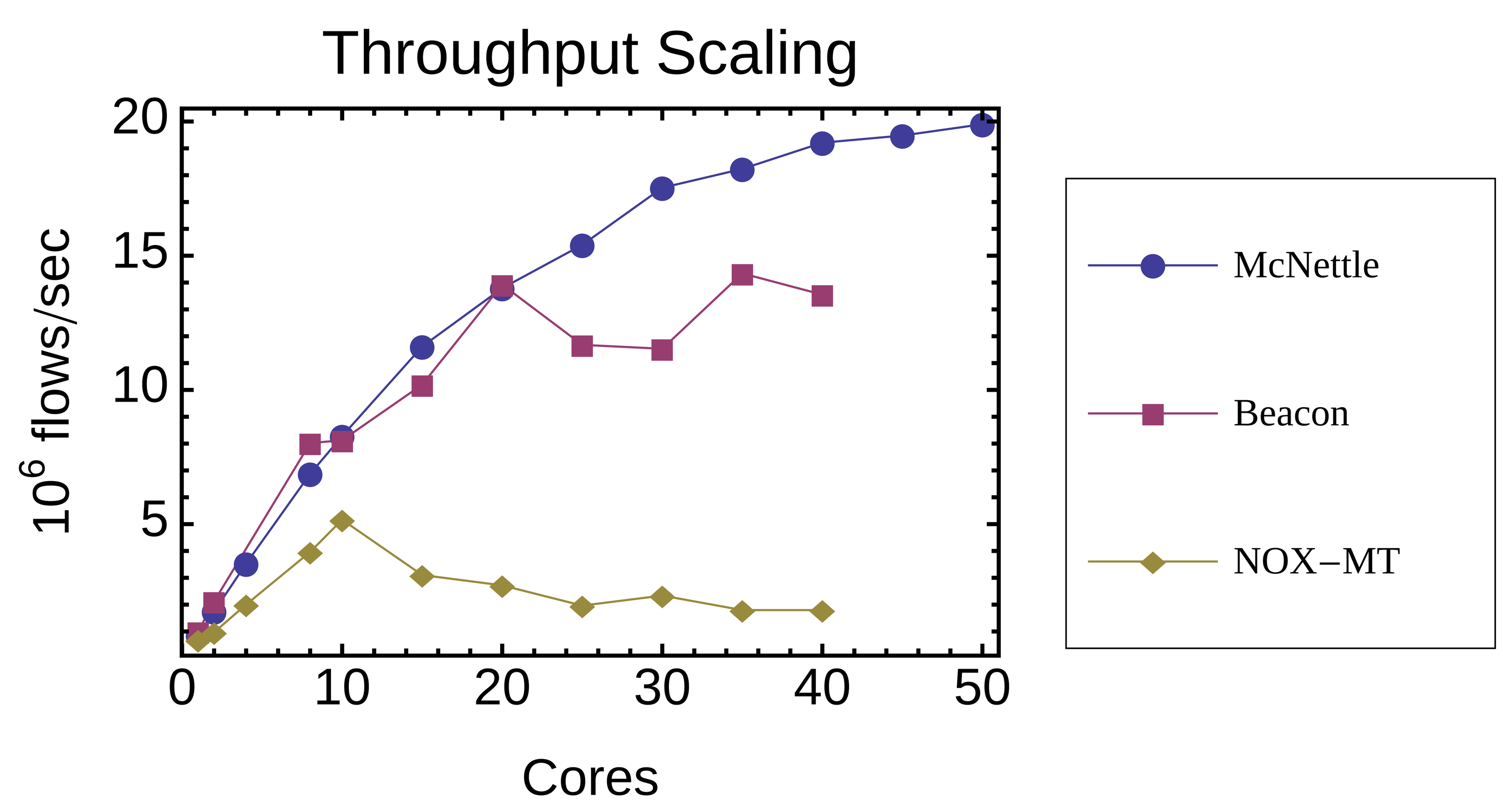


Generators simulate switches with attached hosts; simulates packet-in messages that would be sent if every host starts a TCP session to every other host.

Throughput & Latency







Implementation: Key Points

- Utilize Glasgow Haskell Compiler's (GHC) multicore runtime system, which implements scalable, work-balancing thread scheduler.
- Use batching to minimize system call overhead.
- Custom memory management of critical message buffers to avoid garbage collector & storage allocator overhead.
- Reader-writer queues to efficiently support cross-switch messaging.
- Improve parallelization of GHC garbage collector to reduce pause times.
- Parallelize GHC's IO event loop to reduce response time.
- Separate cores for NIC interrupt handling and McNettle threads.

andreas.voellmy@yale.edu