High Performance Network Stack

ECE/CS598HPN

Radhika Mittal
Tx Processing in the kernel

- NIC hardware
- DMA
- tx_ring
- qdisc
- qdisc_run
- qdisc_restcut
- net_tx_action
- completion queue
- sntr
- send_msg
- IP csum
- IP route
- IP filter
- dev_xmit
- softirq to free
- kernel send buffer
- Application
- write
- TCP process
- kernel
- user
- Memory
Rx Processing in the kernel

NIC hardware → rx_ring → softirq → IP firewall
interrupt IP routing
scheduled

tcp_v4_recv → socket backlog
recv
recv_backlog

TCP process → kernel recv buffer
read

Application → user

Memory

Rx Processing in the kernel

NIC hardware → DMA → rx_ring → softirq → IP firewall IP routing → tcp_v4_rcv → socket backlog → recv_backlog → recv.recv_buffer → TCP process → Application

device driver kernel user
What are some sources of performance overheads?
MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han, Scott Marshal, Byung-Gon Chun, Sylvia Ratnasamy

OSDI’12

Content borrowed from Sangjin’s OSDI talk
Two Types of Network Workloads

• **Bulk Transfer**
  - Large files (HDFS)

• **Message-oriented**
  - Short connections or small messages (HTTP, RPCs, DB, key-value stores, etc)
Two Types of Network Workloads

- **Bulk Transfer**
  - Large files (HDFS)
  - A half CPU core can saturate 10Gbps link

- **Message-oriented**
  - Short connections or small messages (HTTP, RPCs, DB, key-value stores, etc)
  - CPU-intensive
BSD Socket API Performance Issues

```c
n_events = epoll_wait(...); // wait for I/O readiness
for (...) {
    ...
    new_fd = accept(listen_fd); // new connection
    ...
    bytes = recv(fd2, buf, 4096); // new data for fd2
}
```

- Issues with message-oriented workloads
  - System call overhead
BSD Socket API Performance Issues

```c
n_events = epoll_wait(...);  // wait for I/O readiness
for (...) {
    ...
    new_fd = accept(listen_fd); // new connection
    ...
    bytes = recv(fd2, buf, 4096); // new data for fd2
```
BSD Socket API Performance Issues

```c
n_events = epoll_wait(...);  // wait for I/O readiness
for (...) {
    ...
    new_fd = accept(listen_fd);  // new connection
    ...
    bytes = recv(fd2, buf, 4096);  // new data for fd2
}
```

- Issues with message-oriented workloads
  - System call overhead
  - Shared listening socket
  - File abstraction overhead
Microbenchmarks: how bad?

RPC-like test on an 8-core Linux server (with epoll)

768 Clients

Server

new TCP connection

request (64B)

response (64B)

Teardown

10 transactions

1. Message size

2. Connection length

3. Number of cores
Microbenchmarks: how bad?

1. Small Messages Are Bad

- Throughput
- CPU Usage

Low throughput  Message Size (B)  High overhead
Microbenchmarks: how bad?

2. Short Connections Are Bad

Throughput (1M transactions/s)

19x lower

Number of Transactions per Connection
Microbenchmarks: how bad?

3. Multi-Core Will Not Help (Much)
MegaPipe Design

Focus: low-overhead and multi-core scalability.
MegaPipe: Overview

Problem | Cause | Solution
---|---|---
Low per-core performance |  |  
Poor multi-core scalability |  |  
Key Primitives

- Handle
  - Similar to file descriptor
    - But only valid within a channel
  - TCP connection, pipe, disk file, ...

- Channel
  - Per-core, bidirectional pipe between user and kernel
  - Multiplexes I/O operations of its handles
How channels help?

User

Handles →

Channel →

I/O Batching

Kernel
I. I/O Batching

- Transparent batching
  - Exploits parallelism of independent handles

![Diagram showing I/O Batching process](image-url)
How channels help?
How channels help?

Core 1

Core 2

Core 3

Listening socket partitioning

New connections
2. Listening Socket Partitioning

- Per-core accept queue for each channel
  - Instead of the globally shared accept queue
2. Listening Socket Partitioning

- Per-core accept queue for each channel
  - Instead of the globally shared accept queue

```
mp_register()
```

![Diagram showing listening socket partitioning](image)
2. Listening Socket Partitioning

- Per-core accept queue for each channel
  - Instead of the globally shared accept queue

```
mp_accept()
```

```
Listening socket
```

```
Kernel
```

```
User
```

```
New connections
```

```
Accept queue
```

```
Accept queue
```

```
Accept queue
```
How channels help?
How channels help?
3. Light-weight Sockets

- Common-case optimization for sockets
  - Sockets are ephemeral and rarely shared
    - Bypass the VFS layer
    - Convert into a regular file descriptor only when necessary
Evaluation: Microbenchmarks

- Throughput improvement with various message sizes
Evaluation: Microbenchmarks

- Multi-core scalability
  - with various connection lengths (# of transactions)
Evaluation: Macrobenchmarks

- memcached
  - In-memory key-value store
  - Limited scalability
    - Object store is shared by all cores with a global lock

- nginx
  - Web server
  - Highly scalable
    - Nothing is shared by cores, except for the listening socket
Evaluation: Macrobenchmarks

- memcached
  - In-memory key-value store
  - Limited scalability
    - Object store is shared by all cores with a global lock

- nginx
  - Web server
  - Highly scalable
    - Nothing is shared by cores, except for the listening socket
Evaluation: memcached

The chart shows the throughput (1k requests/s) against the number of requests per connection. The green line represents MegaPipe, and the purple dotted line represents the baseline. The chart highlights that MegaPipe has better performance compared to the baseline, with significant improvements at various request rates:

- At 1 request per connection, MegaPipe is 3.6x faster than the baseline.
- At 2 requests per connection, MegaPipe is 3.9x faster.
- At 3 requests per connection, MegaPipe is 2.4x faster.
- At 4 requests per connection, MegaPipe is 1.3x faster.

A red arrow indicates a global lock bottleneck.
Evaluation: memcached

The graph shows the throughput (1k requests/s) on the y-axis against the number of requests per connection on the x-axis. The lines represent different configurations:
- **MegaPipe-FL**
- **Baseline-FL**
- **MegaPipe**
- **Baseline**

The throughput increases with the number of requests per connection for all configurations. The MegaPipe-FL line shows the highest throughput across all connection sizes.
Evaluation: nginx
Conclusion

- Short connections or small messages:
  - High CPU overhead
  - Poorly scaling with multi-core CPUs

- MegaPipe
  - Key abstraction: per-core channel
  - Enabling three optimization opportunities:
    - Batching, partitioning, lwsocket
  - 15+% improvement for memcached, 75% for nginx
Your thoughts?

• What did you like about the paper?

• What are some of its limitations?

• What other sources of performance overhead remain?