MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han

in collaboration with

Scott Marshall

Byung-Gon Chun

Sylvia Ratnasamy

University of California, Berkeley *Yahoo! Research*

tl;dr?

MegaPipe is a new network programming API

for message-oriented workloads

to avoid the performance issues of BSD Socket API

Two Types of Network Workloads

1. Bulk-transfer workload
   - One way, large data transfer
     - Ex: video streaming, HDFS
   - Very cheap
     - A half CPU core is enough to saturate a 10G link

Two Types of Network Workloads

1. Bulk-transfer workload
   - One way, large data transfer
     - Ex: video streaming, HDFS
   - Very cheap
     - A half CPU core is enough to saturate a 10G link

2. Message-oriented workload
   - Short connections or small messages
     - Ex: HTTP, RPC, DB, key-value stores, …
   - CPU-intensive!

BSD Socket API Performance Issues

```
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

```
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

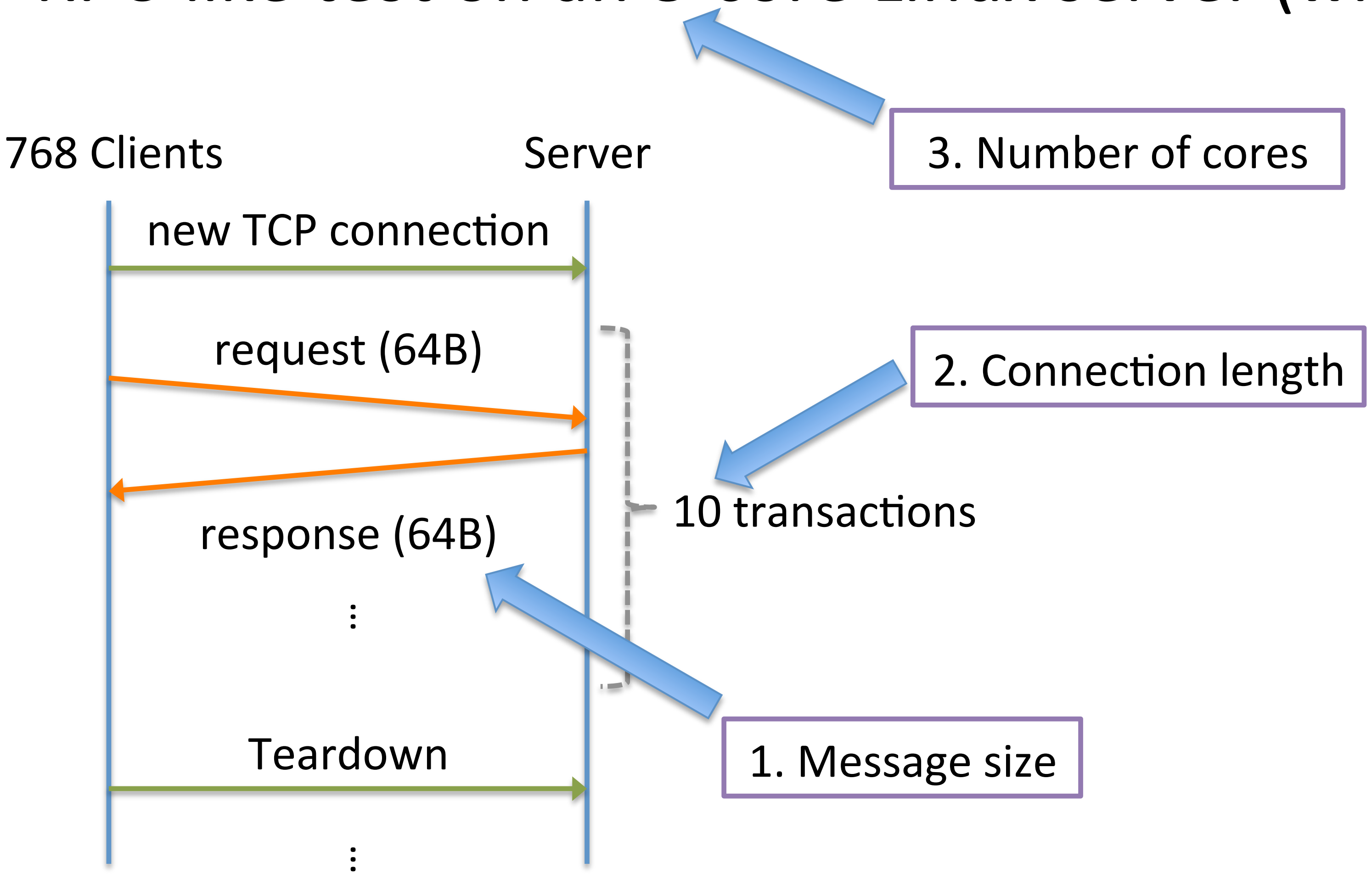
BSD Socket API Performance Issues

```
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
```

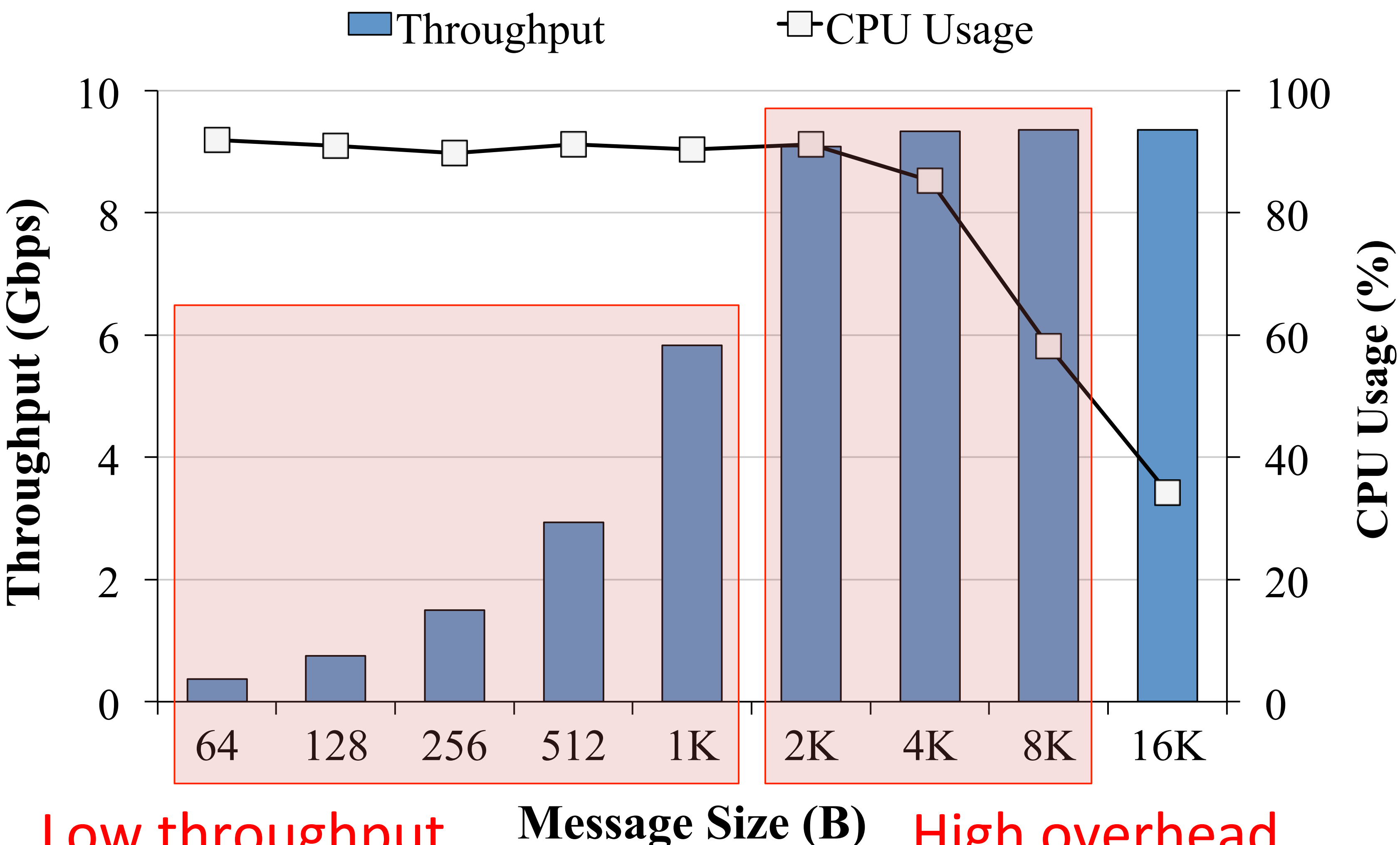
- Issues with message-oriented workloads
  - System call overhead
  - Shared listening socket
  - File abstraction overhead

Microbenchmark: How Bad?

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



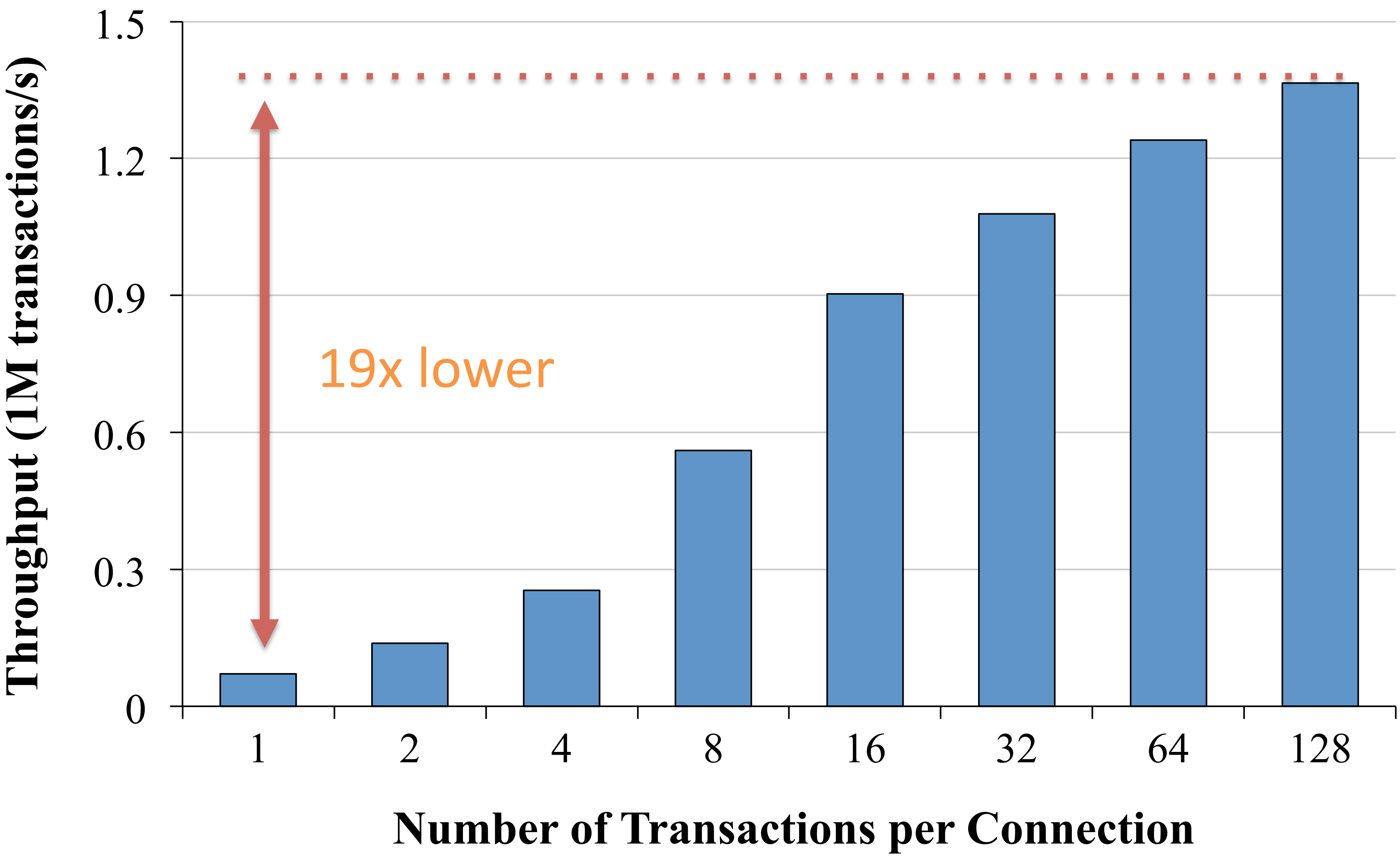
1. Small Messages Are Bad



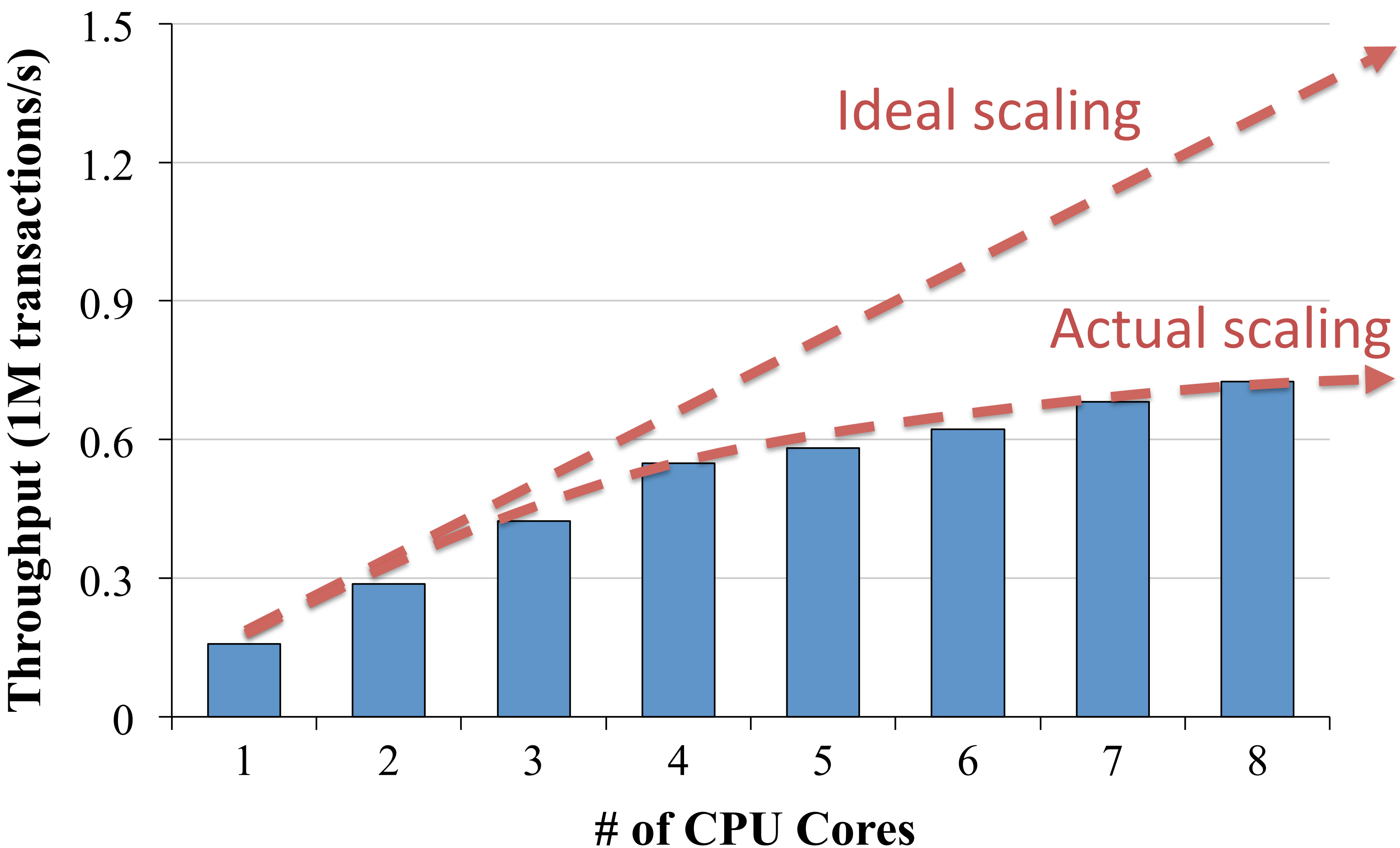
Low throughput

High overhead

2. Short Connections Are Bad



3. Multi-Core Will Not Help (Much)

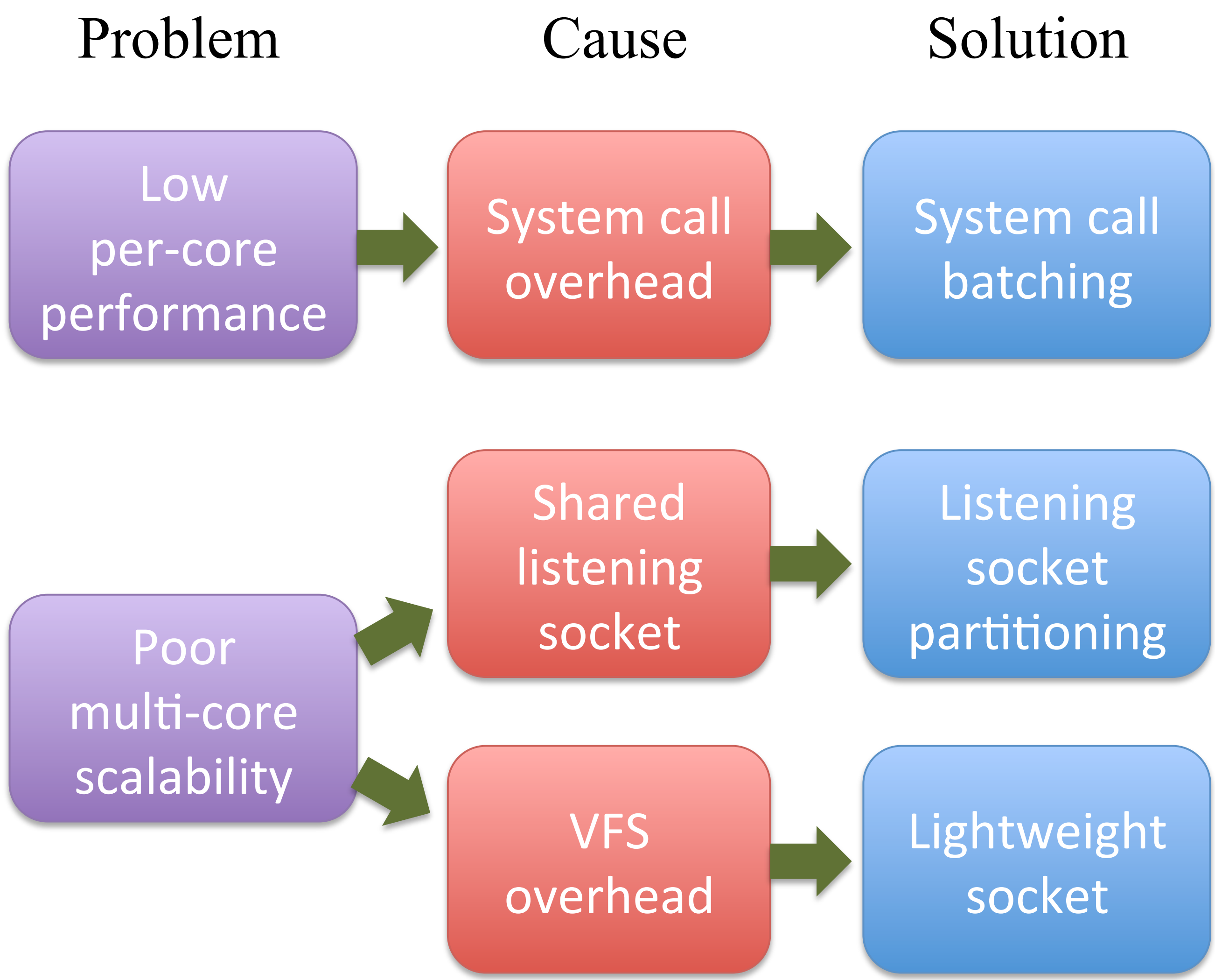


MEGAPIPE DESIGN

Design Goals

- Concurrency as a first-class citizen
- Unified interface for various I/O types
  - Network connections, disk files, pipes, signals, etc.
- Low overhead & multi-core scalability
  - Main focus of this presentation

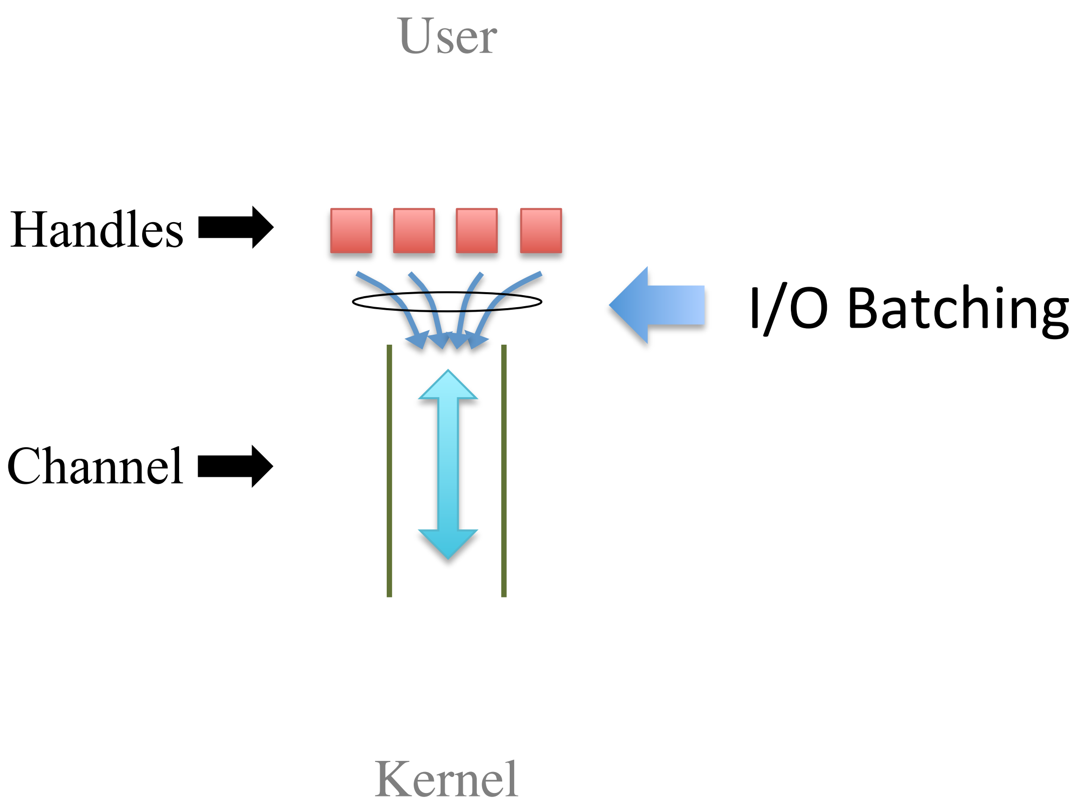
Overview


Problem
Cause
Solution
Low per-core performance
System call overhead
System call batching
Poor multi-core scalability
Shared listening socket
Listening socket partitioning
VFS overhead
Lightweight socket

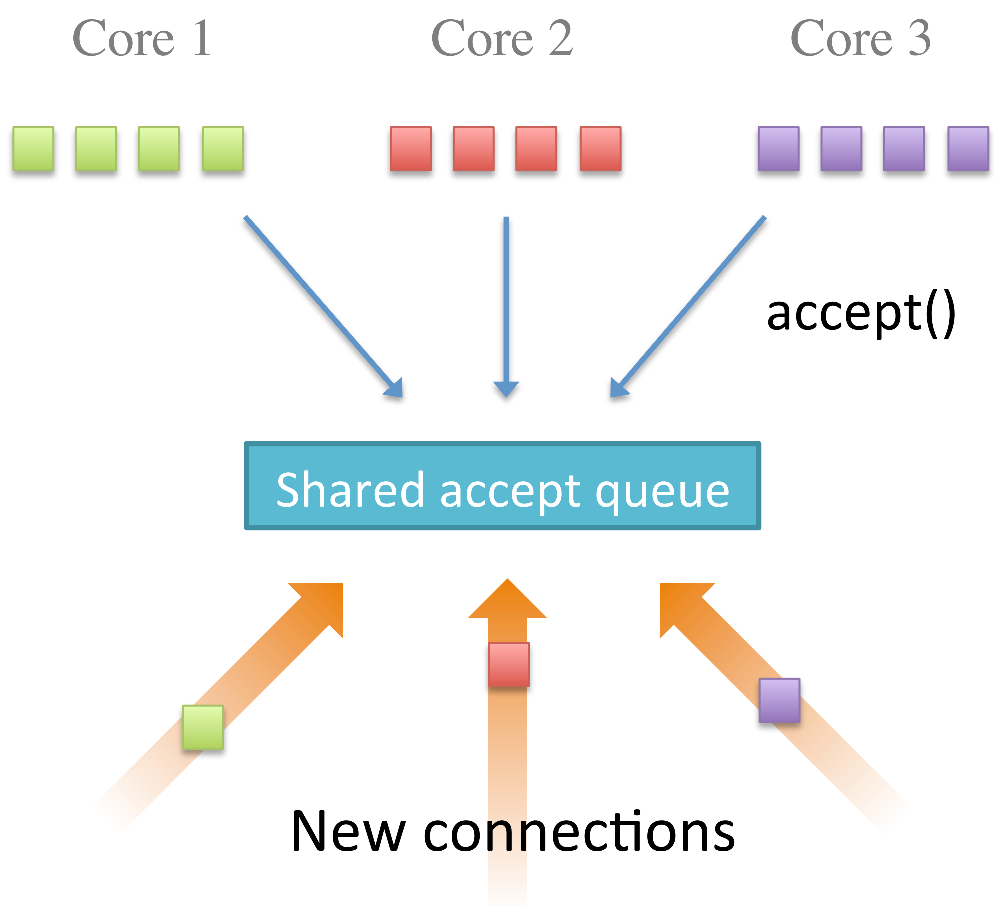
Key Primitives

- 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

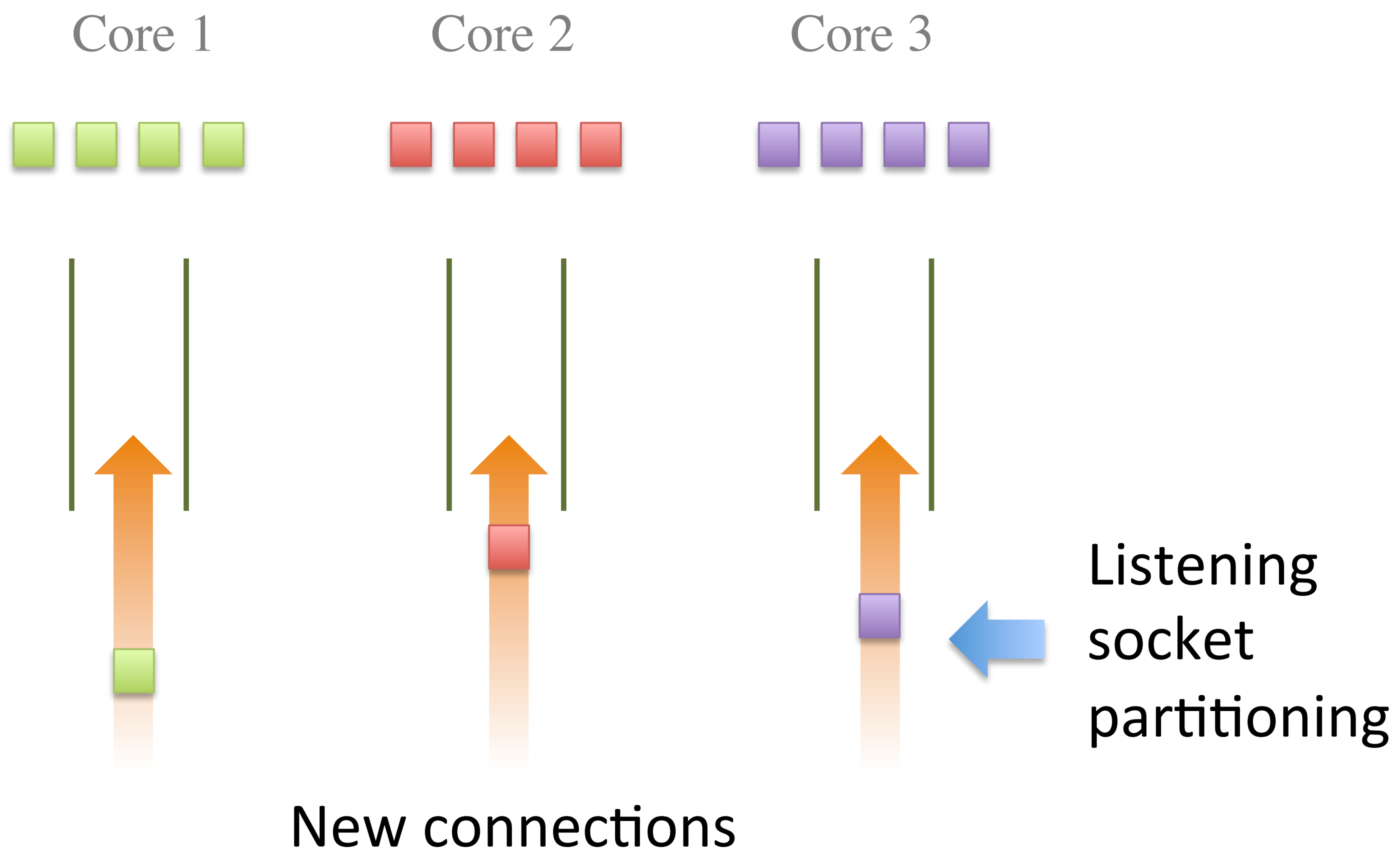
Sketch: How Channels Help (1/3)



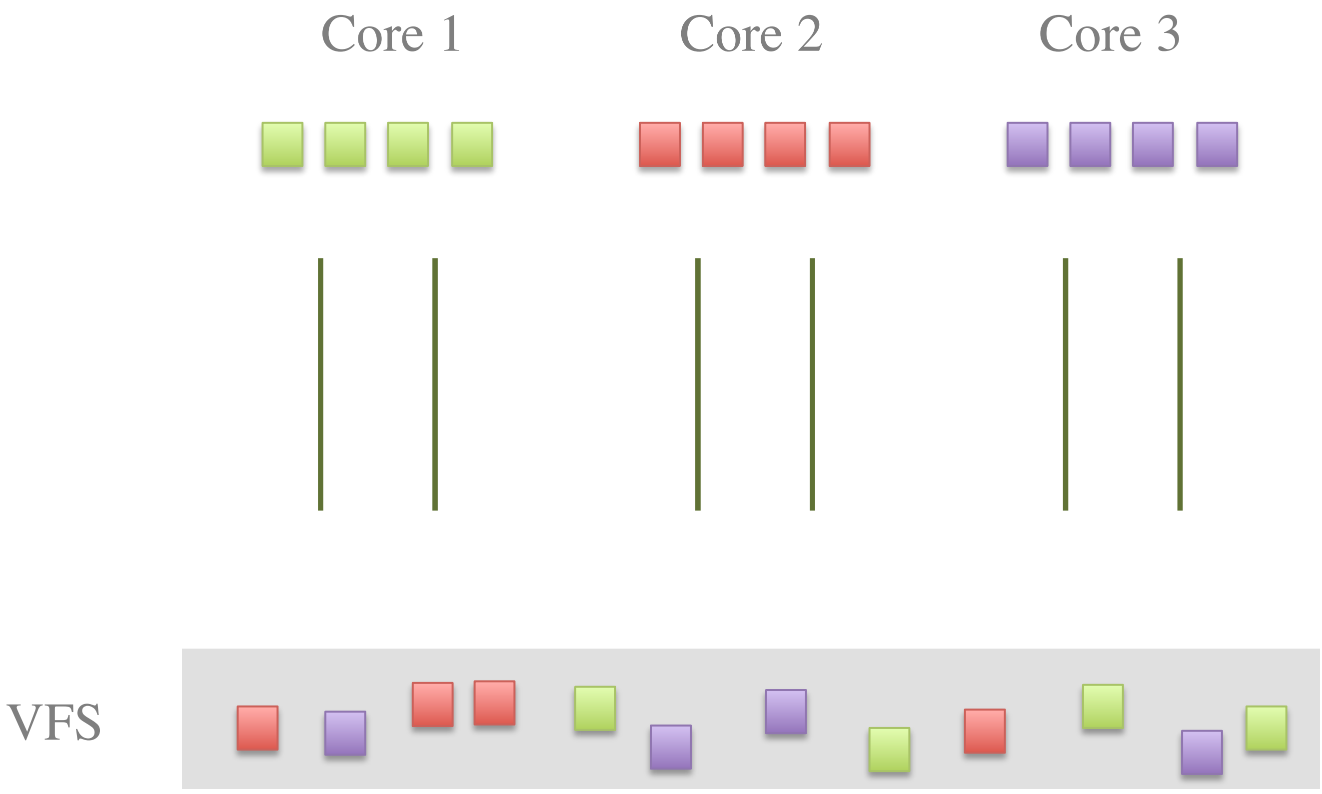
Sketch: How Channels Help (2/3)



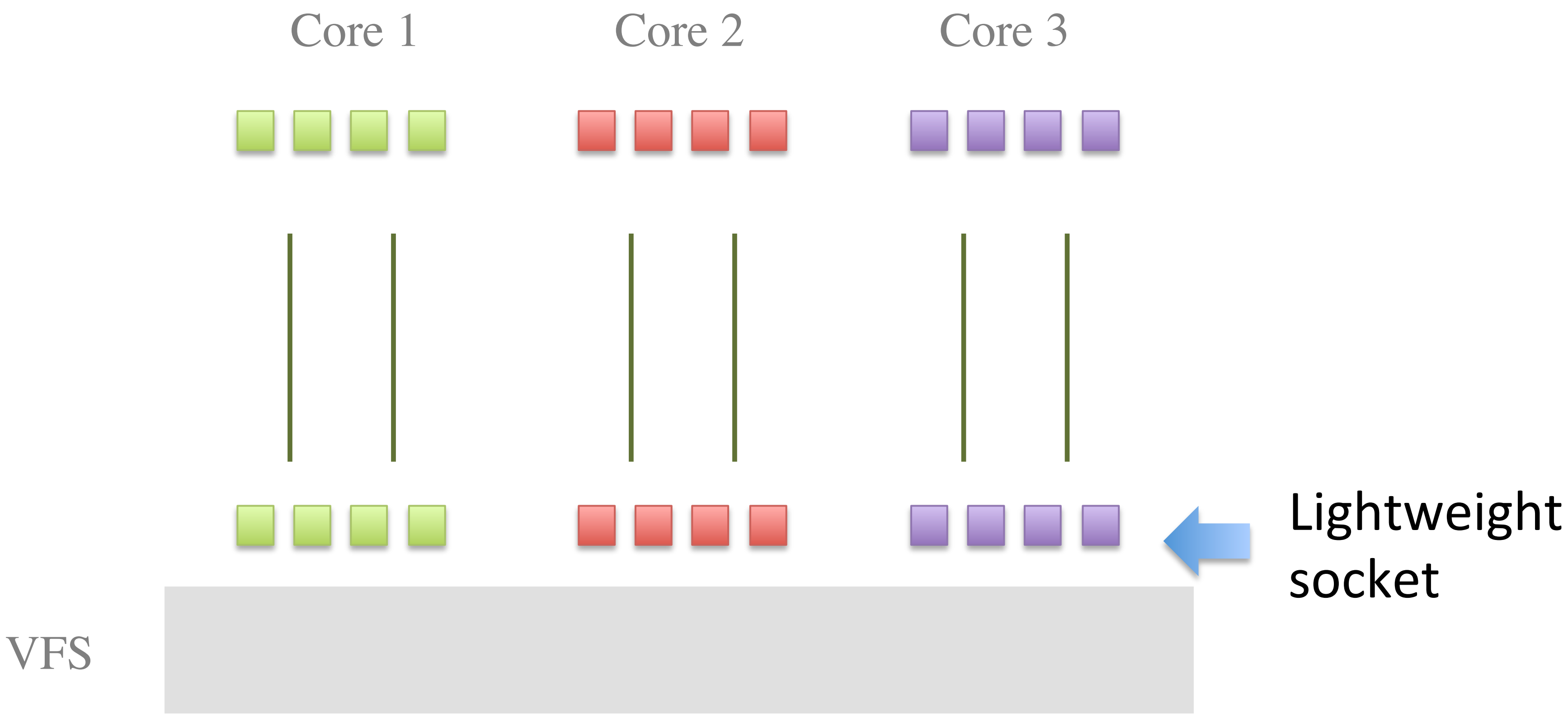
Sketch: How Channels Help (2/3)



Sketch: How Channels Help (3/3)



Sketch: How Channels Help (3/3)



MegaPipe API Functions

- mp_create() / mp_destroy()
  - Create/close a channel
- mp_register() / mp_unregister()
  - Register a handle (regular FD or lwsocket) into a channel
- mp_accept() /mp_read() / mp_write() / …
  - Issue an asynchronous I/O command for a given handle
- mp_dispatch()
  - Dispatch an I/O completion event from a channel

Completion Notification Model

- BSD Socket API
  - Wait-and-Go (Readiness model)

```
epoll_ctl(fd1, EPOLLIN);
epoll_ctl(fd2, EPOLLIN);
epoll_wait(…);
…
ret1 = recv(fd1, …);
…
ret2 = recv(fd2, …);
…
```

☹

- MegaPipe
  - Go-and-Wait (Completion notification)

```
mp_read(handle1, …);
mp_read(handle2, …);
…
ev = mp_dispatch(channel);
…
ev = mp_dispatch(channel);
…
```

☺

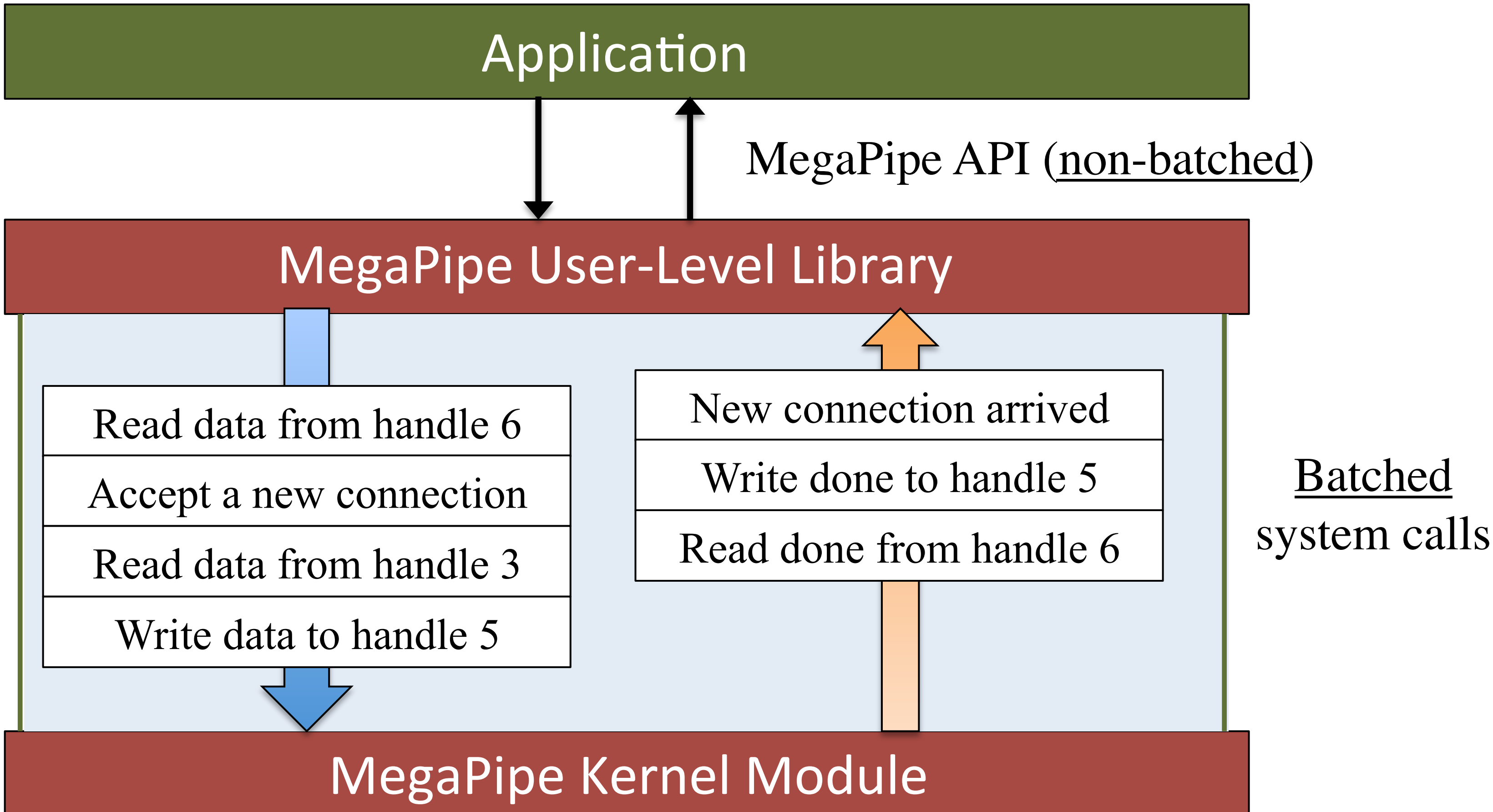
☺ Batching

☺ Easy and intuitive

☺ Compatible with disk files

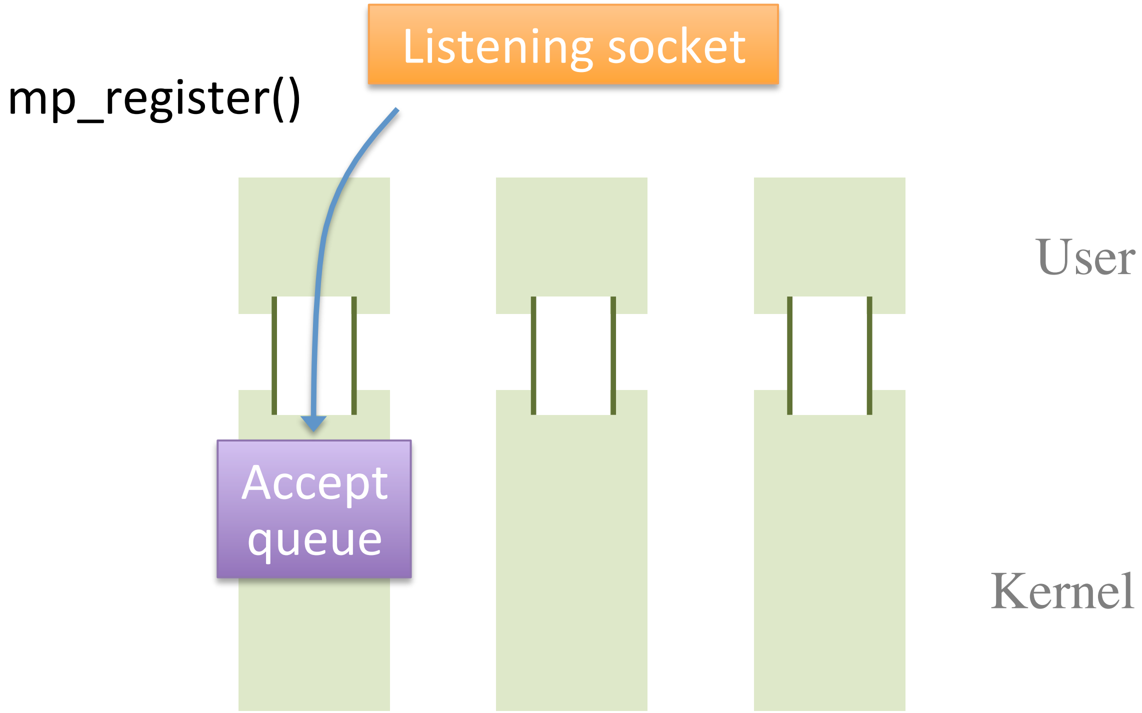
1. I/O Batching

- Transparent batching
  - Exploits parallelism of independent handles



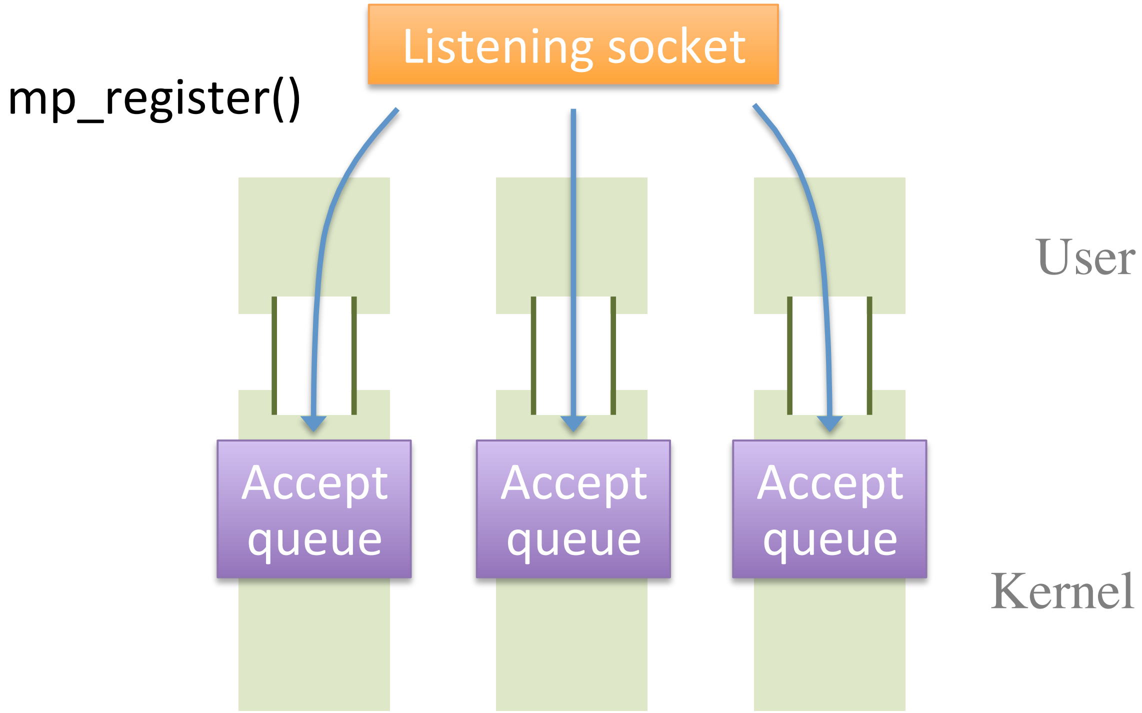
2. Listening Socket Partitioning

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



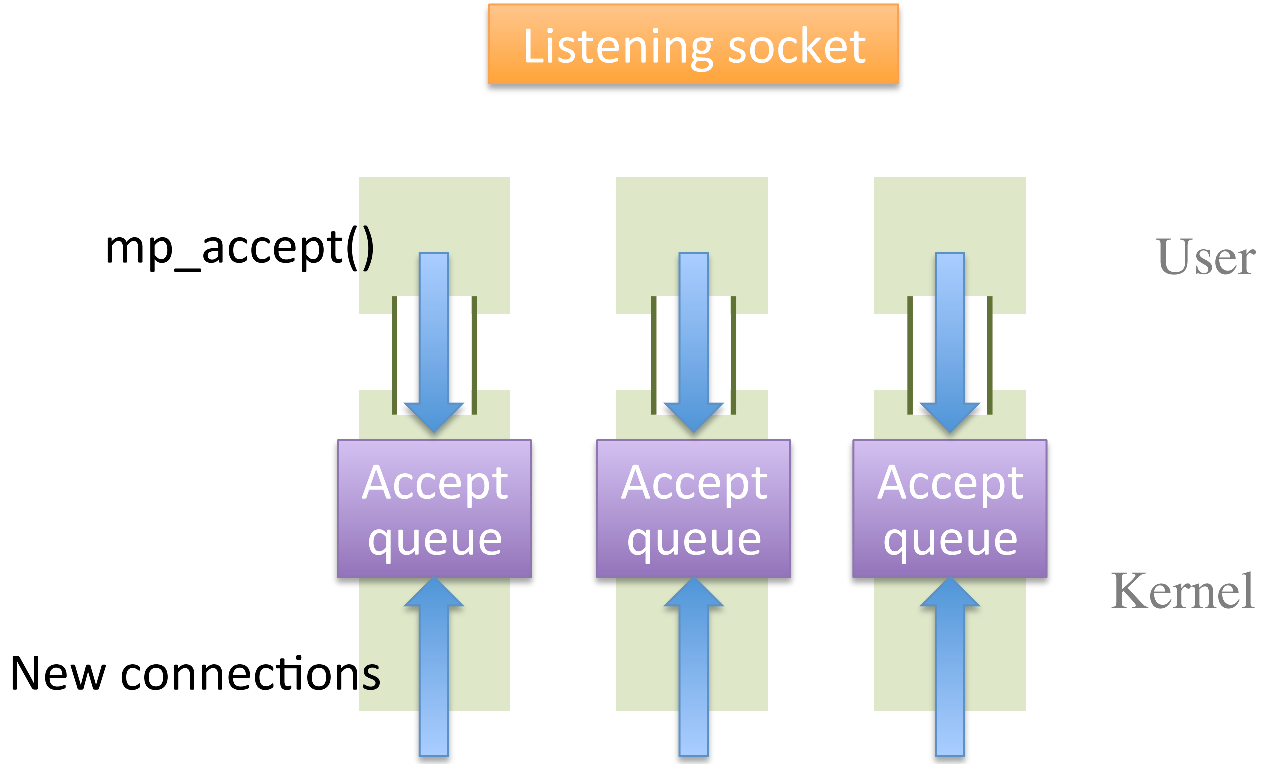
2. Listening Socket Partitioning

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



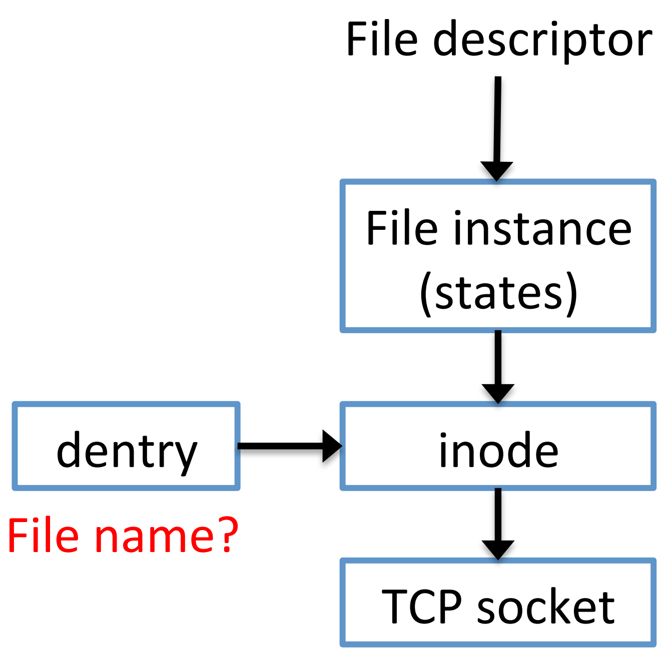
2. Listening Socket Partitioning

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



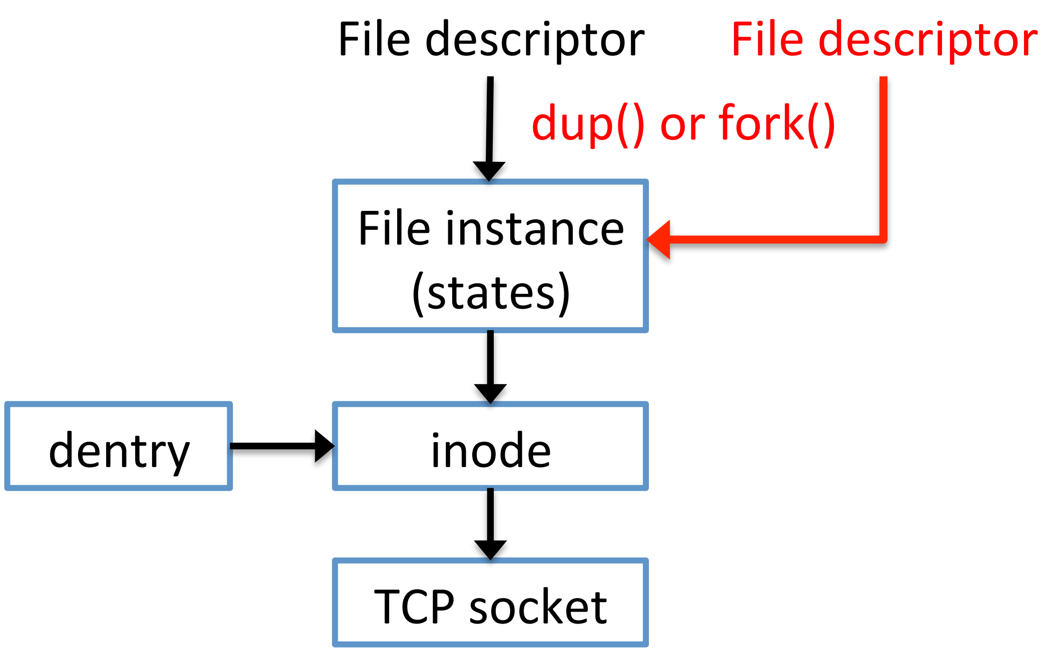
3. lwsocket: Lightweight Socket

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



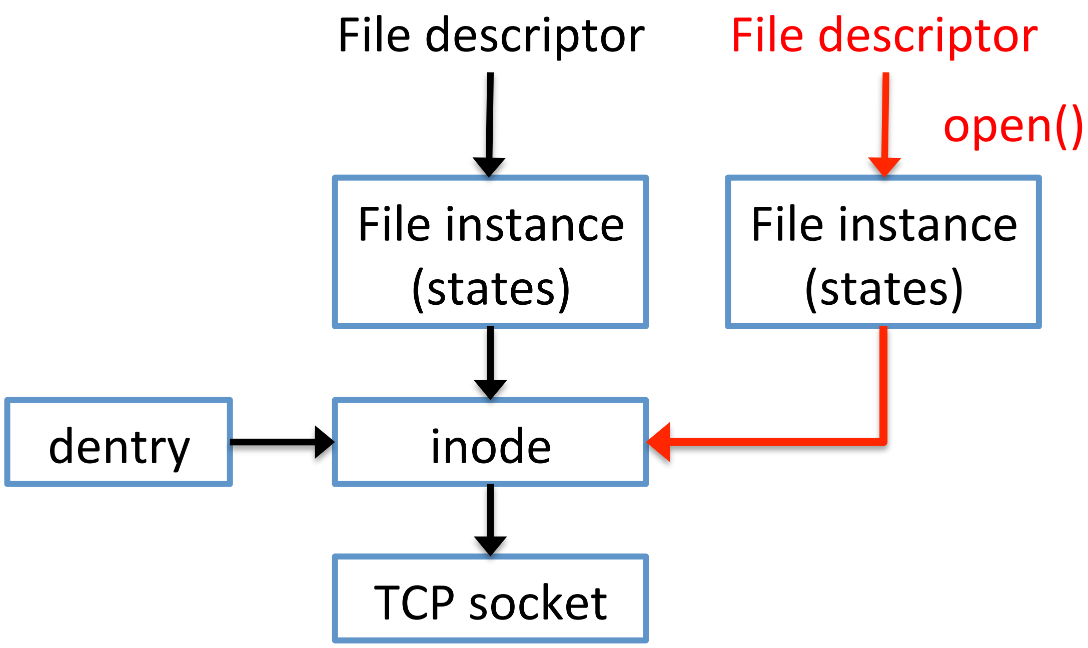
3. lwsocket: Lightweight Socket

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



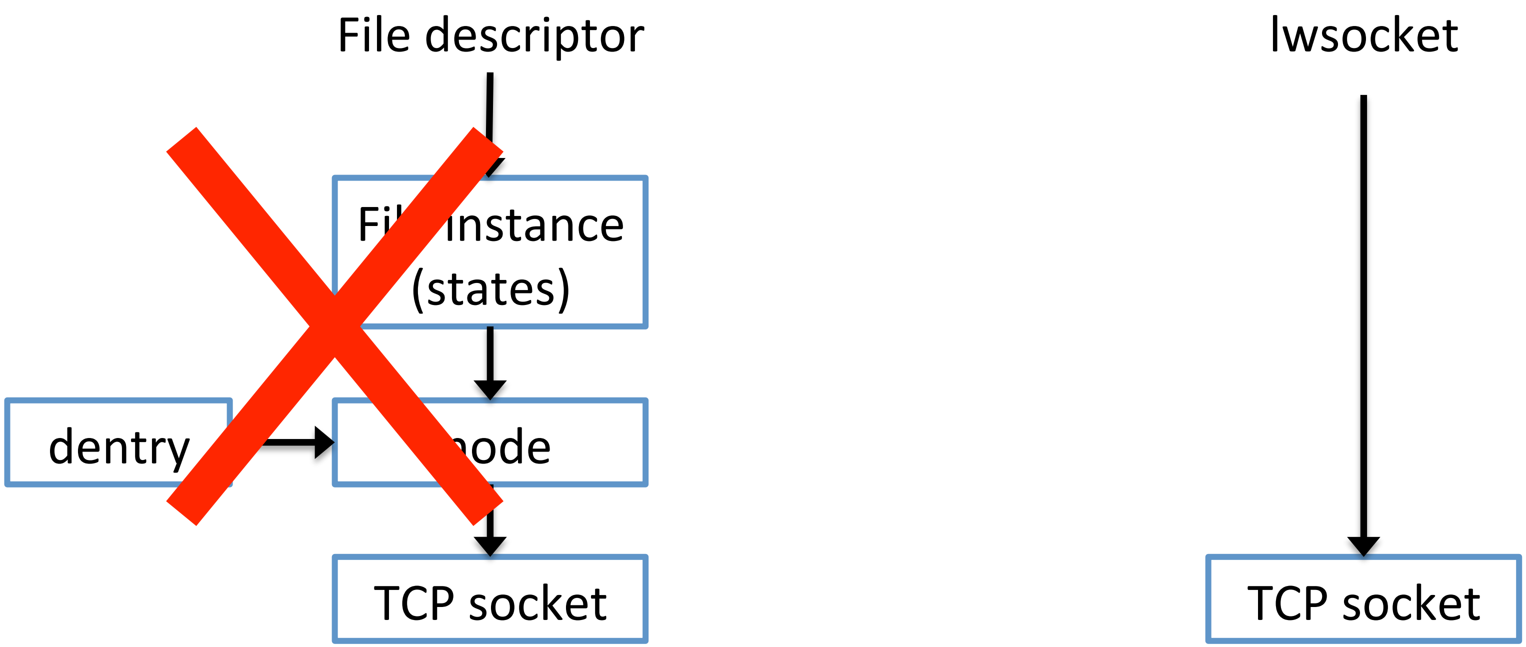
3. lwsocket: Lightweight Socket

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



3. lwsocket: Lightweight Socket

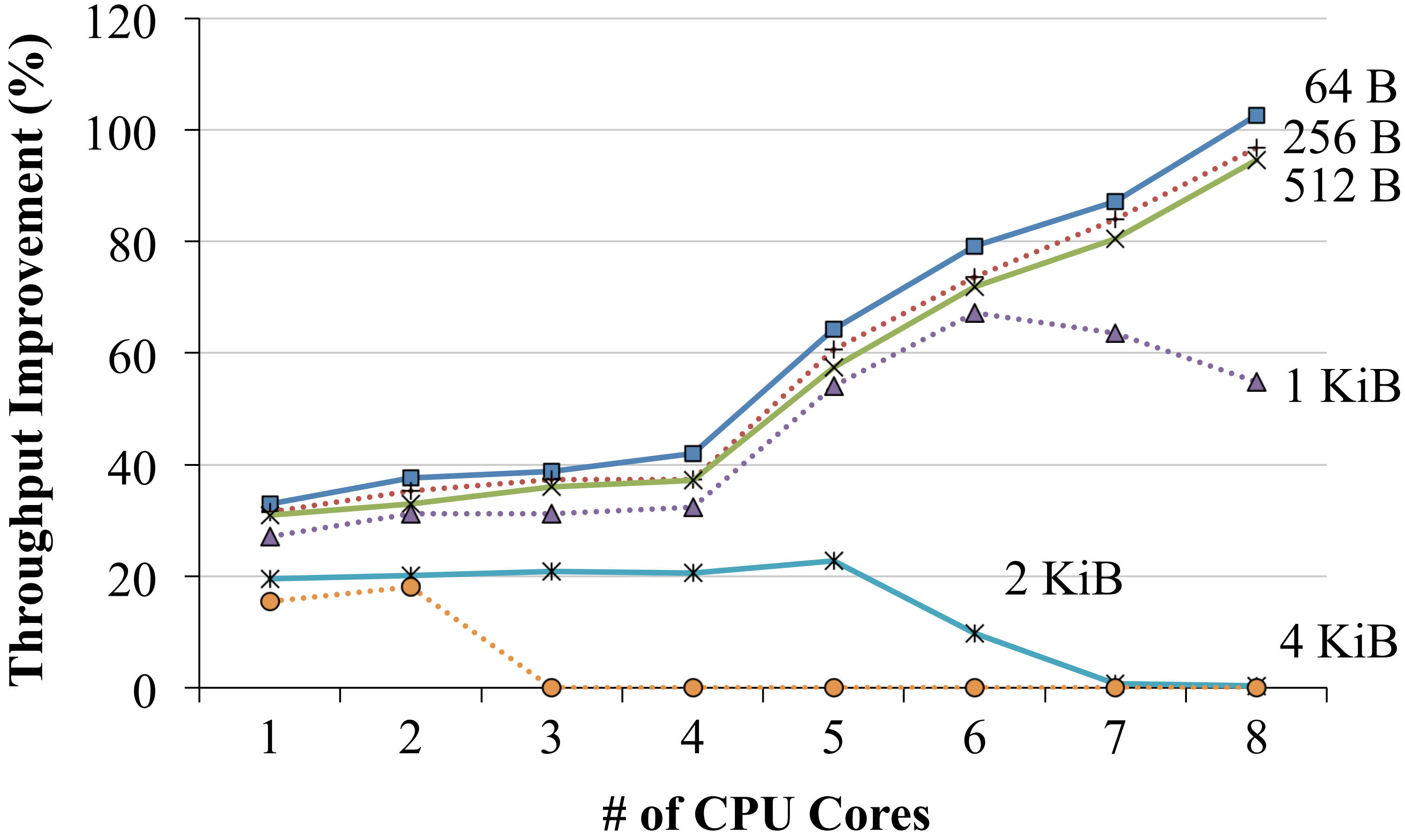
- Common-case optimization for sockets
  - Sockets are ephemeral and rarely shared
    - Bypass the VFS layer
    - Convert into a regular file descriptor only when necessary



EVALUATION

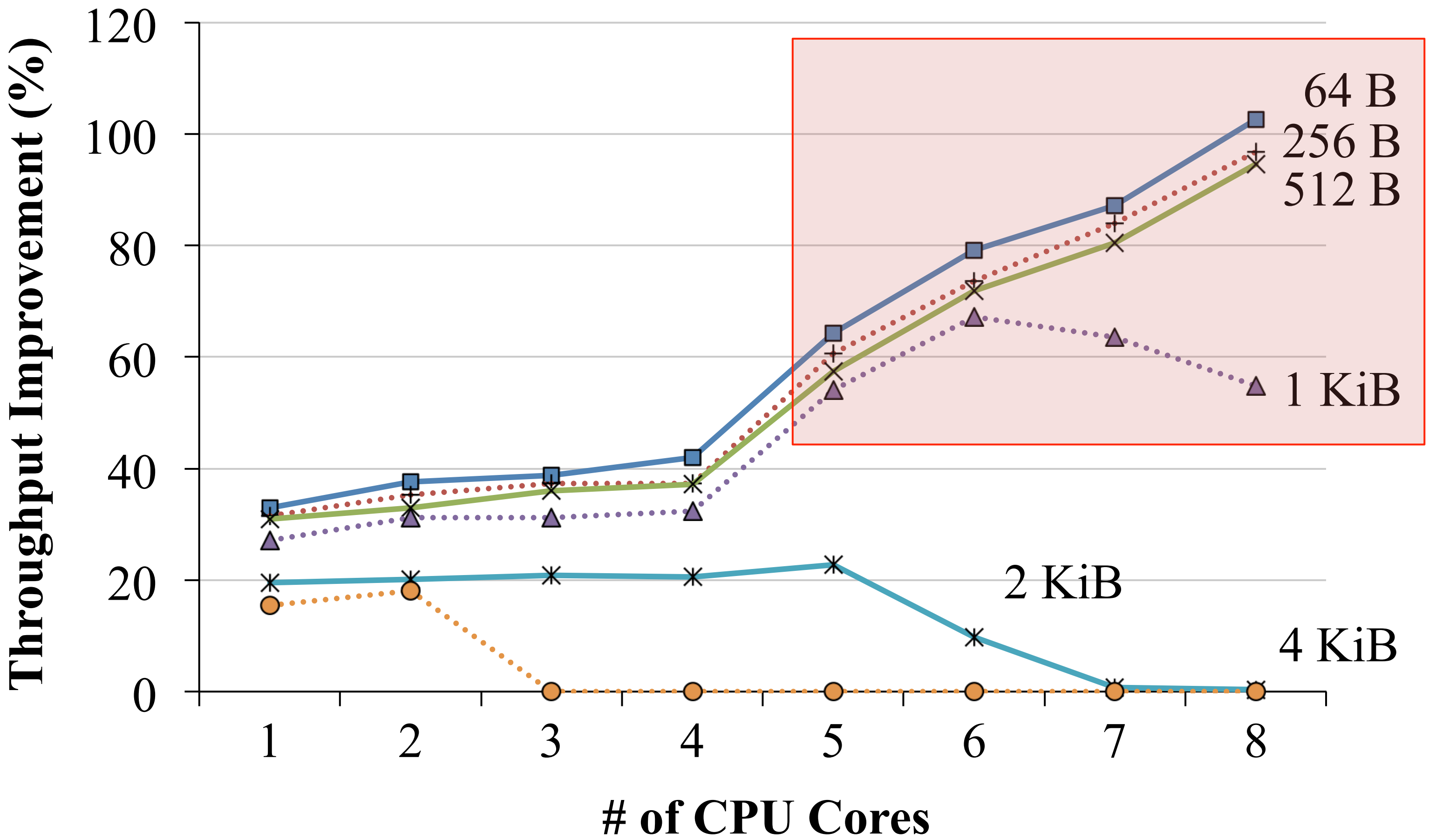
Microbenchmark 1/2

- Throughput improvement with various message sizes



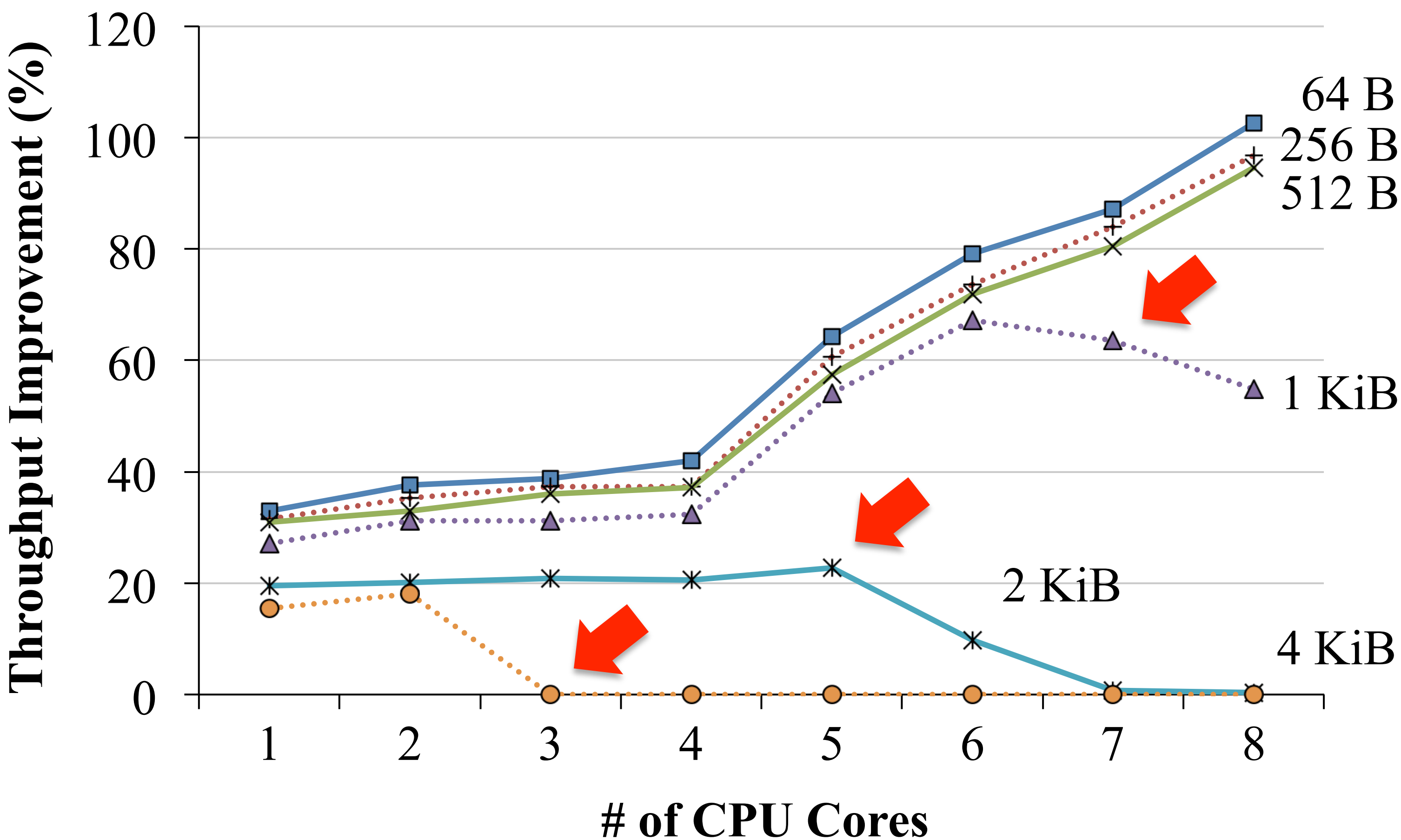
Microbenchmark 1/2

- Throughput improvement with various message sizes



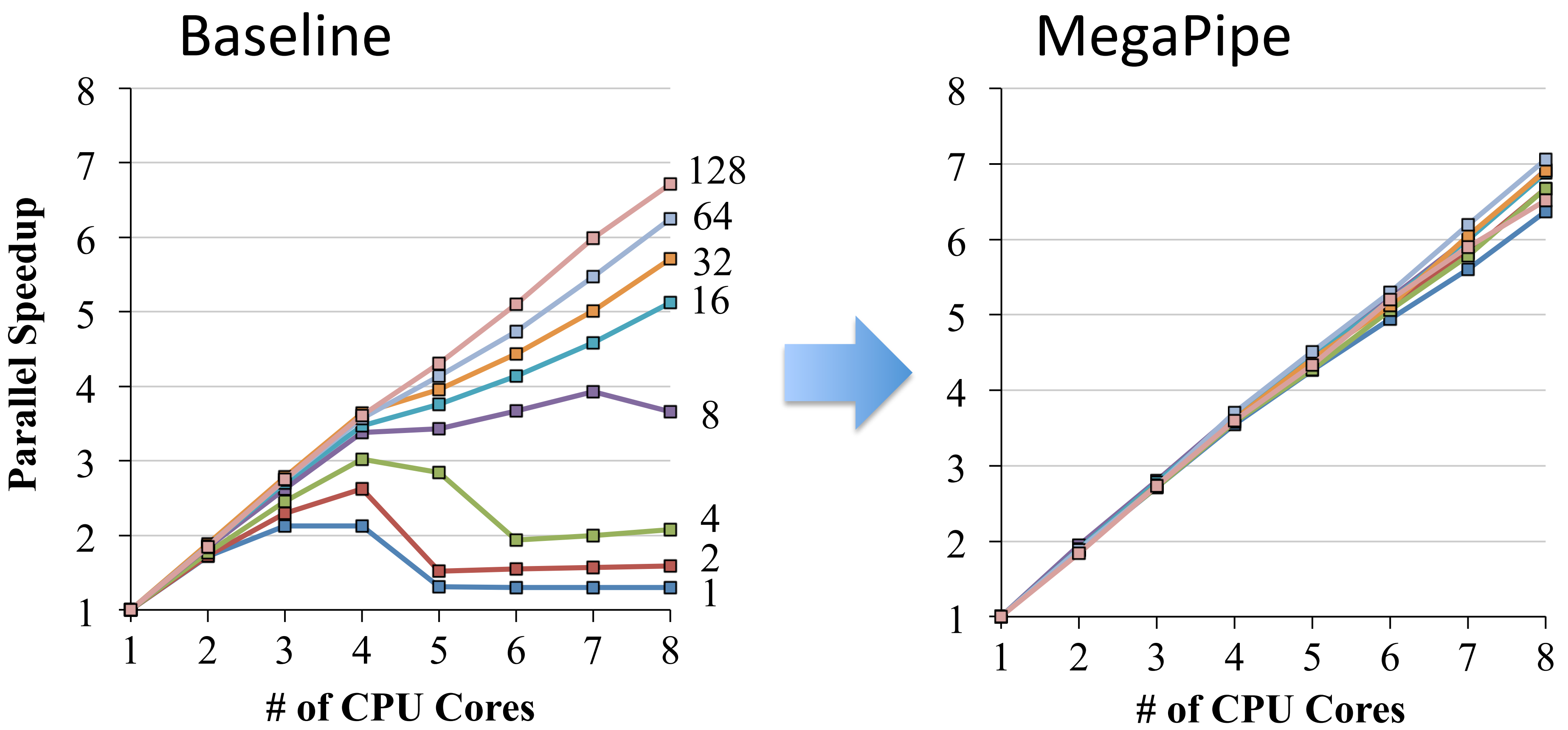
Microbenchmark 1/2

- Throughput improvement with various message sizes



Microbenchmark 2/2

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

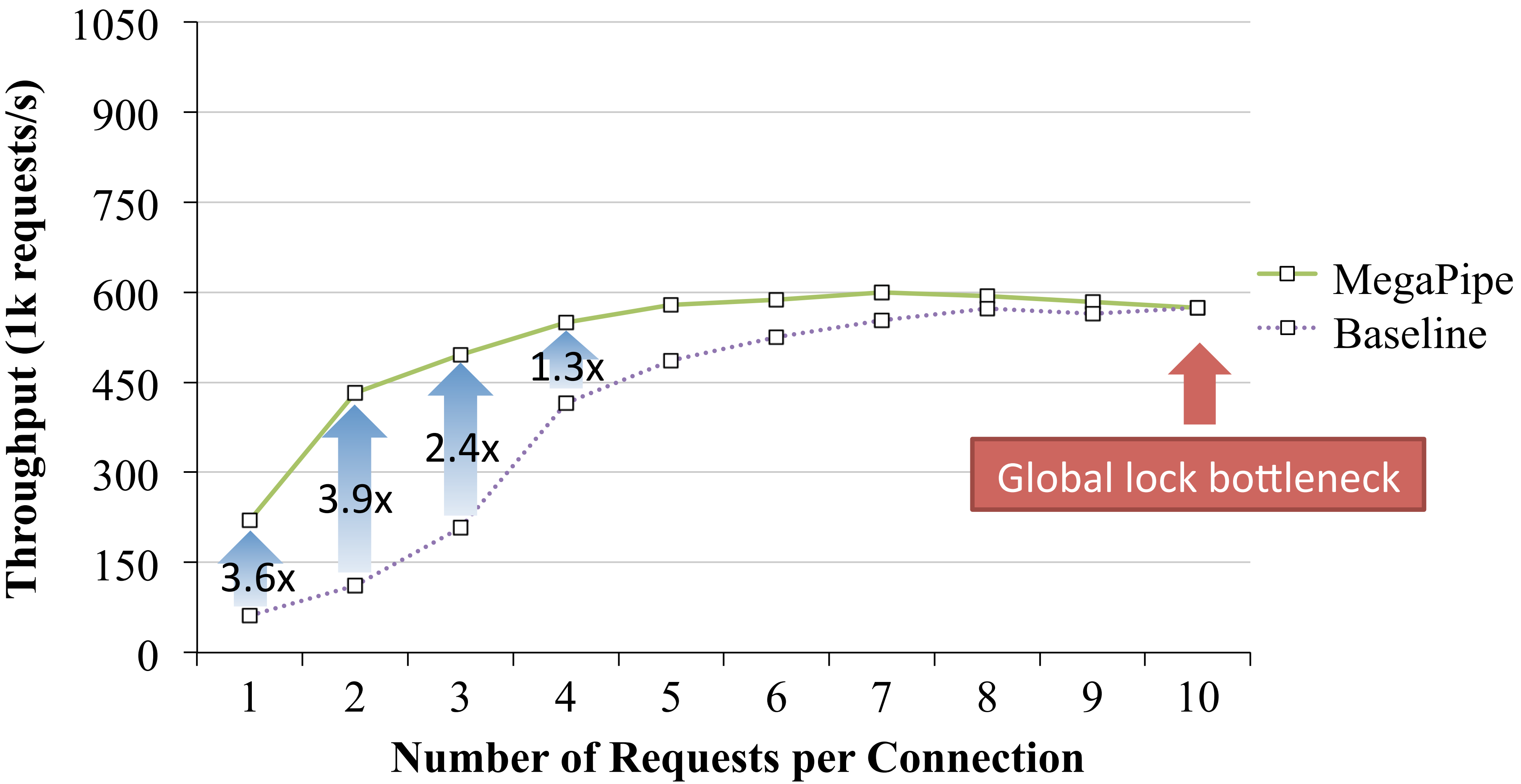


Macrobenchmark

- 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

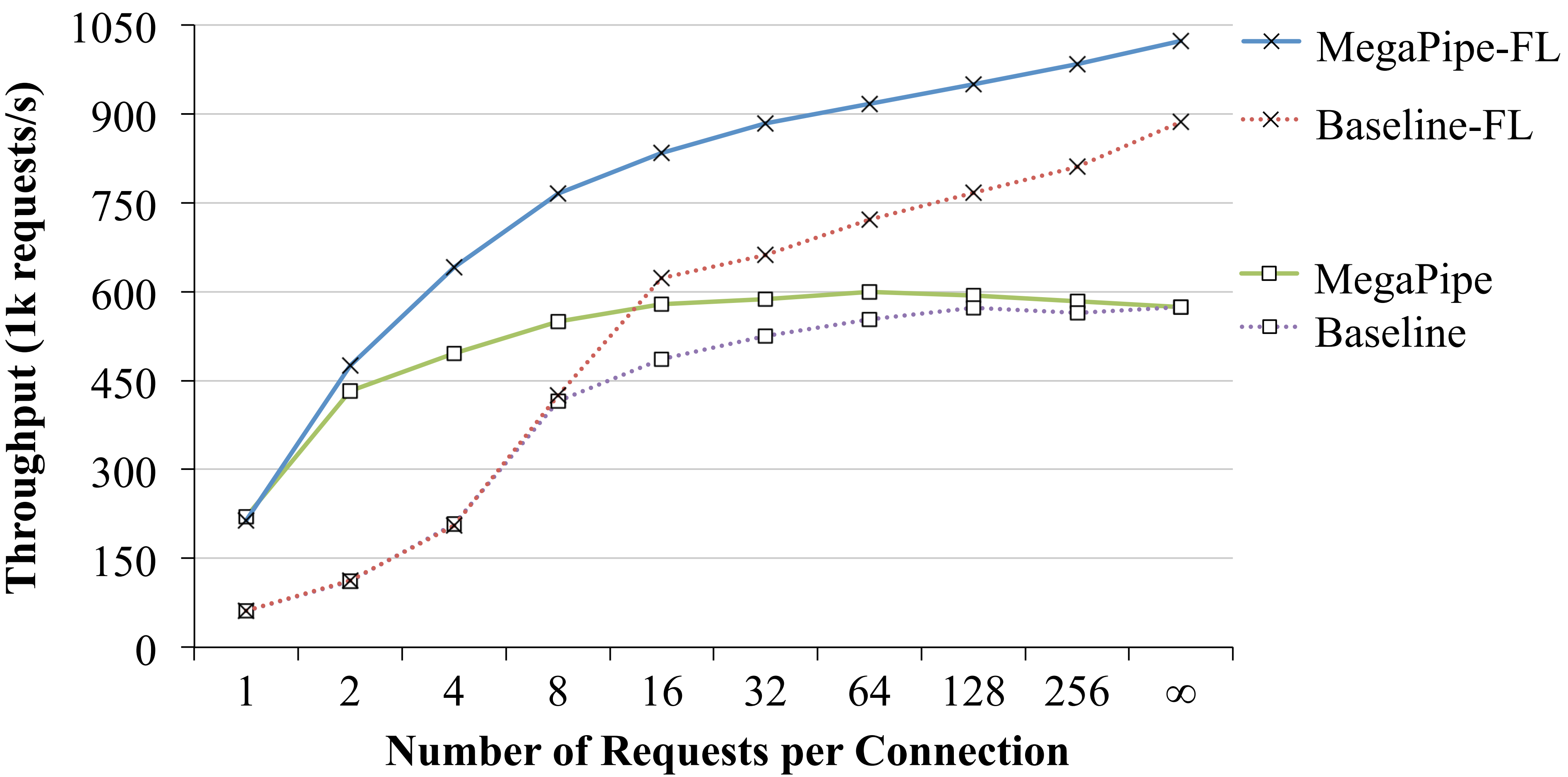
memcached

- memaslap with 90% GET, 10% SET, 64B keys, 1KB values



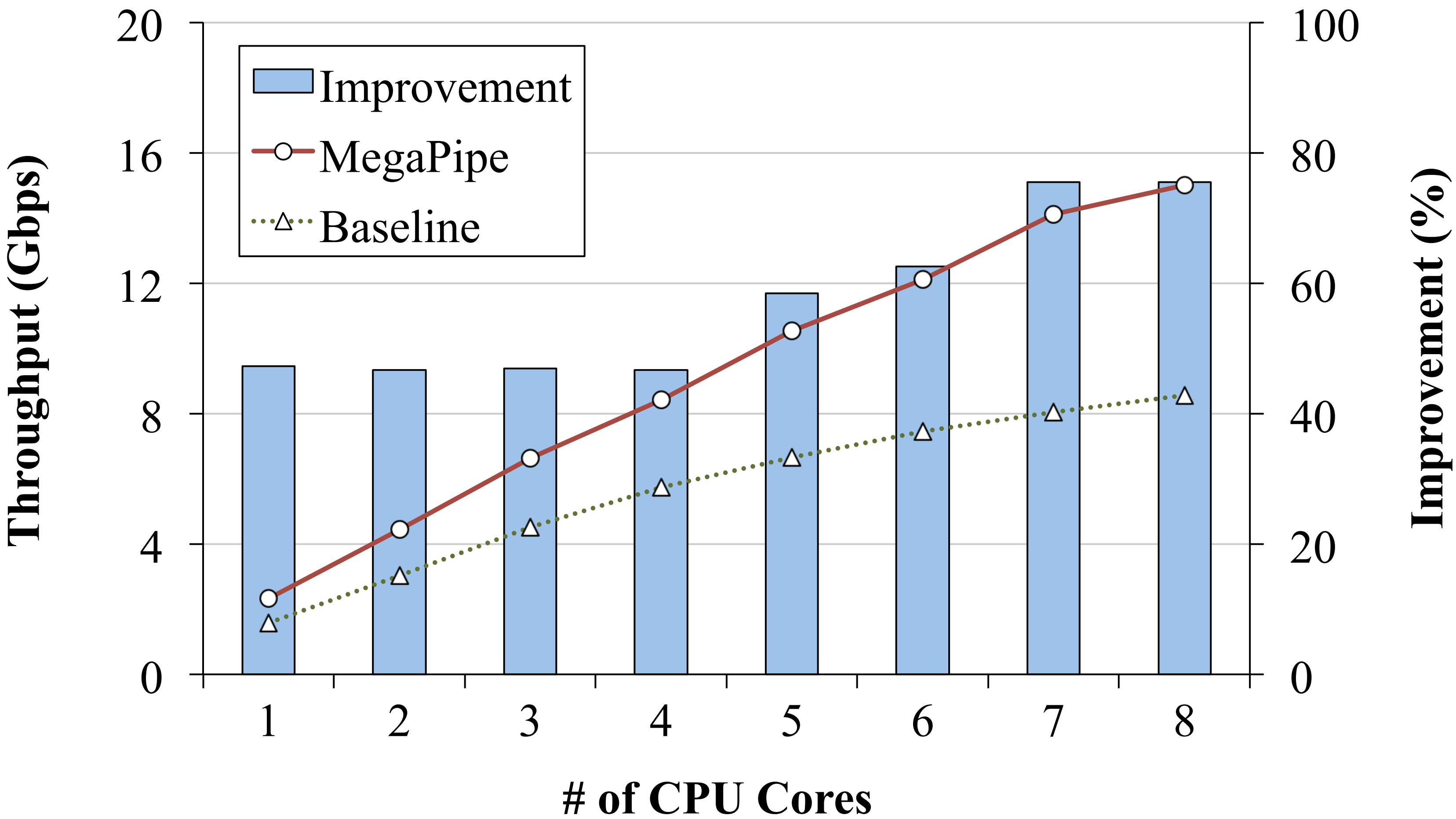
memcached

- memaslap with 90% GET, 10% SET, 64B keys, 1KB values



nginx

- Based on Yahoo! HTTP traces: 6.3KiB, 2.3 trans/conn on avg.



CONCLUSION

Related Work

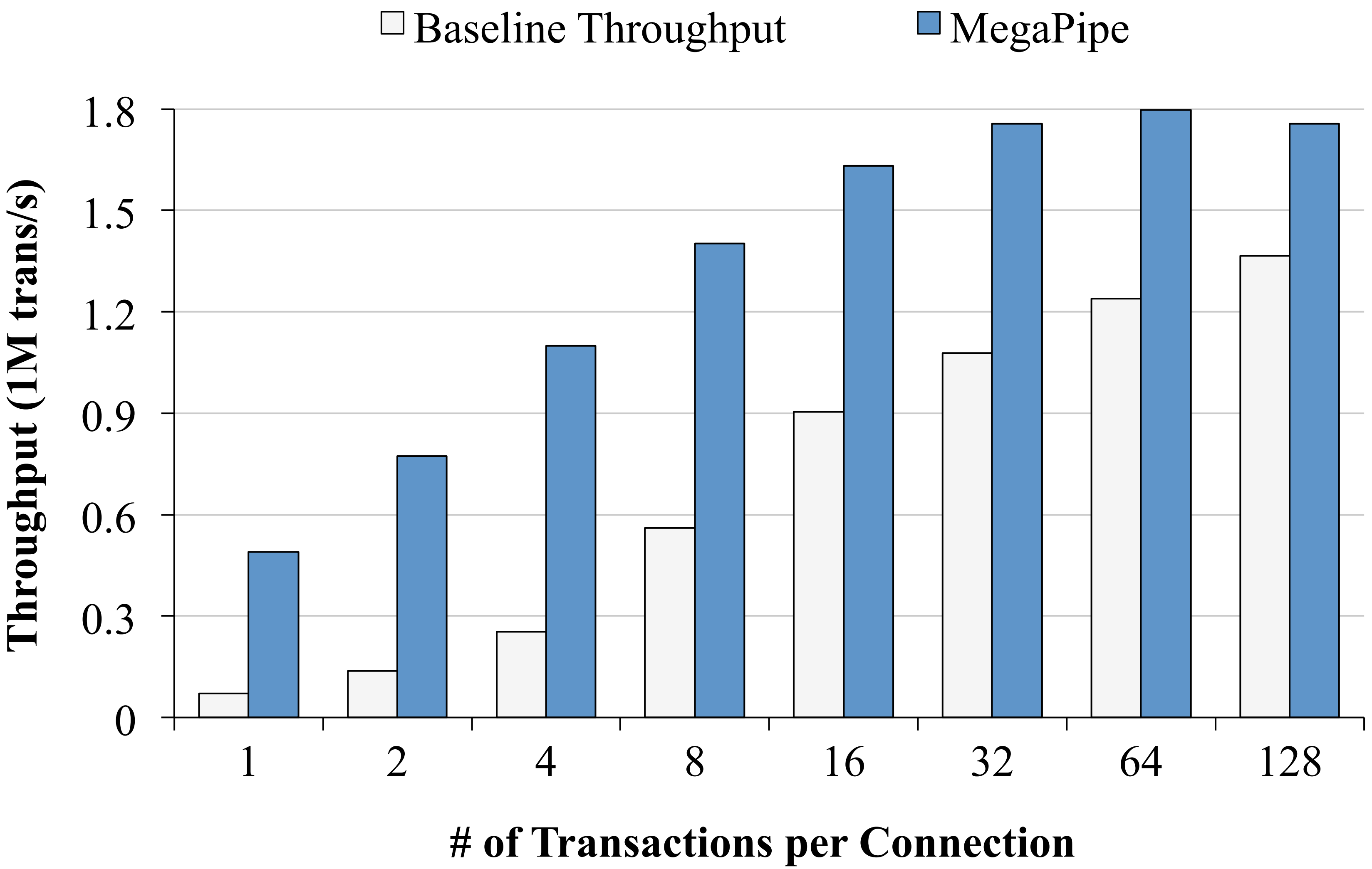
- Batching [FlexSC, OSDI'10] [libflexsc, ATC'11]
  - Exception-less system call
  - MegaPipe solves the scalability issues
- Partitioning [Affinity-Accept, EuroSys'12]
  - Per-core accept queue
  - MegaPipe provides explicit control over partitioning
- VFS scalability [Mosbench, OSDI'10]
  - MegaPipe bypasses the issues rather than mitigating

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

BACKUP SLIDES

Small Messages with MegaPipe

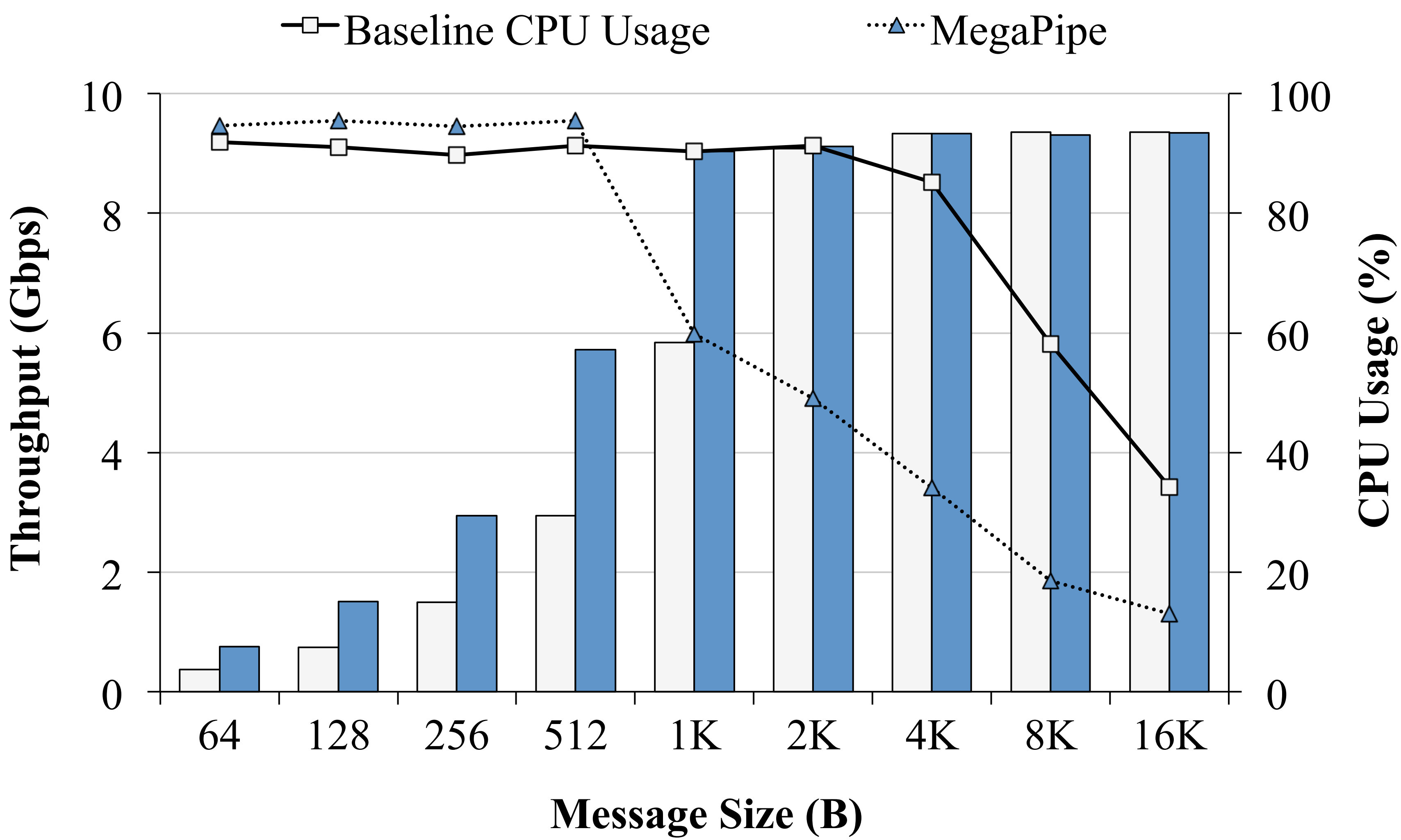


1. Small Messages Are Bad → Why?

- # of messages matters, not the volume of traffic
  - Per-message cost >>> per-byte cost
    - 1KB msg is only 2% more expensive than 64B msg
  - 10G link with 1KB messages → 1M IOPS!
    - Thus 1M+ system calls

- System calls are expensive [FlexSC, 2010]
  - Mode switching between kernel and user
  - CPU cache pollution

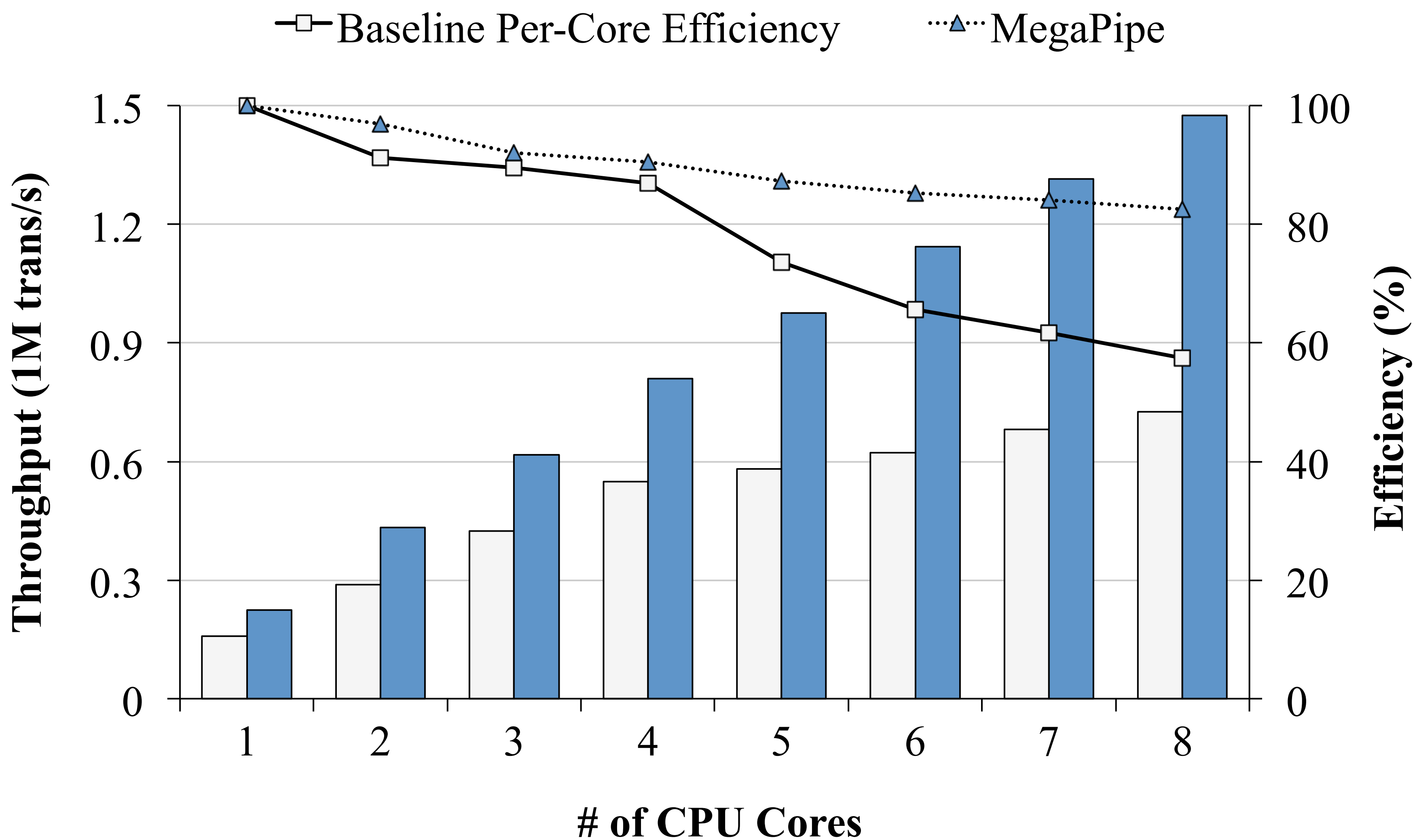
Short Connections with MegaPipe



2. Short Connections Are Bad → Why?

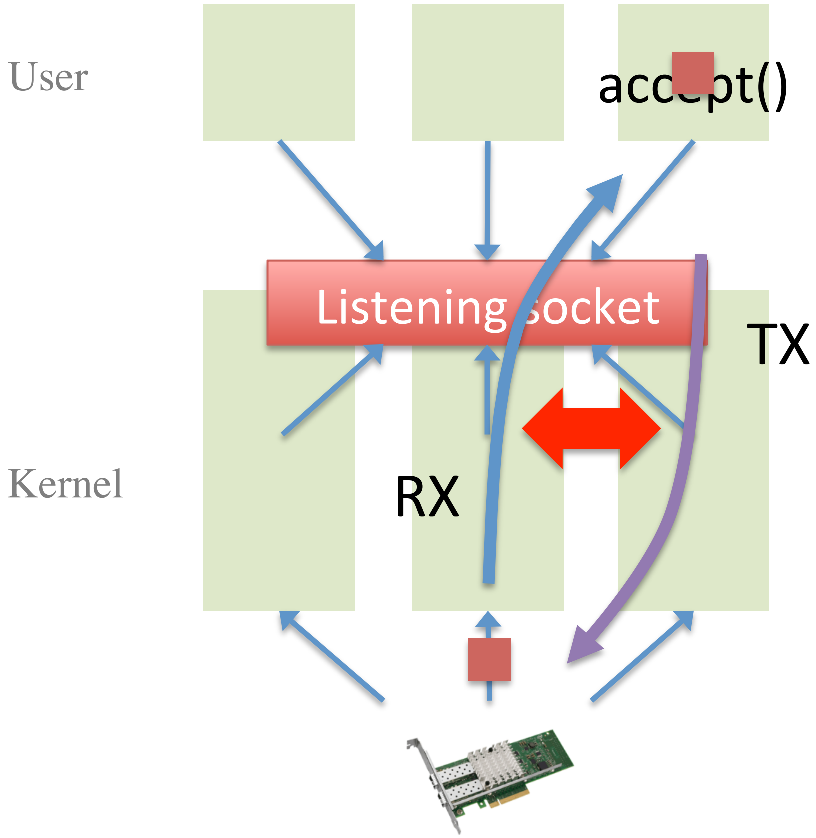
- Connection establishment is expensive
  - Three-way handshaking / four-way teardown
    - More packets
    - More system calls: accept(), epoll_ctl(), close(), …
  - Socket is represented as a file in UNIX
    - File overhead
    - VFS overhead

Multi-Core Scalability with MegaPipe



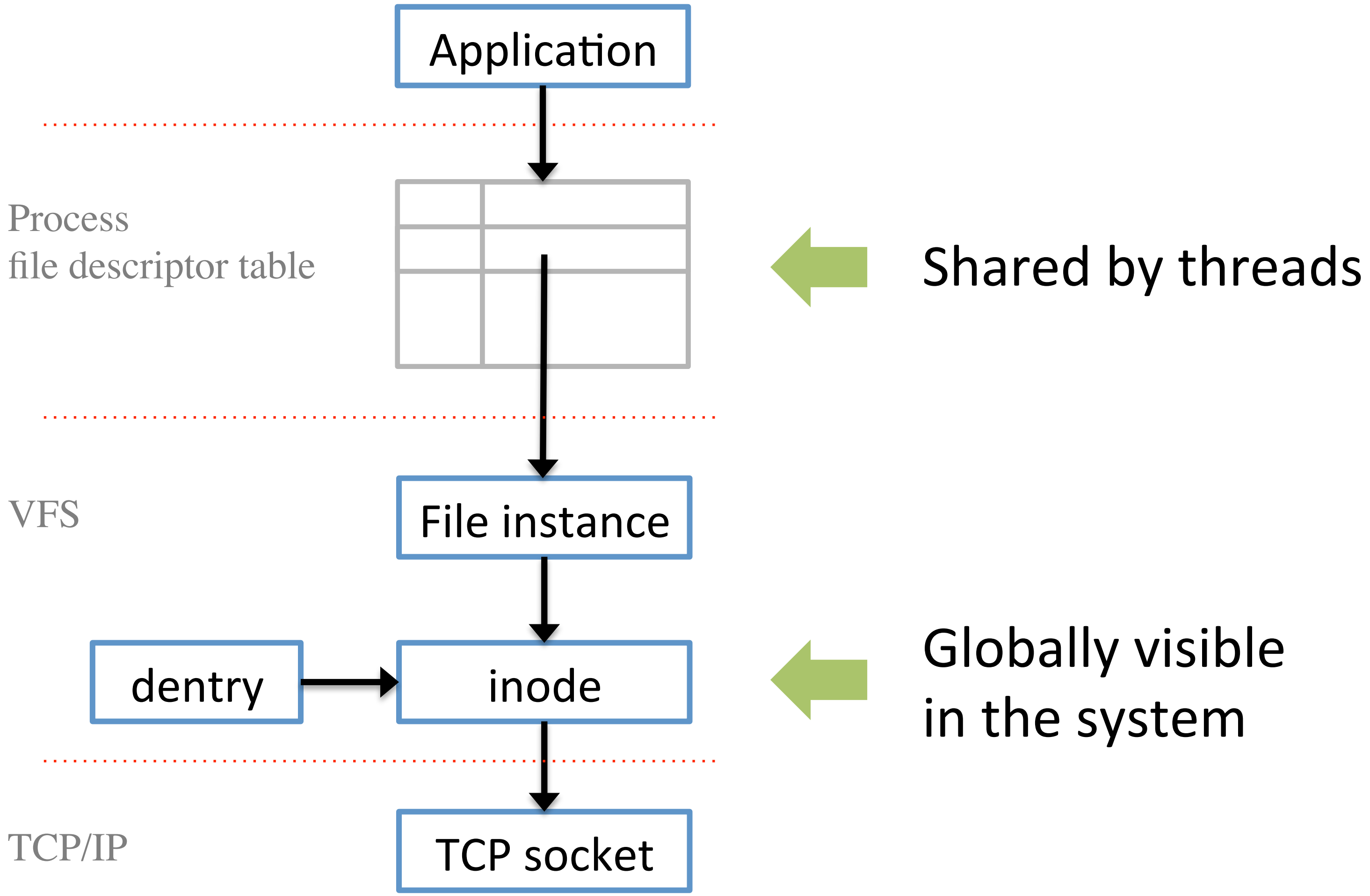
3. Multi-Core Will Not Help → Why?

- Shared queue issues [Affinity-Accept, 2012]
  - Contention on the listening socket
  - Poor connection affinity

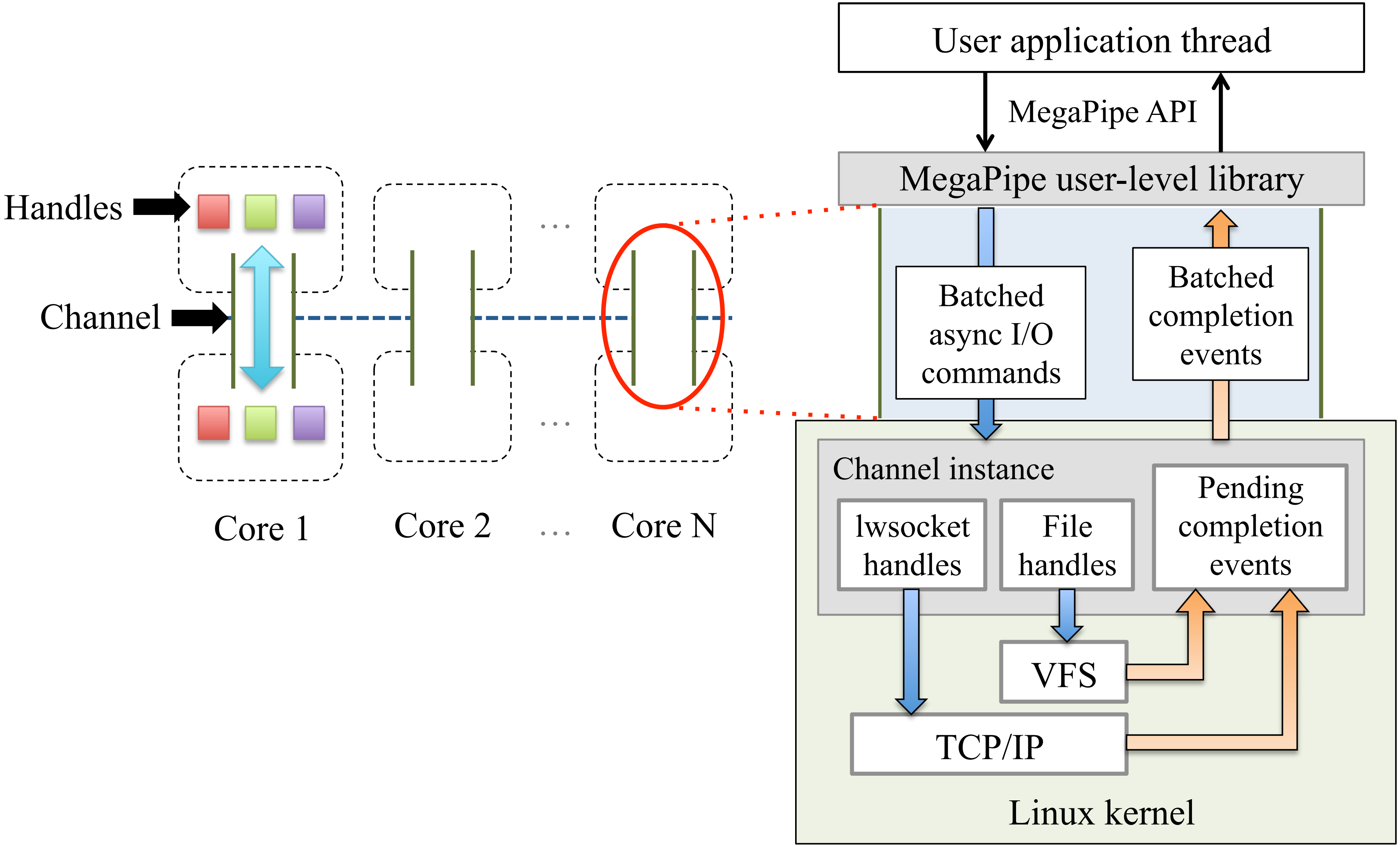


3. Multi-Core Will Not Help → Why?

- File/VFS multi-core scalability issues



Overview


Handles
Channel
Core 1
Core 2
…
Core N
User application thread
MegaPipe API
MegaPipe user-level library
Batched async I/O commands
Batched completion events
Channel instance
lwsocket handles
File handles
Pending completion events
VFS
TCP/IP
Linux kernel

Ping-Pong Server Example

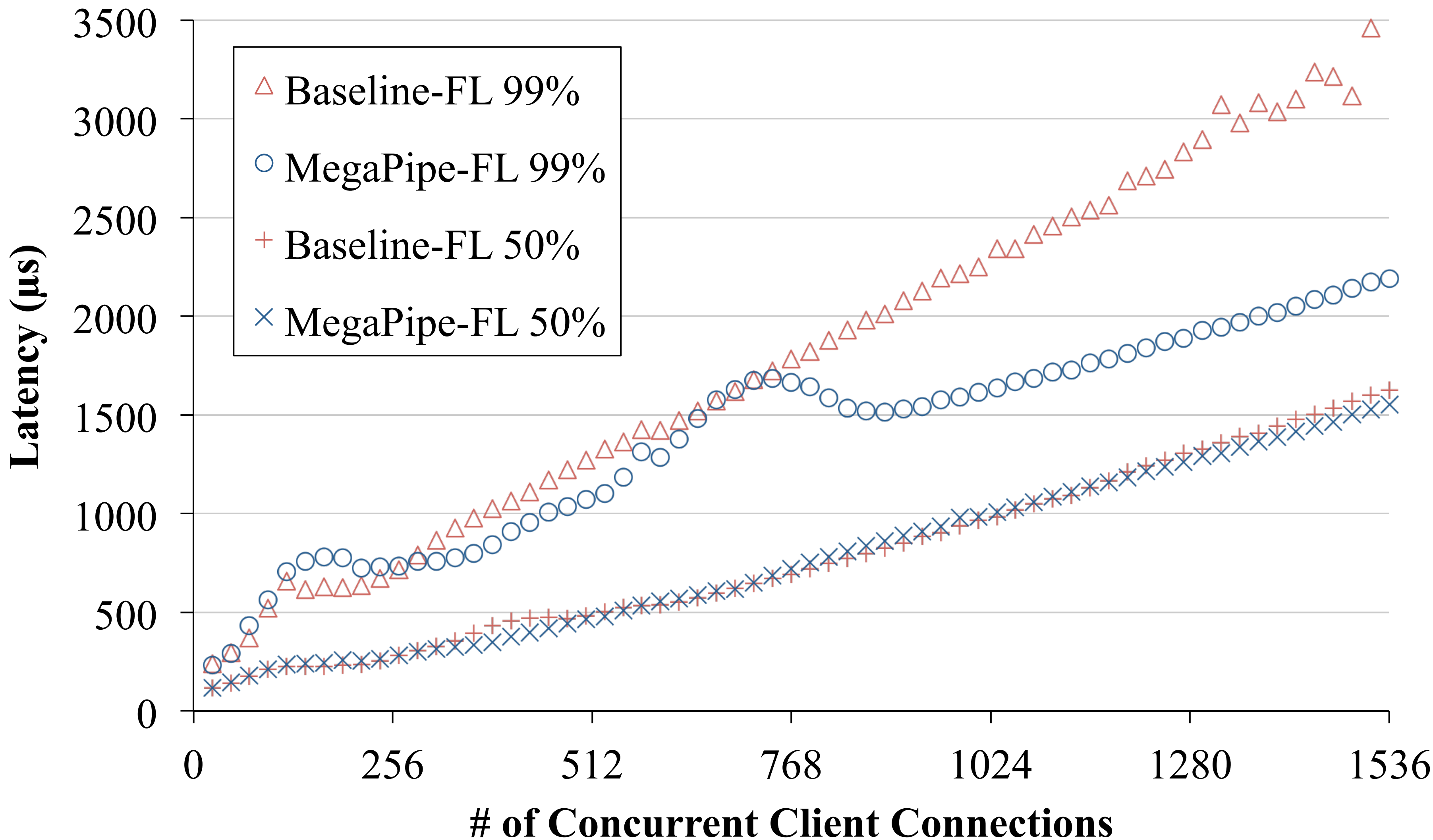
```
ch = mp_create()
handle = mp_register(ch, listen_sd, mask=my_cpu_id)
mp_accept(handle)

while true:
   ev = mp_dispatch(ch)
   conn = ev.cookie
   if ev.cmd == ACCEPT:
      mp_accept(conn.handle)
      conn = new Connection()
      conn.handle = mp_register(ch, ev.fd, cookie=conn)
      mp_read(conn.handle, conn.buf, READSIZE)
   elif ev.cmd == READ:
      mp_write(conn.handle, conn.buf, ev.size)
   elif ev.cmd == WRITE:
      mp_read(conn.handle, conn.buf, READSIZE)
   elif ev.cmd == DISCONNECT:
      mp_unregister(ch, conn.handle)
      delete conn
```

Contribution Breakdown

	Number of transactions per connection							
	1	2	4	8	16	32	64	128
+P	211.6	207.5	181.3	83.5	38.9	29.5	17.2	8.8
P +B	18.8	22.8	72.4	44.6	31.8	30.4	27.3	19.8
PB +L	352.1	230.5	79.3	22.0	9.7	2.9	0.4	0.1
Total	582.4	460.8	333.1	150.1	80.4	62.8	45.0	28.7

Table 3: Accumulation of throughput improvement (%) over baseline, from three contributions of MegaPipe.

memcached latency



Clean-Slate vs. Dirty-Slate

- MegaPipe: a clean-slate approach with new APIs
  - Quick prototyping for various optimizations
  - Performance improvement: worthwhile!
- Can we apply the same techniques back to the BSD Socket API?
  - Each technique has its own challenges
    - Embracing all could be even harder
  - Future Work™