DPDK

DATA PLANE DEVELOPMENT KIT

Topic: Fast User-level TCP Stack on DPDK

Company: KAIST

Title: mTCP: A High-Speed User-Level TCP Stack on DPDK

Name: KyoungSoo Park

mTCP: A High-Speed User-Level TCP Stack on DPDK

KyoungSoo Park

In collaboration with

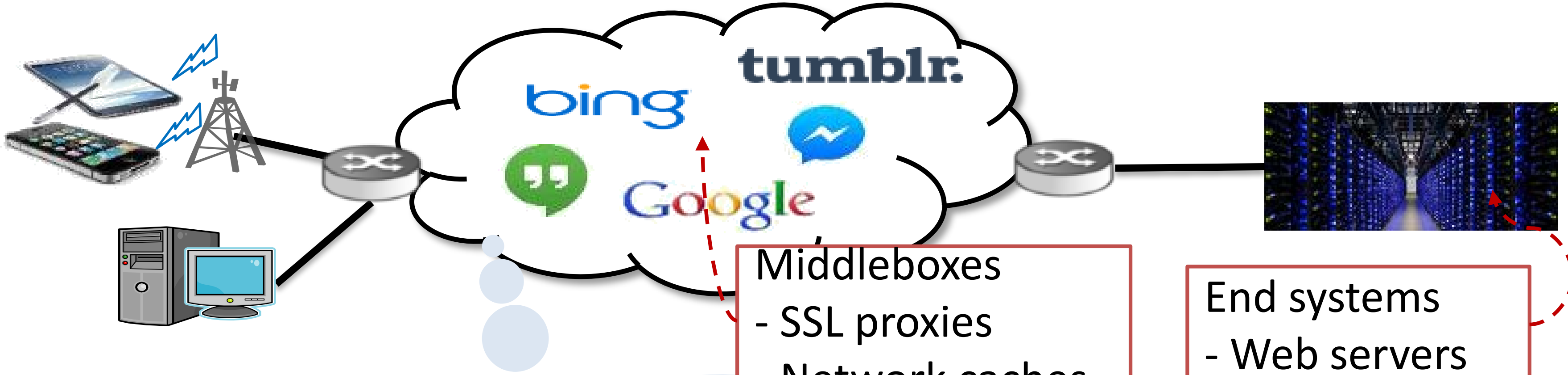
Eunyoung Jeong, Shinae Woo, Asim Jamshed,

Haewon Jeong, Sunghwan Ihm+, Dongsu Han

School of Electrical Engineering, KAIST &

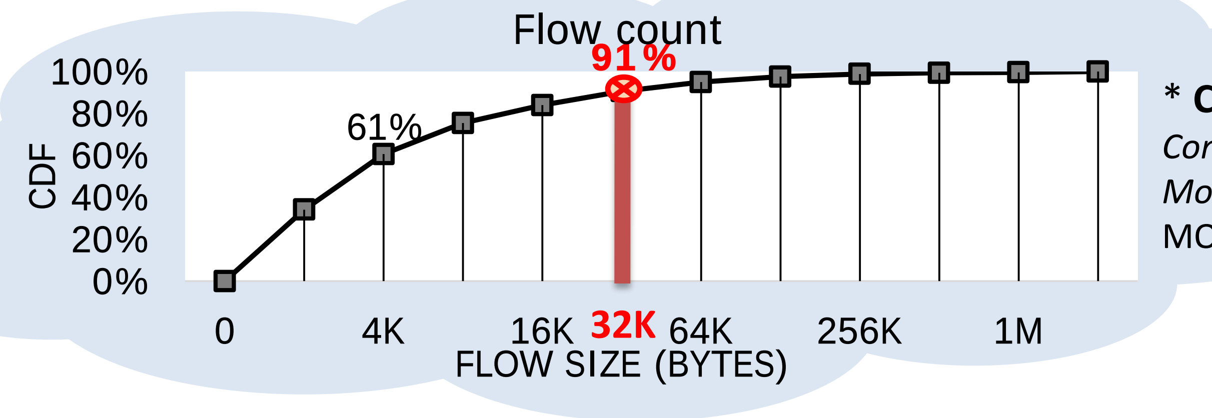
+Department of Computer Science, Princeton University

Short TCP Flows Dominate the Internet



Middleboxes
- SSL proxies
- Network caches

End systems
- Web servers

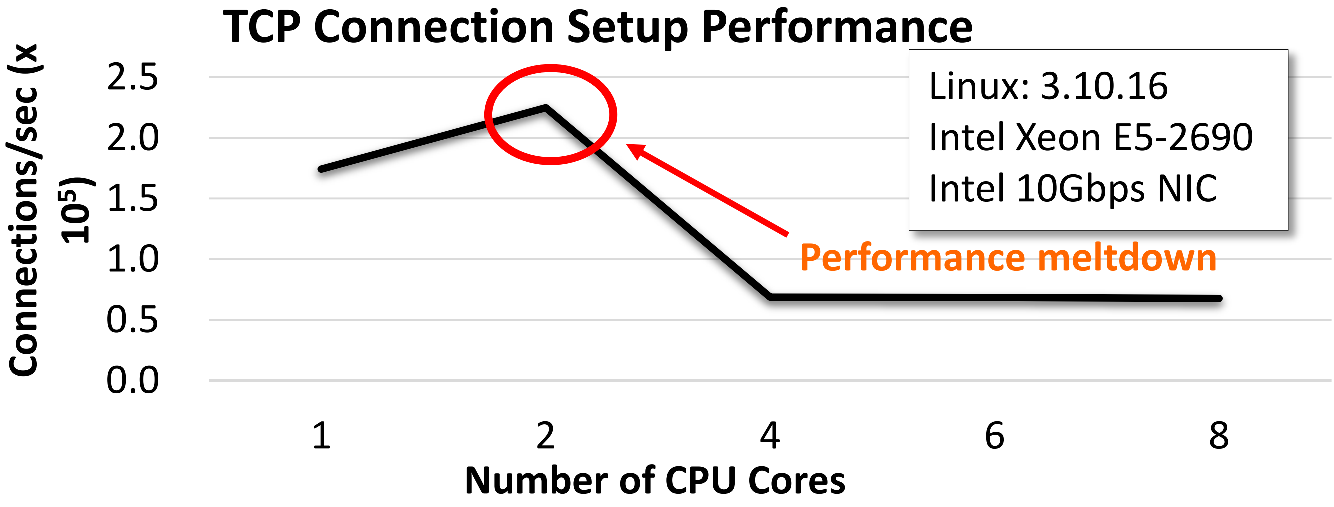


*** Commercial cellular traffic for 7 days**

Comparison of Caching Strategies in Modern Cellular Backhaul Networks, MOBISYS 2013

Suboptimal Linux TCP Performance

- Large flows: Easy to fill up 10 Gbps
- Small flows: Hard to fill up 10 Gbps regardless of # CPU cores
  - Too many packets:
    14.88 Mpps for 64B packets in a 10 Gbps link
  - Kernel is ***not** designed* for multicore systems

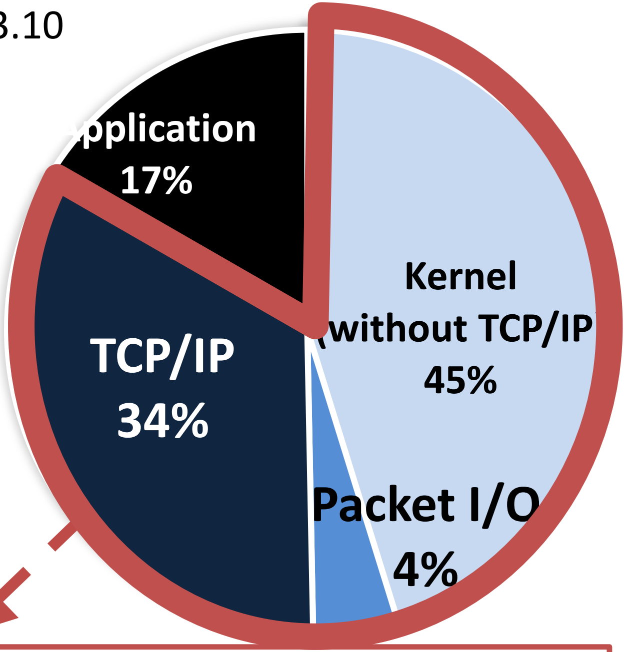


Inefficient Kernel Code for TCP Transactions

CPU Usage Breakdown of Web Server

Web server (Lighttpd) Serving a 64 byte file

Linux-3.10



83% of CPU usage spent inside kernel!

Performance bottlenecks

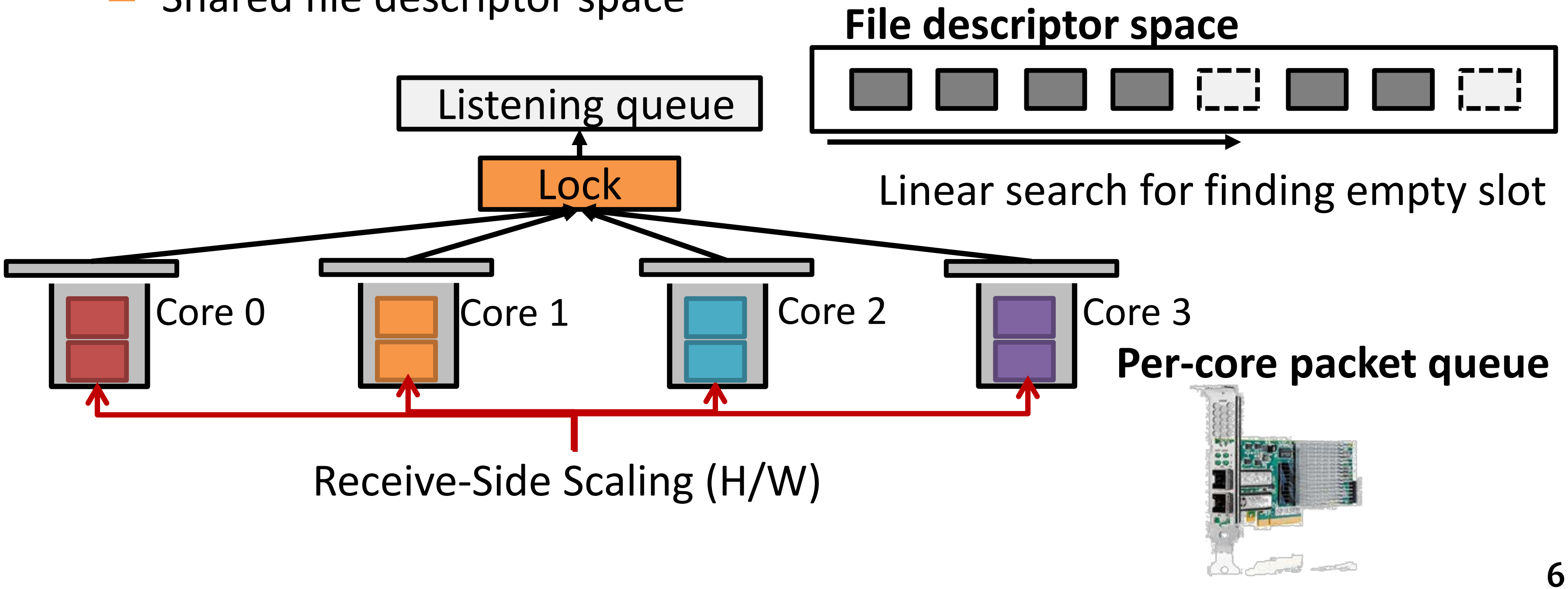
1. Shared resources
2. Broken locality
3. Per packet processing

Bottleneck removed by **mTCP**

1) Efficient use of CPU cycles for TCP/IP processing
→ 2.35x more CPU cycles for app
2) 3x ~ 25x better performance

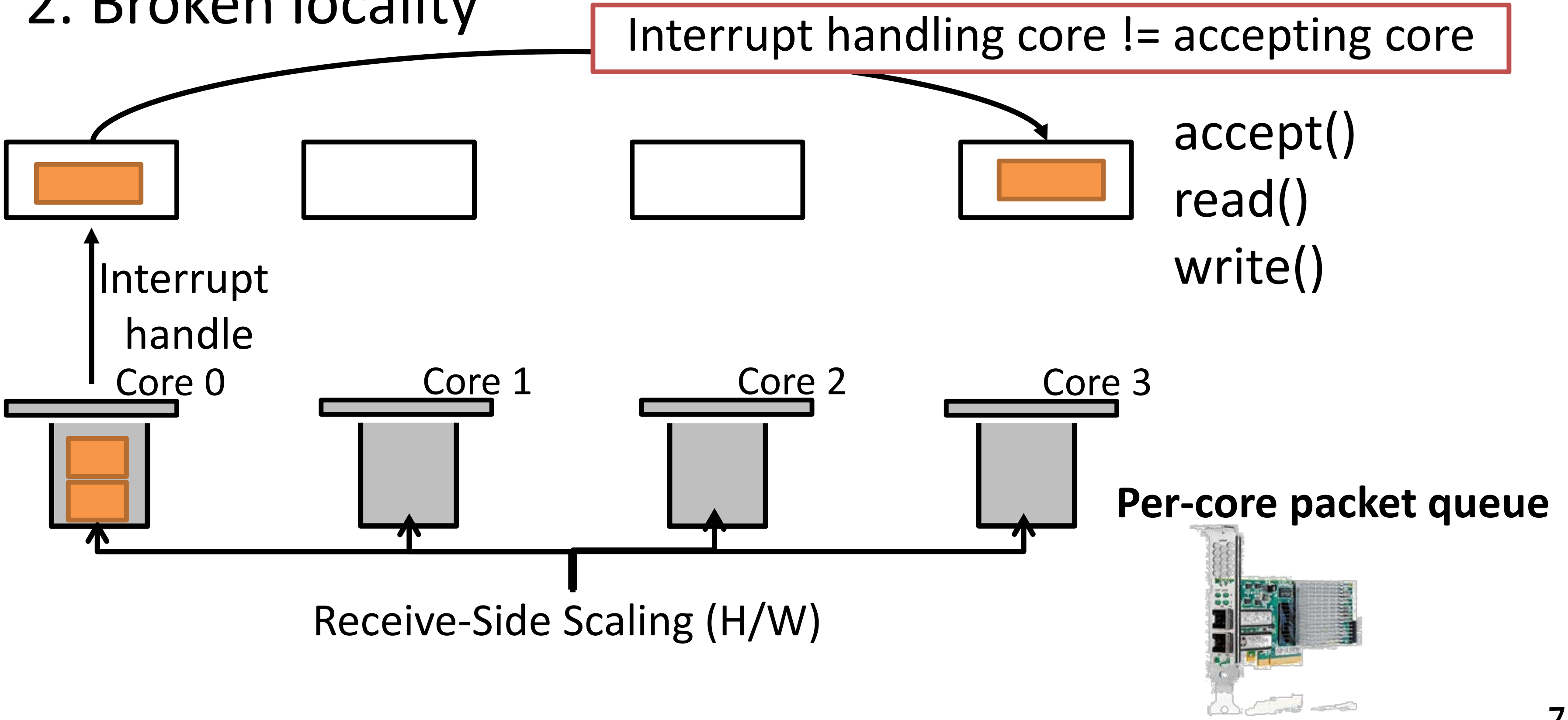
Shared File Descriptors Increase Contention

1. Shared resources
   - Shared listening queue
   - Shared file descriptor space



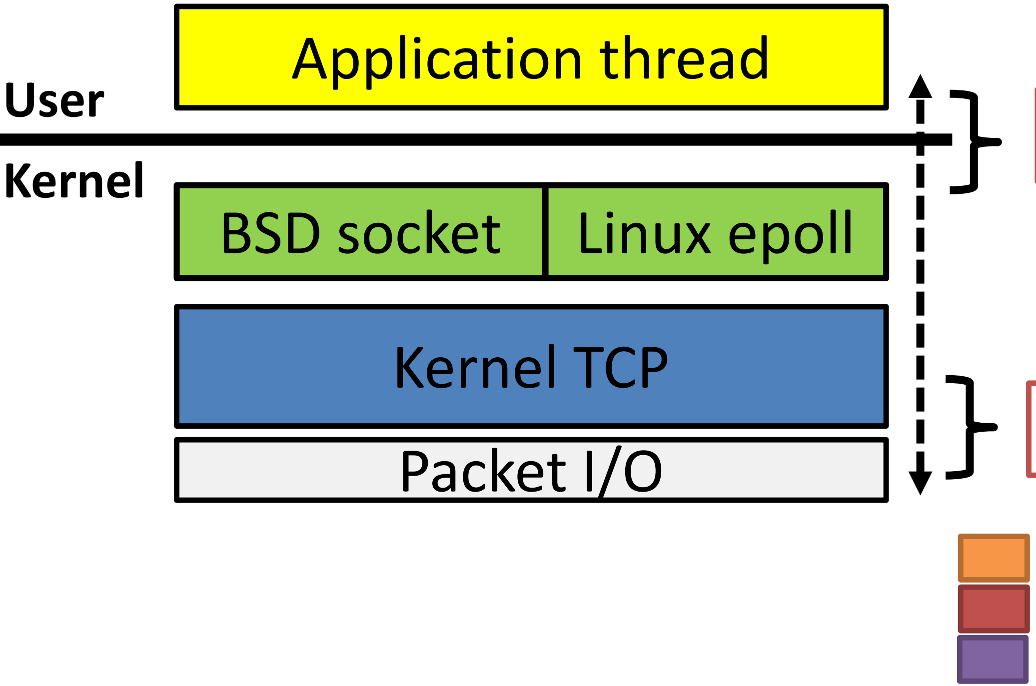
Broken CPU Cache Locality

2. Broken locality



Lack of Support for Batch Processing

3. Per packet, per system call processing



accept(), read(), write()

Inefficient per system call processing

Frequent mode switching

Cache pollution

Inefficient per packet processing

Per packet memory allocation

Previous Works on Reducing Kernel Bottleneck

	Listening queue	Connection locality	App <-> TCP comm.	Packet I/O	API
Linux-2.6	Shared	No	Per system call	Per packet	BSD
Linux-3.9 SO_REUSEPORT	**Per-core**	No	Per system call	Per packet	BSD
Affinity-Accept	**Per-core**	**Yes**	Per system call	Per packet	BSD
MegaPipe	**Per-core**	**Yes**	**Batched system call**	Per packet	**custom**

Still, 78% of CPU cycles are used in kernel!

How much **performance improvement** can we get
if we implement **a user-level TCP stack** with all optimizations?

Clean-slate Design for Fast TCP Processing

- mTCP: A high-performance user-level TCP designed for multicore systems
- Clean-slate approach to divorce kernel's complexity

Problems

1. Shared resources
2. Broken locality
3. Lack of support for batching

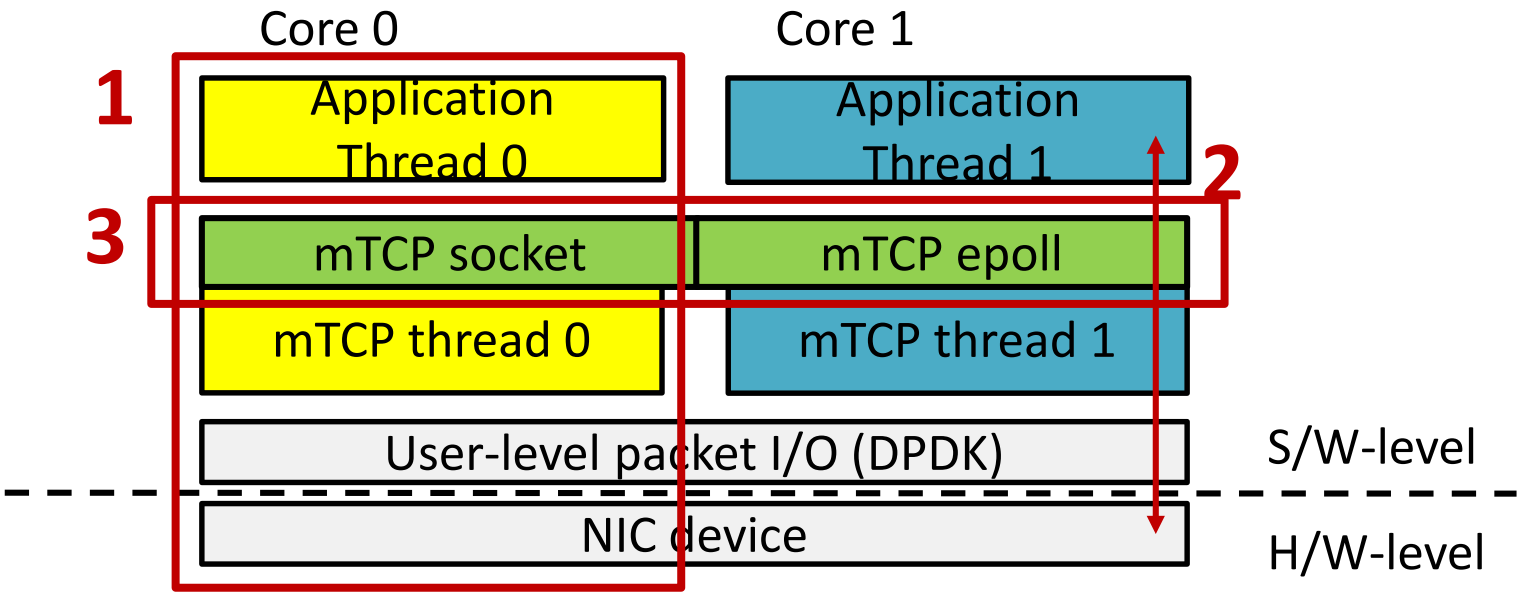
Our contributions

Each core works independently
- No shared resources
- Resources affinity

Batching from flow processing from packet I/O to user API

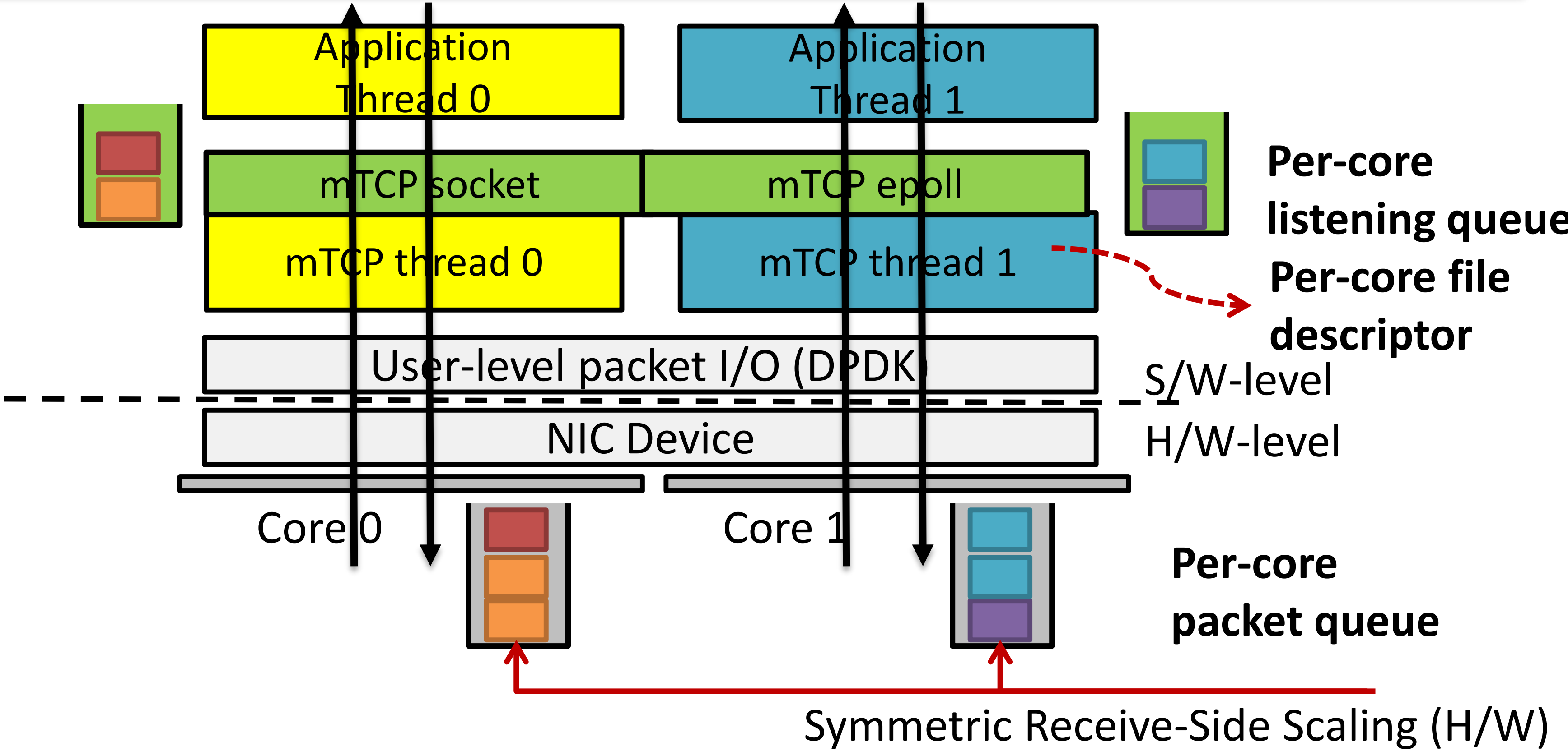
Easily portable APIs for compatibility

Overview of mTCP Stack

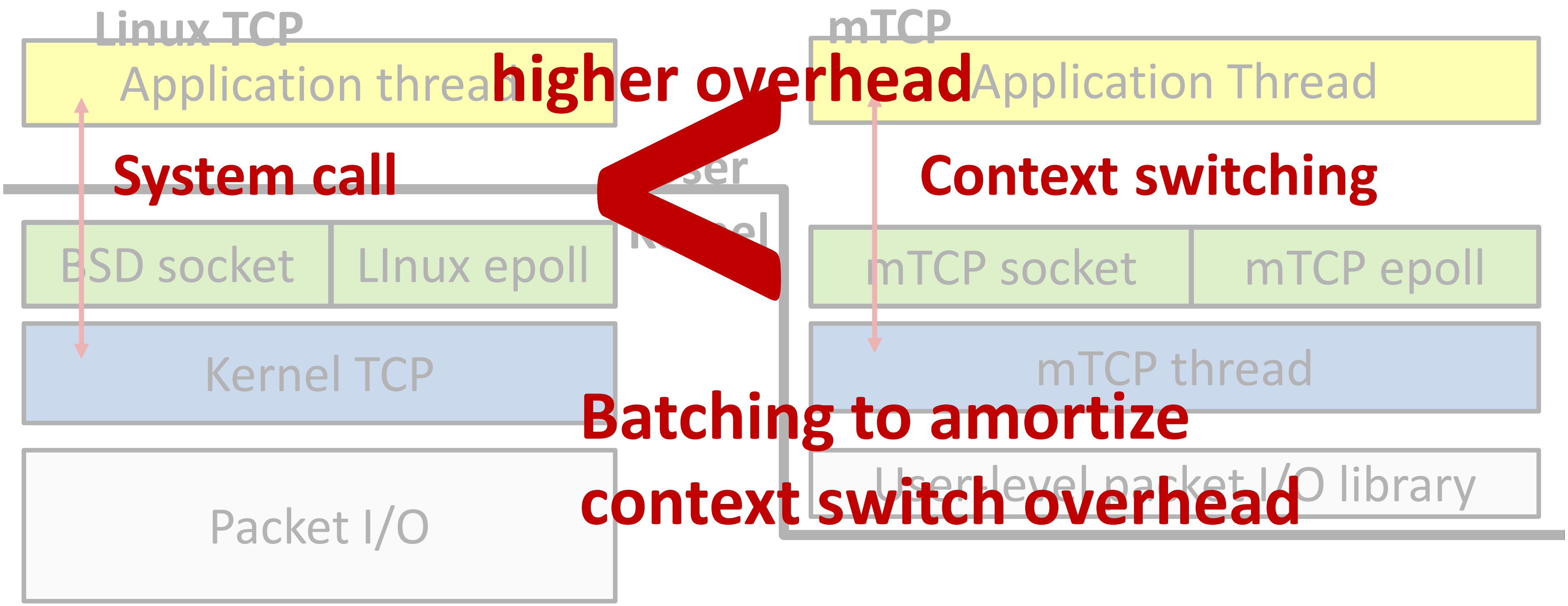


1. Thread model: Pairwise, per-core threading
2. Batching from packet I/O to application
3. mTCP API: Easily portable API (BSD-like)

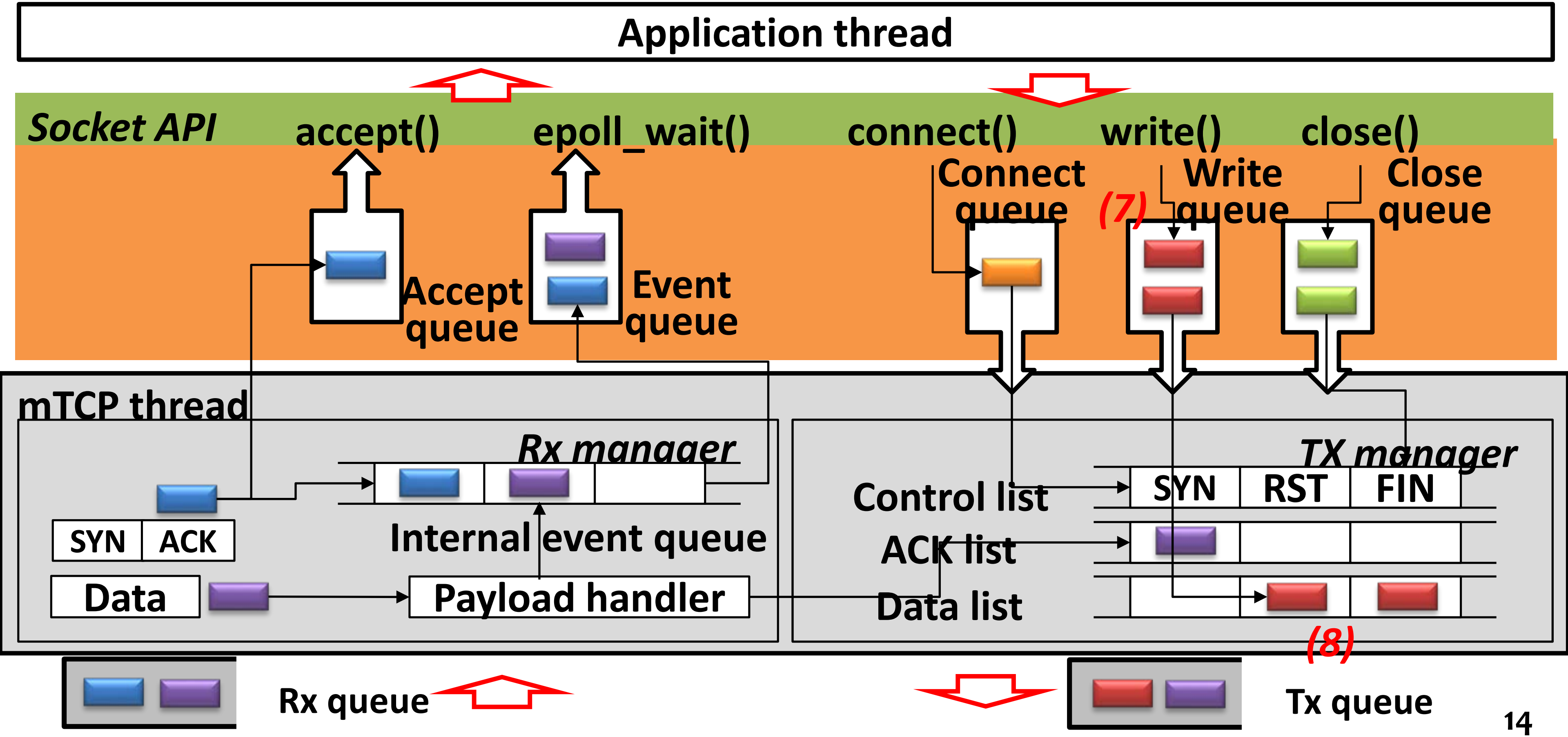
1. Thread Model: Pairwise, Per-core Threading



System Calls to Context Switching?



2. Exploiting Batched Event/Packet Processing



3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append “mtcp_” to BSD socket API
  - **socket() → mtcp_socket()**, **accept() → mtcp_accept()**, etc.
- Event notification: Readiness model using **epoll()**
- Porting existing applications
  - Mostly less than 100 lines of code change

Application	Description	Modified lines / Total lines
Lighttpd	An event-driven web server	**65** / 40K

3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append "mtcp_" to BSD socket API
  - **socket() → mtcp_socket()**, **accept() → mtcp_accept()**, etc.
- Event notification: Readiness model using **epoll()**
- Porting existing applications
  - Mostly less than 100 lines of code change

Application	Description	Modified lines / Total lines
Lighttpd	An event-driven web server	**65** / 40K
ApacheBench	A webserver performance benchmark tool	**29** / 66K

3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append “mtcp_” to BSD socket API
  - **socket() → mtcp_socket()**, **accept() → mtcp_accept()**, etc.
- Event notification: Readiness model using **epoll()**
- Porting existing applications
  - Mostly less than 100 lines of code change

Application	Description	Modified lines / Total lines
Lighttpd	An event-driven web server	**65** / 40K
ApacheBench	A webserver performance benchmark tool	**29** / 66K
SSLShader	A GPU-accelerated SSL proxy [NSDI ’11]	**43** / 6,618

3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append "mtcp_" to BSD socket API
  - **socket() → mtcp_socket()**, **accept() → mtcp_accept()**, etc.
- Event notification: Readiness model using **epoll()**
- Porting existing applications
  - Mostly less than 100 lines of code change

Application	Description	Modified lines / Total lines
Lighttpd	An event-driven web server	**65** / 40K
ApacheBench	A webserver performance benchmark tool	**29** / 66K
SSLShader	A GPU-accelerated SSL proxy [NSDI '11]	**43** / 6,618
WebReplay	A web log replayer	**81** / 3,366

mTCP Socket API

```
/* socket creation, bind, listen functions */
int  mtcp_socket(mctx_t mctx, int domain, int type, int protocol);
int mtcp_bind(mctx_t mctx, int sockid,  const struct sockaddr *addr, socklen_t addrlen);
int mtcp_listen(mctx_t mctx, int sockid, int backlog);

/* accept and connect */
int mtcp_accept(mctx_t mctx, int sockid, struct sockaddr *addr, socklen_t *addrlen);
int mtcp_connect(mctx_t mctx, int sockid, const struct sockaddr *addr, socklen_t addrlen);

/* read functions */
int mtcp_read(mctx_t mctx, int sockid, char *buf, int len);
int mtcp_readv(mctx_t mctx, int sockid, struct iovec *iov, int numIOV);

/* write functions */
int mtcp_write(mctx_t mctx, int sockid, char *buf, int len);
int mtcp_writev(mctx_t mctx, int sockid, struct iovec *iov, int numIOV);

/* socket closure */
int  mtcp_close(mctx_t mctx, int sockid);
int  mtcp_abort(mctx_t mctx, int sockid);

/* rss queue mapping */
int mtcp_init_rss(mctx_t mctx, in_addr_t saddr_base, int num_addr,  in_addr_t daddr, in_addr_t dport);
```

mTCP Socket API

```
/* configuration file reading */
int mtcp_init(char *config_file);
void mtcp_destroy();

int  mtcp_getconf(struct mtcp_conf *conf);
int  mtcp_setconf(const struct mtcp_conf *conf);
int  mtcp_core_affinitize(int cpu);

/* thread context manipulation */
mctx_t mtcp_create_context(int cpu);
void mtcp_destroy_context(mctx_t mctx);

typedef void (*mtcp_sighandler_t)(int);
mtcp_sighandler_t mtcp_register_signal(int signum, mtcp_sighandler_t handler);

/* pipe, getsock/setsockopt, set fd non-blocking mode */
int  mtcp_pipe(mctx_t mctx, int pipeid[2]);
int  mtcp_getsockopt(mctx_t mctx, int sockid, int level, int optname, void *optval, socklen_t *optlen);
int  mtcp_setsockopt(mctx_t mctx, int sockid, int level,  int optname, const void *optval, socklen_t
optlen);
int mtcp_setsock_nonblock(mctx_t mctx, int sockid);

/* mtcp_socket_ioctl: similar to ioctl,  but only FIONREAD is supported currently */
int  mtcp_socket_ioctl(mctx_t mctx, int sockid, int request, void *argp);
```

Sample Server Code

```
static void thread_init(mctx_t mctx)
{
  int sock, lsock, ep, i; /* init declarations */
  struct sockaddr_in saddr; struct mtcp_epoll_event ev, events[MAX_EVENTS];

  /* create listening socket */
  lsock = mtcp_socket(mctx, AF_INET, SOCK_STREAM, 0);
  /* bind and listen to a specific port */
  saddr.sin_family = AF_INET; saddr.sin_addr = INADDR_ANY; saddr.sin_port = 80;
  mtcp_bind(mctx, lsock, (struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
  mtcp_listen(mctx, lsock, 4096);
  /* create epoll queue & enlist listening port in epoll queue */
  ep = mtcp_epoll_create(mctx, MAX_EVENTS);
  ev.events = MTCP_EPOLLIN; ev.data.sockid = lsock;
  mtcp_epoll_ctl(mctx, ep, MTCP_EPOLL_CTL_ADD, lsock, &ev);

  while (1) {
    int nevents = mtcp_epoll_wait(mctx, ep, events, MAX_EVENTS, -1);
    for (i = 0; i < nevents; i++) {
      if (events[i].data.sockid == lsock) {
               sock = mtcp_accept(mctx, lsock, NULL, NULL);
               ...
      } else if (events[i].events == MTCP_EPOLLIN) {
        mtcp_read(mctx, ...);
        ...
      }
    }
  }
}
```

Sample Client Code

```
static void thread_init(mctx_t mctx)
{
  int sock, lsock, ep, i; /* init declarations */
  struct in_addr_t saddr, daddr; struct in_port_t sport, dport;
  struct mtcp_epoll_event ev, events[MAX_EVENTS];

  saddr = INADDR_ANY; daddr = inet_addr(DHOST_IP);
  dport = htons(80);
  /* intialize per-thread client-port RSS pool */
  mtcp_init_rss(mctx, saddr, 1, daddr, dport);
  ep = mtcp_epoll_create(mctx, MAX_EVENTS);

  while (1) {
    if (connection_count < conn_thresh)
      CreateConnection(mctx, ep); /* mtcp_connect() */
    int nevents = mtcp_epoll_wait(mctx, ep, events, MAX_EVENTS, -1);
    for (i = 0; i < nevents; i++) {
      if (events[i].events & MTCP_EPOLLIN)
        HandleReadEvent(...); /* mtcp_read() */
      ...
    }
  }
}
```

mTCP Implementation

- 12,727 lines in C code
  - Packet I/O, TCP flow management, user-level socket API, event system library
- Supports Intel DPDK
  - Fast packet I/O library + event-driven packet I/O
  - Originally based on PacketShader IOEngine [SIGCOMM'10]
- TCP protocol conformance
  - Follows RFC793
  - Congestion control algorithm: NewReno
- Passing correctness test and stress test with Linux TCP stack

Evaluation

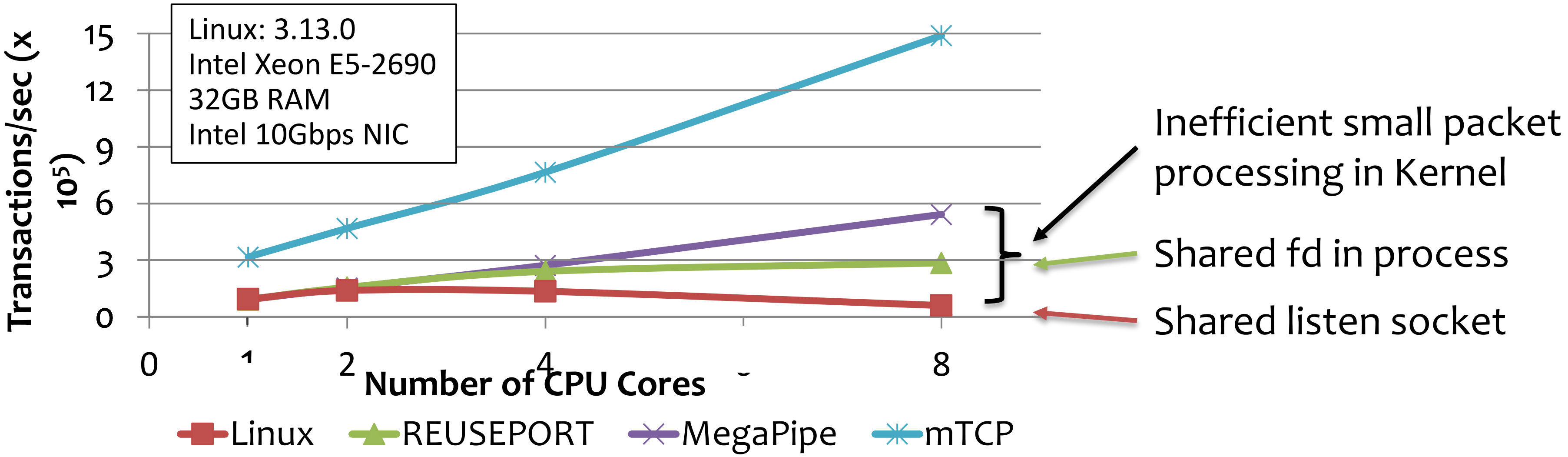
- Does performance scale over CPU cores?
  - Performance comparison with previous solutions
- Does it improve the performance of real applications?
  - Web server (Lighttpd)
    - Performance under the real workload
  - SSL proxy (SSL Shader, NSDI 11)
    - GPU acceleration on crypto algorithms (RSA, AES, SHA1)
    - Bottlenecked at TCP stack
- Third party evaluation
  - HAProxy port to mTCP
  - nginx port to mTCP

Evaluation Setup

- Client – server HTTP transactions
- Server specification
  - One Xeon E5-2690 CPU (8 cores), 2.90 GHz
  - 32 GB RAM, 1 x 10G NIC (Intel 82599 chipset)
- Clients: 5 x machines with the same spec with server

Multicore Scalability

- 64B ping/pong messages per connection (Connections/sec)
- Heavy connection overhead, small packet processing overhead
- 25x Linux, 5x SO_REUSEPORT*[LINUX3.9], 3x MegaPipe*[OSDI'12]

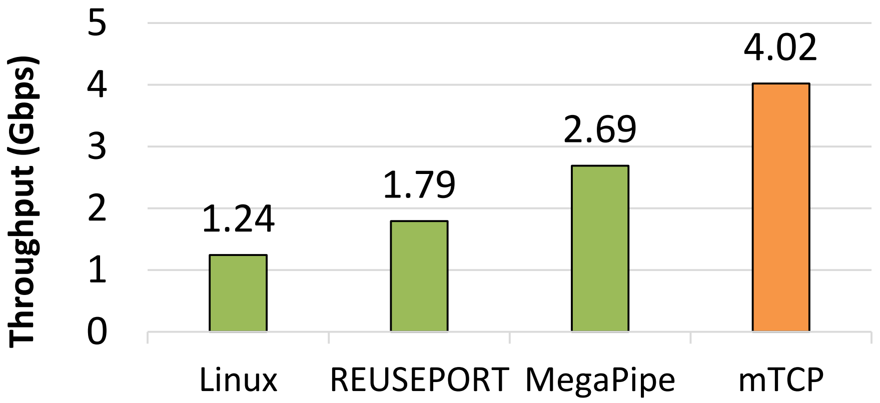


* [LINUX3.9] https://lwn.net/Articles/542629/
* [OSDI'12] MegaPipe: A New Programming Interface for Scalable Network I/O, Berkeley

Performance Improvement on Real Applications

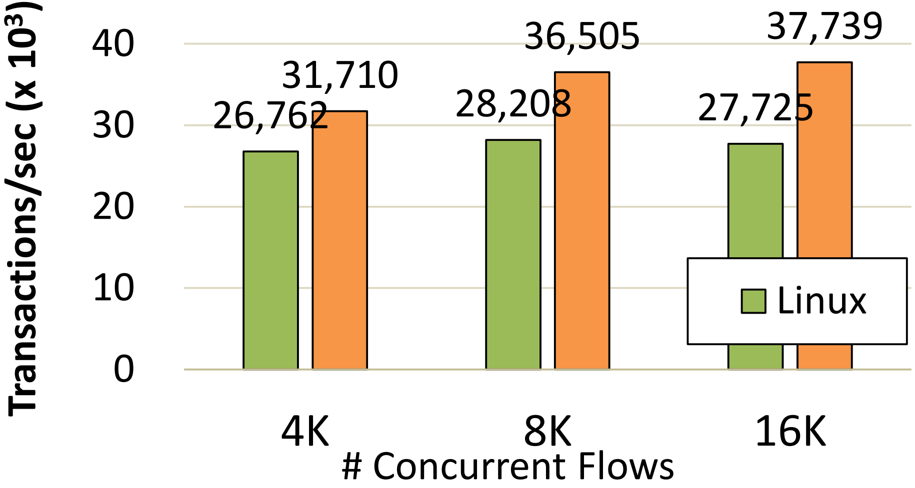
Web Server (Lighttpd)

- Real traffic workload: Static file workload from SpecWeb2009 set
- **3.2x** faster than Linux
- **1.5x** faster than MegaPipe



SSL Proxy (SSLShader)

- Performance Bottleneck in TCP
- Cipher suite
  1024-bit RSA, 128-bit AES, HMAC-SHA1
- Download 1-byte object via HTTPS

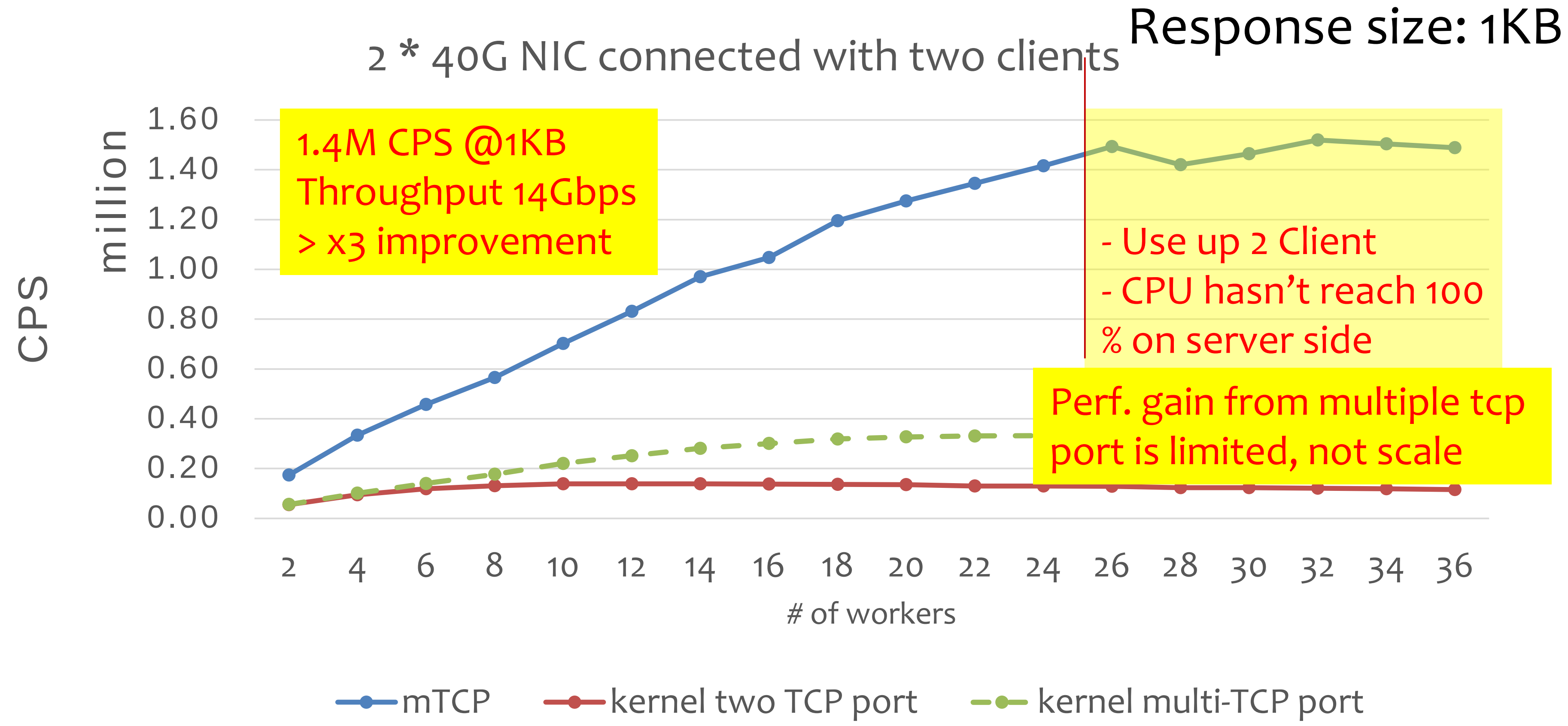


Third Party Evaluation

- Third-party company 1
- Ported HAProxy to mTCP
- Saw 2x performance improvement for small-file transactions

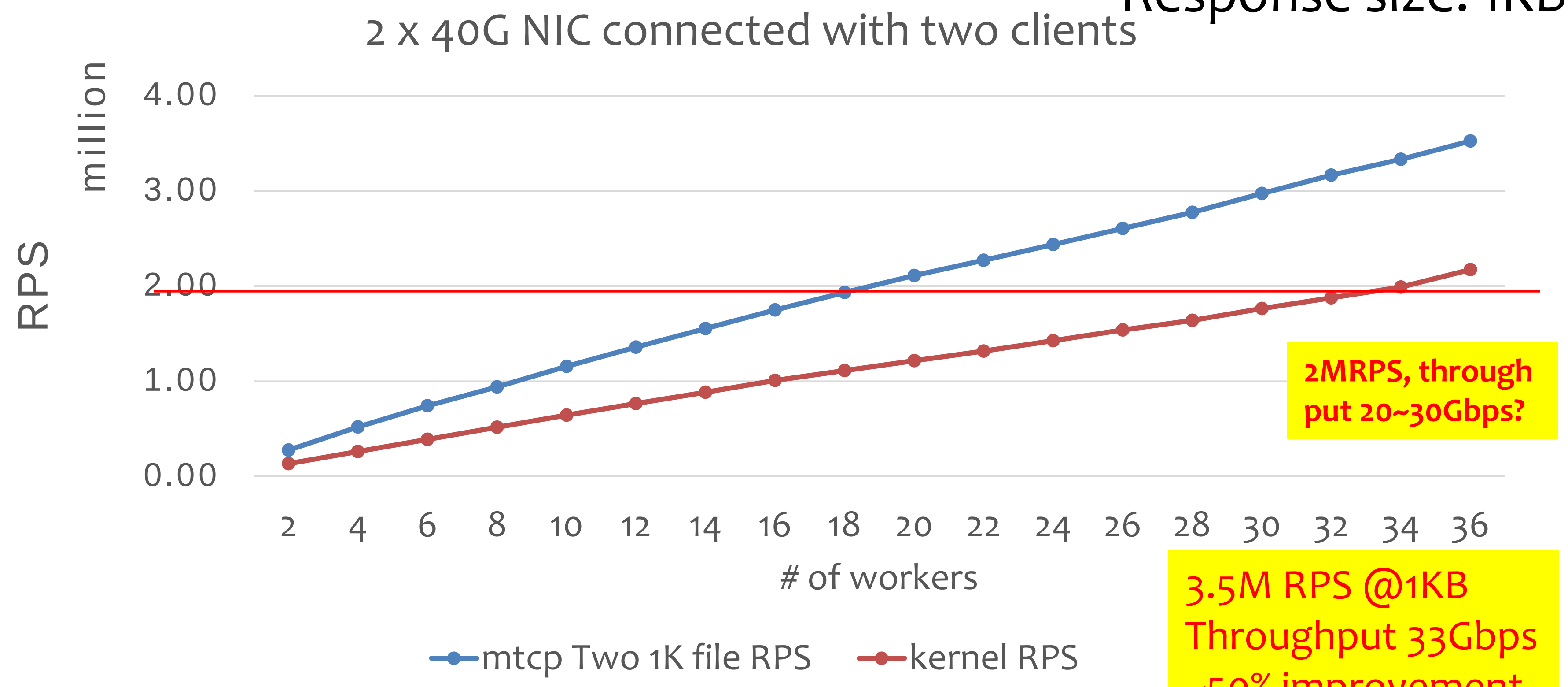
- Third-party company 2
- Ported nginx to mTCP
- Experiment setup
  - Server: 2 x Xeon E5-2699 v3 (36 cores in total)
    - 32 GB memory
    - 2 x 40G Intel NICs
    - DPDK: v2.2RC1
    - nginx release 1.9.6
  - Client: same spec with the server
- We plan to merge the patches into mTCP github

nginx HTTP Connections per Second (CPS)



nginx HTTP Plaintext Responses per sec (RPS)

Response size: 1KB



2MRPS, through put 20~30Gbps?

3.5M RPS @1KB
Throughput 33Gbps
~50% improvement

mTCP features Under Development

- Now supports multi processes
  - Easily port single-threaded network applications
  - Global variables => possible race conditions
  - HAProxy, Lighttpd *etc.*
- TCP Segmentation Offlaoding (TSO) patch ready
  - Third-party contribution
- Virtualization support
  - virtio work for container
- Zero copy I/O
  - DPDK buffer visible to stack
- Beyond end-host TCP stack?
  - mOS networking stack

The need for Resusable Middlebox Networking Stack

- Developing stateful middlebox is non-trivial
  - Typical IDS code ~= 135K SLOC
  - TCP flow management is complex and error prone

- Typical TCP applications

TCP application	User level
Berkeley Socket API	
TCP/IP stack	Kernel level

- Typical middleboxes?
  - Middlebox logic
  - Packet processing
  - Flow management
  - Spaghetti code?

  No layered abstraction!

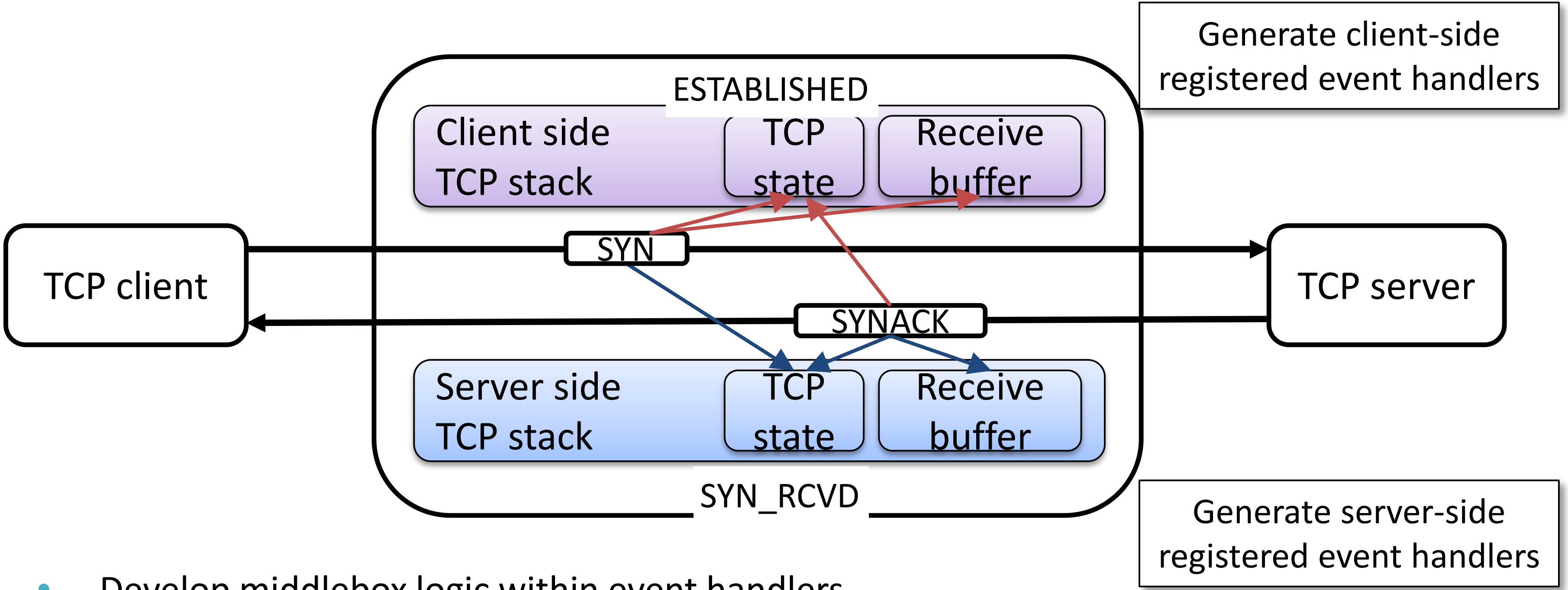
- Why not Berkeley-socket API for middleboxes?
  - Nice abstraction: separates flow management from application
  - Knowledge of internal TCP implementation <u>not needed</u>

mOS Middlebox Networking stack

- Networking stack specialization for middleboxes
  - Abstraction for sub-TCP layer middlebox operations
- Key concepts
  - Separation of flow management from custom middlebox app logic
  - User-defined event definition and generation
- Benefits
  - Clean, modular development of stateful middleboxes
  - Developers focus on core logic only
    - Reuse a networking stack for TCP flow management
  - High performance from mTCP implementation
    - Optimized for multicore systems
    - Fast packet/TCP processing on DPDK

Middlebox Stack Monitors Both Ends

- Dual mTCP stack management
- ***Infer*** the states of both client and server TCP stacks



- <u>Develop middlebox logic within event handlers</u>

Conclusion

- mTCP: A high-performing user-level TCP stack
  for multicore systems
  - Clean-slate user-level design to overcome inefficiency in kernel
- Make full use of extreme parallelism & batch processing
  - Per-core resource management
  - Lock-free data structures & cache-aware threading
  - Eliminate system call overhead
  - Reduce context switch cost by event batching
- Achieve high performance scalability
  - Small message transactions: **3x to 25x better**
  - Existing applications: 33% (SSLShader) to 320% (lighttpd)

Thank You

Source code is available @

http://shader.kaist.edu/mtcp/

https://github.com/eunyoung14/mtcp