mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems

EunYoung Jeong, Shinae Woo, Muhammad Jamshed, and Haewon Jeong, ***Korea Advanced Institute of Science and Technology (KAIST);*** **Sunghwan Ihm,** ***Princeton University;*** **Dongsu Han and KyoungSoo Park,** ***Korea Advanced Institute of Science and Technology (KAIST)***

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems

EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong
Sunghwan Ihm*, Dongsu Han, and KyoungSoo Park

KAIST *Princeton University

Abstract

Scaling the performance of short TCP connections on multicore systems is fundamentally challenging. Although many proposals have attempted to address various shortcomings, inefficiency of the kernel implementation still persists. For example, even state-of-the-art designs spend 70% to 80% of CPU cycles in handling TCP connections in the kernel, leaving only small room for innovation in the user-level program.

This work presents mTCP, a high-performance user-level TCP stack for multicore systems. mTCP addresses the inefficiencies from the ground up—from packet I/O and TCP connection management to the application interface. In addition to adopting well-known techniques, our design (1) translates multiple expensive system calls into a single shared memory reference, (2) allows efficient flow-level event aggregation, and (3) performs batched packet I/O for high I/O efficiency. Our evaluations on an 8-core machine showed that mTCP improves the performance of small message transactions by a factor of 25 compared to the latest Linux TCP stack and a factor of 3 compared to the best-performing research system known so far. It also improves the performance of various popular applications by 33% to 320% compared to those on the Linux stack.

1 Introduction

Short TCP connections are becoming widespread. While large content transfers (*e.g.*, high-resolution videos) consume the most bandwidth, short "transactions" [1] dominate the number of TCP flows. In a large cellular network, for example, over 90% of TCP flows are smaller than 32 KB and more than half are less than 4 KB [45].

Scaling the processing speed of these short connections is important not only for popular user-facing online services [1, 2, 18] that process small messages. It is also critical for backend systems (*e.g.*, memcached clusters [36]) and middleboxes (*e.g.*, SSL proxies [32] and redundancy elimination [31]) that must process TCP connections at high speed. Despite recent advances in software packet processing [4, 7, 21, 27, 39], supporting high TCP transaction rates remains very challenging. For example, Linux TCP transaction rates peak at about 0.3 million transactions per second (shown in Section 5), whereas packet I/O can scale up to tens of millions packets per second [4, 27, 39].

Prior studies attribute the inefficiency to either the high system call overhead of the operating system [28, 40, 43] or inefficient implementations that cause resource contention on multicore systems [37]. The former approach drastically changes the I/O abstraction (*e.g.*, socket API) to amortize the cost of system calls. The practical limitation of such an approach, however, is that it requires significant modifications within the kernel and forces existing applications to be re-written. The latter one typically makes incremental changes in existing implementations and, thus, falls short in fully addressing the inefficiencies.

In this paper, we explore an alternative approach that delivers high performance without requiring drastic changes to the existing code base. In particular, we take a clean-slate approach to assess the performance of an untethered design that divorces the limitation of the kernel implementation. To this end, we build a user-level TCP stack from the ground up by leveraging high-performance packet I/O libraries that allow applications to directly access the packets. Our user-level stack, mTCP, is designed for three explicit goals:

1. Multicore scalability of the TCP stack.
2. Ease of use (i.e., application portability to mTCP).
3. Ease of deployment (i.e., no kernel modifications).

Implementing TCP in the user level provides many opportunities. In particular, it can eliminate the expensive system call overhead by translating syscalls into inter-process communication (IPC). However, it also in-

[1] We refer to a request-response pair as a transaction. These transactions are typically small in size.

	Accept queue	Conn. Locality	Socket API	Event Handling	Packet I/O	Application Modification	Kernel Modification
PSIO [12], DPDK [4], PF_RING [7], netmap [21]	No TCP stack				**Batched**	No interface for transport layer	**No (NIC driver)**
Linux-2.6	Shared	None	BSD socket	Syscalls	Per packet	**Transparent**	**No**
Linux-3.9	**Per-core**	None	BSD socket	Syscalls	Per packet	Add option SO_REUSEPORT	**No**
Affinity-Accept [37]	**Per-core**	**Yes**	BSD socket	Syscalls	Per packet	**Transparent**	Yes
MegaPipe [28]	**Per-core**	**Yes**	lwsocket	**Batched syscalls**	Per packet	Event model to completion I/O	Yes
FlexSC [40], VOS [43]	Shared	None	BSD socket	**Batched syscalls**	Per packet	Change to use new API	Yes
mTCP	**Per-core**	**Yes**	**User-level socket**	**Batched function calls**	**Batched**	Socket API to mTCP API	**No (NIC driver)**

Table 1: Comparison of the benefits of previous work and mTCP.

troduces fundamental challenges that must be addressed—processing IPC messages, including shared memory messages, involve context-switches that are typically much more expensive than the system calls themselves [3, 29].

Our key approach is to amortize the context-switch overhead over a batch of packet-level and socket-level events. While packet-level batching [27] and system-call batching [28, 40, 43] (including socket-level events) have been explored individually, integrating the two requires a careful design of the networking stack that translates packet-level events to socket-level events and vice-versa.

This paper makes two key contributions:

First, we demonstrate that significant performance gain can be obtained by integrating packet- and socket-level batching. In addition, we incorporate all known optimizations, such as per-core listen sockets and load balancing of concurrent flows on multicore CPUs with receive-side scaling (RSS). The resulting TCP stack outperforms Linux and MegaPipe [28] by up to 25x (w/o `SO_REUSEPORT`) and 3x, respectively, in handling TCP transactions. This directly translates to application performance; mTCP increases existing applications' performance by 33% (SSLShader) to 320% (lighttpd).

Second, unlike other designs [23,30], we show that such integration can be done purely at the user level in a way that ensures ease of porting without requiring significant modifications to the kernel. mTCP provides BSD-like socket and epoll-like event-driven interfaces. Migrating existing event-driven applications is easy since one simply needs to replace the socket calls to their counterparts in mTCP (*e.g.*, `accept()` becomes `mtcp_accept()`) and use the per-core listen socket.

2 Background and Motivation

We first review the major inefficiencies in existing TCP implementations and proposed solutions. We then discuss our motivation towards a user-level TCP stack.

2.1 Limitations of the Kernel's TCP Stack

Recent studies proposed various solutions to address four major inefficiencies in the Linux TCP stack: lack of connection locality, shared file descriptor space, inefficient packet processing, and heavy system call overhead [28].

Lack of connection locality: Many applications are multi-threaded to scale their performance on multicore systems. However, they typically share a listen socket that accepts incoming connections on a well-known port. As a result, multiple threads contend for a lock to access the socket's accept queue, resulting in a significant performance degradation. Also, the core that executes the kernel code for handling a TCP connection may be different from the one that runs the application code that actually sends and receives data. Such lack of connection locality introduces additional overhead due to increased CPU cache misses and cache-line sharing [37].

Affinity-Accept [37] and MegaPipe [28] address this issue by providing a local accept queue in each CPU core and ensuring flow-level core affinity across the kernel and application thread. Recent Linux kernel (3.9.4) also partly addresses this by introducing the `SO_REUSEPORT` [14] option, which allows multiple threads/processes to bind to the same port number.

Shared file descriptor space: In POSIX-compliant operating systems, the file descriptor (fd) space is shared within a process. For example, Linux searches for the minimum available fd number when allocating a new socket. In a busy server that handles a large number of concurrent connections, this incurs significant overhead due to lock contention between multiple threads [20]. The use of file descriptors for sockets, in turn, creates extra overhead of going through the Linux Virtual File System (VFS), a pseudo-filesystem layer for supporting common file operations. MegaPipe eliminates this layer for sockets by explicitly partitioning the fd space for sockets and regular files [28].

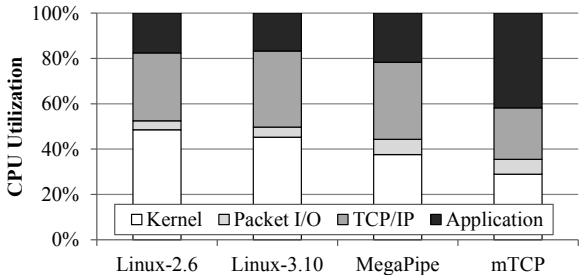


Figure 1: CPU usage breakdown when running `lighttpd` serving a 64B file per connection.

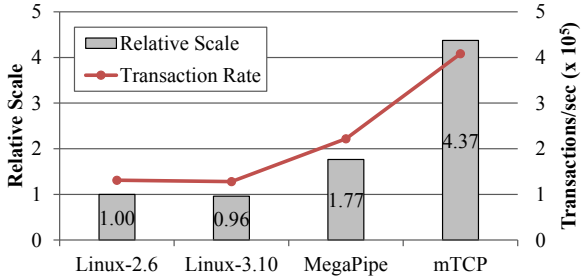


Figure 2: Relative scale of # transactions processed per CPU cycle in the kernel (including TCP/IP and I/O) across four `lighttpd` versions.

Inefficient per-packet processing: Previous studies indicate per-packet memory (de)allocation and DMA overhead, NUMA-unaware memory access, and heavy data structures (*e.g.*, `sk_buff`) as the main bottlenecks in processing small packets [27, 39]. To reduce the per-packet overhead, it is essential to batch process multiple packets. While many recent user-level packet I/O libraries [4, 7, 27, 39] address these problems, these libraries do not provide a full-fledged TCP stack, and not all optimizations are incorporated into the kernel.

System call overhead: The BSD socket API requires frequent user/kernel mode switching when there are many short-lived concurrent connections. As shown in FlexSC [40] and VOS [43], frequent system calls can result in processor state (*e.g.*, top-level caches, branch prediction table, etc.) pollution that causes performance penalties. Previous solutions propose system call batching [28, 43] or efficient system call scheduling [40] to amortize the cost. However, it is difficult to readily apply either approach to existing applications since they often require user and/or kernel code modification due to the changes to the system call interface and/or its semantics.

Table 1 summarizes the benefits provided by previous work compared to a vanilla Linux kernel. Note that there is not a single system that provides all of the benefits.

2.2 Why User-level TCP?

While many previous designs have tried to scale the performance of TCP in multicore systems, few of them truly overcame the aforementioned inefficiencies of the kernel. This is evidenced by the fact that even the best-performing system, MegaPipe, spends a dominant portion of CPU cycles (∼80%) inside the kernel. Even more alarming is the fact that these CPU cycles are not utilized efficiently; according to our own measurements, Linux spends more than 4x the cycles (in the kernel and the TCP stack combined) than mTCP does while handling the same number of TCP transactions.

To reveal the significance of this problem, we profile the server's CPU usage when it is handling a large number of concurrent TCP transactions (8K to 48K concurrent TCP connections). For this experiment, we use a simple web server (`lighttpd` v1.4.32 [8]) running on an 8-core Intel Xeon CPU (2.90 GHz, E5-2690) with 32 GB of memory and a 10 Gbps NIC (Intel 82599 chipsets). Our clients use `ab` v2.3 [15] to repeatedly download a 64B file per connection. Multiple clients are used in our experiment to saturate the CPU utilization of the server. Figure 1 shows the breakdown of CPU usage comparing four versions of the lighttpd server: a multithreaded version that harnesses all 8 CPU cores on Linux 2.6.32 and 3.10.12 [2] (Linux), a version ported to MegaPipe [3] (MegaPipe), and a version using mTCP, our user-level TCP stack, on Linux 2.6.32 (mTCP). Note that MegaPipe adopts all recent optimizations such as per-core accept queues and file descriptor space, as well as user-level system call batching, but reuses the existing kernel for packet I/O and TCP/IP processing.

Our results indicate that Linux and MegaPipe spend 80% to 83% of CPU cycles in the kernel which leaves only a small portion of the CPU to user-level applications. Upon further investigation, we find that lock contention for shared in-kernel data structures, buffer management, and frequent mode switch are the main culprits. This implies that the kernel, including its stack, is the major bottleneck. Furthermore, the results in Figure 2 show that the CPU cycles are not spent efficiently in Linux and MegaPipe. The bars indicate the relative number of transactions processed per each CPU cycle inside the kernel and the TCP stack (*e.g.*, outside the application), normalized by the performance of Linux 2.6.32. We find that mTCP uses the CPU cycles 4.3 times more effectively than Linux. As a result, mTCP achieves 3.1x and 1.8x the performance of Linux 2.6 and MegaPipe, respectively, while using fewer CPU cycles in the kernel and the TCP stack.

Now, the motivation of our work is clear. Can we design a user-level TCP stack that incorporates all existing optimizations into a single system and achieve all benefits that individual systems have provided in the past? How much of a performance improvement can we get if we build such a system? Can we bring the performance of existing packet I/O libraries to the TCP stack?

[2] This is the latest Linux kernel version as of this writing.

[3] We use Linux 3.1.3 for MegaPipe due to its patch availability.

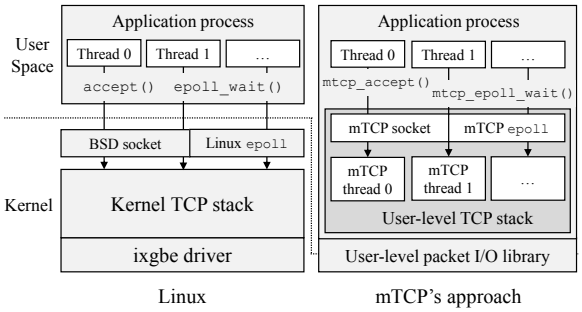


Figure 3: mTCP Design Overview.

To answer these questions, we build a TCP stack in the user level. User-level TCP is attractive for many reasons. First, it allows us to easily depart from the kernel's complexity. In particular, due to shared data structures and various semantics that the kernel has to support (*e.g.*, POSIX and VFS), it is often difficult to separate the TCP stack from the rest of the kernel. Furthermore, it allows us to directly take advantage of the existing optimizations in the high-performance packet I/O library, such as netmap [39] and Intel DPDK [4]. Second, it allows us to apply batch processing as the first principle, harnessing the ideas in FlexSC [40] and VOS [43] without extensive kernel modifications. In addition to performing batched packet I/O, the user-level TCP naturally collects multiple flow-level events to and from the user application (*e.g.*, `connect()`/`accept()` and `read()`/`write()` for different connections) without the overhead of frequent mode switching in system calls. Finally, it allows us to easily preserve the existing application programming interface. Our TCP stack is backward-compatible in that we provide a BSD-like socket interface.

3 Design

The goal of mTCP is to achieve high scalability on multicore systems while maintaining backward compatibility to existing multi-threaded, event-driven applications. Figure 3 presents an overview of our system. At the highest level, applications link to the mTCP library, which provides a socket API and an event-driven programming interface for backward compatibility. The two underlying components, user-level TCP stack and packet I/O library, are responsible for achieving high scalability. Our user-level TCP implementation runs as a thread on each CPU core within the same application process. The mTCP thread directly transmits and receives packets to and from the NIC using our custom packet I/O library. Existing user-level packet libraries only allow one application to access an NIC port. Thus, mTCP can only support one application per NIC port. However, we believe this can be addressed in the future using virtualized network interfaces (more details in Section 3.3). Applications can still choose to work with the existing TCP stack, provided that they only use NICs that are not used by mTCP.

In this section, we first present the design of mTCP's highly scalable lower-level components in Sections 3.1 and 3.2. We then discuss the API and semantics that mTCP provides to support applications in Section 3.3.

3.1 User-level Packet I/O Library

Several packet I/O systems allow high-speed packet I/O ($\sim$100M packets/sec) from a user-level application [4, 7, 12]. However, they are not suitable for implementing a transport layer since their interface is mainly based on polling. Polling can significantly waste precious CPU cycles that can potentially benefit the applications. Furthermore, our system requires efficient multiplexing between TX and RX queues from multiple NICs. For example, we do not want to block a TX queue while sending a data packet when a control packet is waiting to be received. This is because if we block the TX queue, important control packets, such as SYN or ACK, may be dropped, resulting in a significant performance degradation due to retransmissions.

To address these challenges, mTCP extends the PacketShader I/O engine (PSIO) [27] to support an efficient *event-driven* packet I/O interface. PSIO offers high-speed packet I/O by utilizing RSS that distributes incoming packets from multiple RX queues by their flows, and provides flow-level core affinity to minimize the contention among the CPU cores. On top of PSIO's high-speed packet I/O, the new event-driven interface allows an mTCP thread to efficiently wait for events from RX and TX queues from multiple NIC ports at a time.

The new event-driven interface, `ps_select()`, works similarly to `select()` except that it operates on TX/RX queues of interested NIC ports for packet I/O. For example, mTCP specifies the interested NIC interfaces for RX and/or TX events with a timeout in microseconds, and `ps_select()` returns immediately if any event of interest is available. If such an event is not detected, it enables the interrupts for the RX and/or TX queues and yields the thread context. Eventually, the interrupt handler in the driver wakes up the thread if an I/O event becomes available or the timeout expires. `ps_select()` is also similar to the `select()`/`poll()` interface supported by netmap [39]. However, unlike netmap, we do not integrate this with the general-purpose event system in Linux to avoid its overhead.

The use of PSIO brings the opportunity to amortize the overhead of system calls and context switches throughout the entire system, in addition to eliminating the per-packet memory allocation and DMA overhead. In PSIO, packets are received and transmitted in batches [27], amortizing the cost of expensive PCIe operations, such as DMA address mapping and IOMMU lookups.

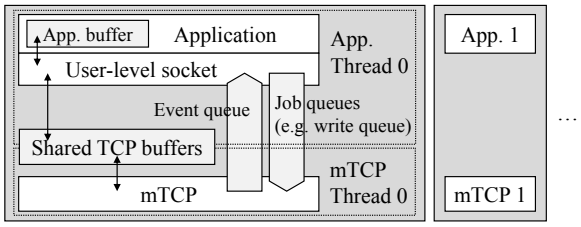


Figure 4: Thread model of mTCP.

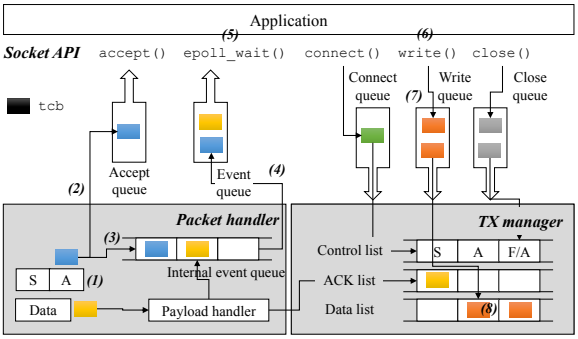


Figure 5: An example of TCP processing in mTCP.

3.2 User-level TCP Stack

A user-level TCP stack naturally eliminates many system calls (*e.g.*, socket I/O), which can potentially reduce a significant part of the Linux TCP overhead. One approach to a user-level TCP stack is to implement it completely as a library that runs as part of the application's main thread. This "zero-thread TCP" could potentially provide the best performance since this translates costly system calls into light-weight user-level function calls. However, the fundamental limitation of this approach is that the correctness of internal TCP processing depends on the timely invocation of TCP functions from the application.

In mTCP, we choose to create a separate TCP thread to avoid such an issue and to minimize the porting effort for existing applications. Figure 4 shows how mTCP interacts with the application thread. The application uses mTCP library functions that communicate with the mTCP thread via shared buffers. The access to the shared buffers is granted only through the library functions, which allows safe sharing of the internal TCP data. When a library function needs to modify the shared data, it simply places a request (*e.g.*, `write()` request) to a job queue. This way, multiple requests from different flows can be piled to the job queue at each loop, which are processed in batch when the mTCP thread regains the CPU. Flow events from the mTCP thread (*e.g.*, new the CPU core. Flow events from the mTCP thread (*e.g.*, new connections, new data arrival, etc.) are delivered in a similar way

This, however, requires additional overhead of managing concurrent data structures and context switch between the application and the mTCP thread. Such cost is unfortunately not negligible, typically much larger than the system call overhead [29]. One measurement on a recent Intel CPU shows that a thread context switch takes 19 times the duration of a null system call [3].

In this section, we describe how mTCP addresses these challenges and achieves high scalability with the user-level TCP stack. We first start from how mTCP processes TCP packets in Section 3.2.1, then present a set of key optimizations we employ to enhance its performance in Sections 3.2.2, 3.2.3, and 3.2.4.

3.2.1 Basic TCP Processing

When the mTCP thread reads a batch of packets from the NIC's RX queue, mTCP passes them to the TCP packet processing logic which follows the standard TCP specification. For each packet, mTCP first searches (or creates) a TCP control block (`tcb`) of the corresponding flow in the flow hash table. As in Figure 5, if a server side receives an ACK for its SYN/ACK packet (1), the `tcb` for the new connection will be enqueued to an accept queue (2), and a read event is generated for the listening socket (3). If a new data packet arrives, mTCP copies the payload to the socket's read buffer and enqueues a read event to an internal event queue. mTCP also generates an ACK packet and keeps it in the ACK list of a TX manager until it is written to a local TX queue.

After processing a batch of received packets, mTCP flushes the queued events to the application event queue (4) and wakes up the application by signaling it. When the application wakes up, it processes multiple events in a single event loop (5), and writes responses from multiple flows without a context switch. Each socket's `write()` call writes data to its send buffer (6), and enqueues its `tcb` to the write queue (7). Later, mTCP collects the `tcb`s that have data to send, and puts them into a send list (8). Finally, a batch of outgoing packets from the list will be sent by a packet I/O system call, transmitting them to the NIC's TX queue.

3.2.2 Lock-free, Per-core Data Structures

To minimize inter-core contention between the mTCP threads, we localize all resources (*e.g.*, flow pool, socket buffers, etc.) in each core, in addition to using RSS for flow-level core affinity. Moreover, we completely eliminate locks by using lock-free data structures between the application and mTCP. On top of that, we also devise an efficient way of managing TCP timer operations.

Thread mapping and flow-level core affinity: We preserve flow-level core affinity in two stages. First, the packet I/O layer ensures to evenly distribute TCP connection workloads across available CPU cores with RSS. This essentially reduces the TCP scalability problem to each core. Second, mTCP spawns one TCP thread for each application thread and co-locates them in the same physical CPU core. This preserves the core affinity of

packet and flow processing, while allowing them to use the same CPU cache without cache-line sharing.

Multi-core and cache-friendly data structures: We keep most data structures, such as the flow hash table, socket id manager, and the pool of `tcb` and socket buffers, local to each TCP thread. This significantly reduces any sharing across threads and CPU cores, and achieves high parallelism. When a data structure must be shared across threads (*e.g.*, between mTCP and the application thread), we keep all data structures local to each core and use lock-free data structures by using a single-producer and single-consumer queue. We maintain write, connect, and close queues, whose requests go from the application to mTCP, and an accept queue where new connections are delivered from mTCP to the application.

In addition, we keep the size of frequently accessed data structures small to maximize the benefit of the CPU cache, and make them aligned with the size of a CPU cache line to prevent any false sharing. For example, we divide `tcb` into two parts where the first-level structure holds 64 bytes of the most frequently-accessed fields and two pointers to next-level structures that have 128 and 192 bytes of receive/send-related variables, respectively.

Lastly, to minimize the overhead of frequent memory allocation/deallocation, we allocate a per-core memory pool for `tcb`s and socket buffers. We also utilize huge pages to reduce the TLB misses when accessing the `tcb`s. Because their access pattern is essentially random, it often causes a large number of TLB misses. Putting the memory pool of `tcb`s and a hash table that indexes them into huge pages reduces the number of TLB misses.

Efficient TCP timer management: TCP requires timer operations for retransmission timeouts, connections in the TIME_WAIT state, and connection keep-alive checks. mTCP provides two types of timers: one managed by a sorted list and another built with a hash table. For coarse-grained timers, such as managing connections in the TIME_WAIT state and connection keep-alive check, we keep a list of `tcb`s sorted by their timeout values. Every second, we check the list and handle any `tcb`s whose timers have expired. Note that keeping the list sorted is trivial since a newly-added entry should have a strictly larger timeout than any of those that are already in the list. For fine-grained retransmission timers, we use the remaining time (in milliseconds) as the hash table index, and process all `tcb`s in the same bucket when a timeout expires for the bucket. Since retransmission timers are used by virtually all `tcb`s whenever a data (or SYN/FIN) packet is sent, keeping a sorted list would consume a significant amount of CPU cycles. Such fine-grained event batch processing with millisecond granularity greatly reduces the overhead.

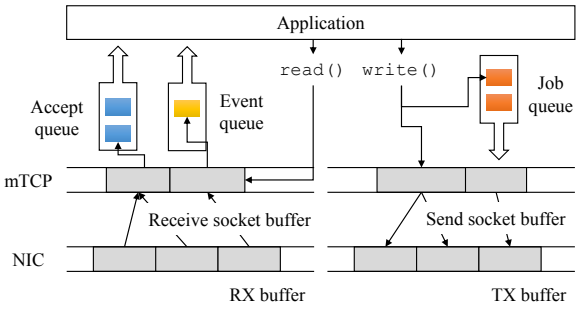


Figure 6: Batch processing of events and jobs.

3.2.3 Batched Event Handling

mTCP transparently enables batch processing of multiple flow events, which effectively amortizes the context switch cost over multiple events. After receiving packets in batch, mTCP processes them to generate a batch of flow-level events. These events are then passed up to the application, as illustrated in Figure 6. The TX direction works similarly, as the mTCP library transparently batches the write events into a write queue. While the idea of amortizing the system call overhead using batches is not new [28, 43], we demonstrate that benefits similar to that of batched syscalls can be effectively achieved in user-level TCP.

In our experiments with 8 RX/TX queues per 10 Gbps port, the average number of events that an mTCP thread generates in a single scheduling period is about 2,170 for both TX and RX directions (see Section 5.1). This ensures that the cost of a context switch is amortized over a large number of events. Note the fact that the use of multiple queues does not decrease the number of the events processed in a batch.

3.2.4 Optimizing for Short-lived Connections

We employ two optimizations for supporting many short-lived concurrent connections.

Priority-based packet queueing: For short TCP connections, the control packets (*e.g.*, SYN and FIN) have a critical impact on the performance. Since the control packets are mostly small-sized, they can often be delayed for a while when they contend for an output port with a large number of data packets. We prioritize control packets by keeping them in a separate list. We maintain three kinds of lists for TX as shown in Figure 5. First, a control list contains the packets that are directly related to the state of a connection such as SYN, SYN/ACK, and ACK, or FIN and FIN/ACK. We then manage ACKs for incoming data packets in an ACK list. Finally, we keep a data list to send data in the socket buffers of TCP flows. When we put actual packets in a TX queue, we first fill the packets from a control list and an ACK list, and later queue the data packets. By doing this, we prioritize important packets

to prevent short connections from being delayed by other long connections. [4]

Lightweight connection setup: In addition, we find that a large portion of connection setup cost is from allocating memory space for TCP control blocks and socket buffers. When many threads concurrently call `malloc()` or `free()`, the memory manager in the kernel can be easily contended. To avoid this problem, we pre-allocate large memory pools and manage them at user level to satisfy memory (de)allocation requests locally in the same thread.

3.3 Application Programming Interface

One of our primary design goals is to minimize the porting effort of existing applications so that they can easily benefit from our user-level TCP stack. Therefore, our programming interface must preserve the most commonly used semantics and application interfaces as much as possible. To this end, mTCP provides a socket API and an event-driven programming interface.

User-level socket API: We provide a BSD-like socket interface; for each BSD socket function, we have a corresponding function call (*e.g.*, `accept()` becomes `mtcp_accept()`). In addition, we provide functionalities that are frequently used with sockets, e.g., fcntl and ioctl, for setting the socket as nonblocking or getting/setting the socket buffer size. To support various applications that require inter-process communication using `pipe()`, we also provide `mtcp_pipe()`.

The socket descriptor space in mTCP (including the fds of `pipe()` and `epoll()`) is local to each mTCP thread; each mTCP socket is associated with a thread context. This allows parallel socket creation from multiple threads by removing lock contention on the socket descriptor space. We also relax the semantics of `socket()` such that it returns any available socket descriptor instead of the minimum available fd. This reduces the overhead of finding the minimum available fd.

User-level event system: We provide an `epoll()`-like event system. While our event system aggregates the events from multiple flows for batching effects, we do not require any modification in the event handling logic. Applications can fetch the events through `mtcp_epoll_wait()` and register events through `mtcp_epoll_ctl()`, which correspond to `epoll_wait()` and `epoll_ctl()` in Linux. Our current `mtcp_epoll()` implementation supports events from mTCP sockets (including listening sockets) and pipes. We plan to integrate other types of events (*e.g.*, timers) in the future.

[4] This optimization can potentially make the system more vulnerable to attacks, such as SYN flooding. However, existing solutions, such as SYN cookies, can be used to mitigate the problem.

Applications: mTCP integrates all techniques known at the time of this writing without requiring substantial kernel modification while preserving the application interface. Thus, it allows applications to easily scale their performance without modifying their logic. We have ported many applications, including `lighttpd`, `ab`, and `SSLShader` to use mTCP. For most applications we ported, the number of lines changed were less than 100 (more details in Section 4). We also demonstrate in Section 5 that a variety of applications can directly enjoy the performance benefit by using mTCP.

However, this comes with a few trade-offs that applications must consider. First, the use of shared memory space offers limited protection between the TCP stack and the application. While the application cannot directly access the shared buffers, bugs in the application can corrupt the TCP stack, which may result in an incorrect behavior. Although this may make debugging more difficult, we believe this form of fate-sharing is acceptable since users face a similar issue in using other shared libraries such as dynamic memory allocation/deallocation. Second, applications that rely on the existing socket fd semantics must change their logic. However, most applications rarely depend on the minimum available fd at socket(), and even if so, porting them will not require significant code change. Third, moving the TCP stack will also bypass all existing kernel services, such as the firewall and packet scheduling. However, these services can also be moved into the user-level and provided as application modules. Finally, our prototype currently only supports a single application due to the limitation of the user-level packet I/O system. We believe, however, that this is not a fundamental limitation of our approach; hardware-based isolation techniques such as VMDq [5] and SR-IOV [13] support multiple virtual guest stacks inside the same host using multiple RX/TX queues and hardware-based packet classification. We believe such techniques can be leveraged to support multiple applications that share a NIC port.

4 Implementation

We implement 11,473 lines of C code (LoC), including packet I/O, TCP flow management, user-level socket API and event system, and 552 lines of code to patch the PSIO library.[5] For threading and thread synchronization, we use `pthread`, the standard POSIX thread library [11].

Our TCP implementation follows RFC793 [17]. It supports basic TCP features such as connection management, reliable data transfer, flow control, and congestion control. For reliable transfer, it implements cumulative acknowledgment, retransmission timeout, and fast retransmission. mTCP also implements popular options such as timestamp, Maximum Segment Size (MSS), and window scaling. For

[5] The number is counted by SLOCCount 2.26.

congestion control, mTCP implements NewReno [10], but it can easily support other mechanisms like TCP CUBIC [26]. For correctness, we have extensively tested our mTCP stack against various versions of Linux TCP stack, and have it pass stress tests, including cases where a large number of packets are lost or reordered.

4.1 mTCP Socket API

Our BSD-like socket API takes on per-thread semantics. Each mTCP socket function is required to have a context, `mctx_t`, which identifies the corresponding mTCP thread. Our event notification function, `mtcp_epoll`, also enables easy migration of existing event-driven applications. Listing 1 shows an example mTCP application.

```
mctx_t mctx = mtcp_create_context();
ep_id = mtcp_epoll_create(mctx, N);
mtcp_listen(mctx, listen_id, 4096);
while (1) {
  n=mtcp_epoll_wait(mctx,ep_id,events,N,-1);
  for (i = 0; i < n; i++) {
    sockid = events[i].data.sockid;
    if (sockid == listen_id) {
      c = mtcp_accept(mctx, listen_id, NULL);
      mtcp_setsock_nonblock(mctx, c);
      ev.events = EPOLLIN | EPOLLOUT;
      ev.data.sockid = c;
      mtcp_epoll_ctl(mctx, ep_id,
          EPOLL_CTL_ADD, c, &ev);
    } else if (events[i].events == EPOLLIN) {
      r = mtcp_read(mctx, sockid, buf, LEN);
      if (r == 0)
        mtcp_close(mctx, sockid);
    } else if (events[i].events == EPOLLOUT){
      mtcp_write(mctx, sockid, buf, len);
    }
  }
}
```

Listing 1: Sample mTCP application.

mTCP supports `mtcp_getsockopt()` and `mtcp_setsockopt()` for socket options, and `mtcp_readv()` and `mtcp_writev()` for scatter-gather I/O as well.

4.2 Porting Existing Applications

We ported four different applications to mTCP.

Web server (lighttpd-1.4.32): Lighttpd is an open-sourced single-threaded web server that uses event-driven I/O for servicing client requests. We enabled multi-threading to support a per-core listen socket and ported it to mTCP. We changed only ∼65 LoC to use mTCP-specific event and socket function calls. For multi-threading, a total of ∼800 lines[6] were modified out of lighttpd's ∼40,000 LoC.

We also ported lighttpd to MegaPipe for comparison. Because its API is based on the I/O completion model, the porting required more effort as it involved revamping lighttpd's event-based `fdevent` backend library; an additional 126 LoC were required to enable MegaPipe I/O from the multi-threaded version.

Apache benchmarking tool (ab-2.3): ab is a performance benchmarking tool that generates HTTP requests. It acts as a client to measure the performance of a Web server. Scaling its performance is important because saturating a 10 Gbps port with small transactions requires multiple machines that run ab. However, with mTCP we can reduce the number of machines by more than a factor of 4 (see Section 5.3).

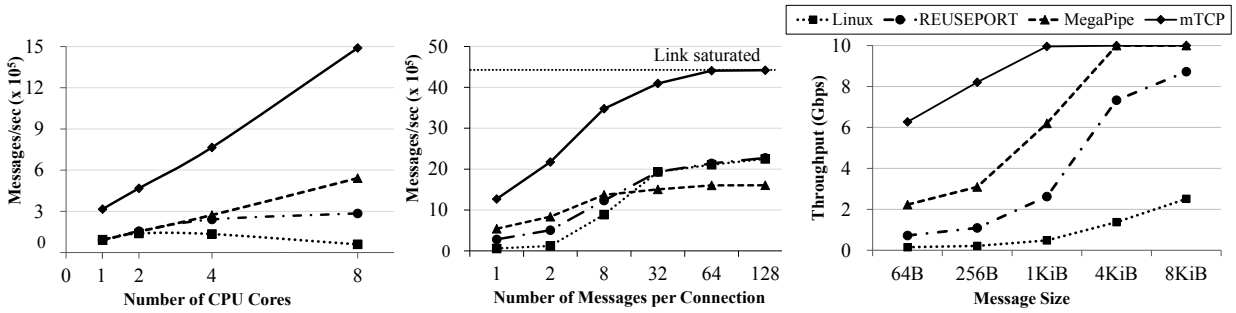
Porting ab was similar to porting lighttpd since ab is also single-threaded. However, ab uses the Apache Portable Runtime (APR) library [16] that encapsulates socket function calls, so we ported the APR library (version 1.4.6) to use mTCP. We modified 29 lines of the APR library (out of 66,493 LoC), and 503 lines out of 2,319 LoC of the ab code for making it multi-threaded.

SSL reverse proxy (SSLShader): SSLShader is a high-performance SSL reverse proxy that offloads crypto operations to GPUs [32]. For small-file workloads, SSLShader reports the performance bottleneck in TCP, spending over 60% CPU cycles in the TCP stack, under-utilizing the GPU. Porting SSLShader to mTCP was straightforward since SSLShader was already multi-threaded and uses `epoll()` for event notification. Besides porting socket function calls, we also replace `pipe()` with `mtcp_pipe()`, which is used to notify the completion of crypto operations by GPU threads. Out of 6,618 lines of C++ code, only 43 lines were modified to use mTCP. It took less than a day to port to mTCP and to finish basic testing and debugging.

Realistic HTTP replay client/server (WebReplay): WebReplay is a pair of client and server programs that reproduces realistic HTTP traffic based on the traffic log collected at a 10 Gbps backhaul link in a large cellular network [45]. Each line in the log has a request URL, a response size, start and end timestamps, and a list of SHA1 hashes of the 4KB content chunks of the original response. The client generates HTTP requests on start timestamps. Using the content hashes, the server dynamically generates a response that preserves the redundancy in the original traffic; the purpose of the system is to reproduce Web traffic with a similar amount of redundancy as the original. Using this, one can test the correctness and performance of network redundancy elimination (NRE) systems that sit between the server and the client. To simulate the traffic at a high speed, however, the WebReplay server must handle 100Ks of concurrent short connections, which requires high TCP performance.

WebReplay is multi-threaded and uses the `libevent` library [6] which in turn calls `epoll()` for event notification. Porting it to mTCP was mostly straightforward in

[6] Some global variables had to be localized to avoid race conditions.



(a) Different number of cores (b) Different number of messages (c) Different sizes of messages

Figure 7: Performance of short TCP connections with 64B messages. (a) and (c) use one message per connection.

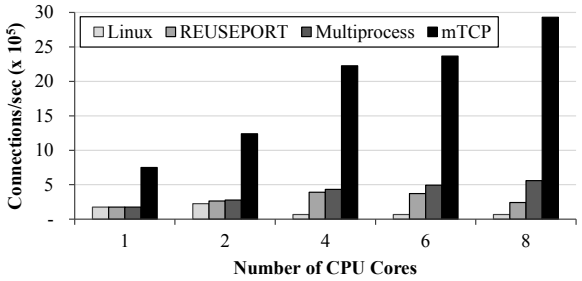


Figure 8: Comparison of connection accept throughputs.

that it only required replacing the socket and libevent calls with the corresponding mTCP API. We modified 44/37 LoC out of 1,703/1,663 lines of server and client code, respectively.

5 Evaluation

We answer three questions in this section:

1. *Handling short TCP transactions*: Does mTCP provide high-performance in handling short transactions? In Section 5.1, we show that mTCP outperforms MegaPipe and Linux (w/o `SO_REUSEPORT`) by 3x and 25x, respectively; mTCP connection establishment alone is 13x and 5x faster than Linux and MegaPipe, respectively.
2. *Correctness:* Does mTCP provide correctness without introducing undesirable side-effects? Section 5.2 shows that mTCP provide fairness and does not introduce long latency.
3. *Application performance*: Does mTCP benefit real applications under realistic workloads? In Section 5.3, we show that mTCP increases the performance of various applications running realistic workload by 33% to 320%.

Experiment Setup: We compare mTCP on Linux 2.6.32 with the TCP stack on the latest Linux kernel (version 3.10.12, with and without `SO_REUSEPORT`) as well as MegaPipe on Linux 3.1.3. We use a machine with one 8-core CPU (Intel Xeon E5-2690 @ 2.90 GHz), 32 GB RAM, and an Intel 10 GbE NIC as a server, and use up to 5 clients of the same type to saturate the server. While mTCP itself does not depend on the kernel version, the underlying PSIO library currently works on Linux 2.6.32. For Linux, we use ixgbe-3.17.3 as the NIC driver.

5.1 Handling Short TCP Transactions

Message benchmark: We first show mTCP's scalability with a benchmark for a server sending a short message as a response. All servers are multi-threaded with a single listening port. Our workload generates a 64 byte message per connection, unless otherwise specified. The performance result is averaged over a one minute period in each experiment. Figure 7 shows the performance as a function of the number of CPU cores, the number of messages per connection (MPC), and message size.

Figure 7(a) shows that mTCP scales almost linearly with the number of CPU cores. Linux without `SO_REUSEPORT` ('Linux') shows poor scaling due to the shared accept queue, and Linux with `SO_REUSEPORT` ('REUSEPORT') scales but not linearly with the number of cores. At 8 cores, mTCP shows 25x, 5x, 3x higher performance over Linux, REUSEPORT, and MegaPipe, respectively.

Figure 7(b) shows that the mTCP's benefit still holds even when persistent connections are used. mTCP scales well as the number of messages per connection (MPC) increases, and it nearly saturates the 10G link from 64 MPC. However, the performance of the other systems almost flattens out well below the link capacity. Even at 32 MPC, mTCP outperforms all others by a significant margin (up to 2.7x), demonstrating mTCP's effectiveness in handling small packets.

Finally, Figure 7(c) shows the throughput by varying the message size. mTCP's performance improvement is more noticeable with small messages, due to its fast processing of small packets. However, both Linux servers fail to saturate the 10 Gbps link for any message size. MegaPipe saturates the link from 4KiB, and mTCP can saturate the link from 1KiB messages.

Connection accept throughput: Figure 8 compares connection throughputs of mTCP and Linux servers. The

		Min	Mean	Max	Stdev
Connect	Linux	0	36	63,164	511.6
	mTCP	0	1	500	1.1
Processing	Linux	0	87	127,323	3,217
	mTCP	1	13	2,323	9.7
Total	Linux	0	124	127,323	3,258
	mTCP	9	14	2,348	9.8

Table 2: Distribution of response times (ms) for 64B HTTP messages for 10 million requests (8K concurrency).

server is in a tight loop that simply accepts and closes new connections. We close the connection by sending a reset (`RST`) to prevent the connection from lingering in the TIME_WAIT state. To remove the bottleneck from the shared fd space, we add 'Multiprocess' which is a multi-process version of the REUSEPORT server. mTCP shows 13x, 7.5x, 5x performance improvement over Linux, REUSEPORT, and Multiprocess, respectively. Among the Linux servers, the multi-process version scales the best while other versions show a sudden performance drop at multiple cores. This is due to the contention on the shared accept queue as well as shared fd space. However, Multiprocess shows limited scaling, due to the lack of batch processing and other inefficiencies in the kernel.

5.2 Fairness and Latency

Fairness: To verify the throughput fairness among mTCP connections, we use `ab` to generate 8K concurrent connections, each downloading a 10 MiB file to saturate a 10 Gbps link. On the server side, we run `lighttpd` with mTCP and Linux TCP. We calculate Jain's Fairness Index with the (average) transfer rate of each connection. As the value gets closer to 1.0, it shows better fairness. We find that Linux and mTCP show 0.973 and 0.999, respectively. mTCP effectively removes the long tail in the response time distribution, whereas Linux often drops `SYN` packets and enters a long timeout.

Latency: Since mTCP relies heavily on batching, one might think it may introduce undesirably long latency. Table 2 shows the latency breakdown when we run ab with 8K concurrent connections against the 64B message server. We generate 10 million requests in total. Linux and mTCP versions respectively achieve 45K and 428K transactions per second on average. As shown in the table, mTCP slightly increases the minimum (9 ms vs. 0 ms) and the median (13 ms vs. 3 ms) response times. However, the mean and maximum response times are 8.8x and 54.2x smaller than those of Linux, while handling 9.5x more transactions/sec. In addition, the standard deviation of the response times in mTCP is much smaller, implying that mTCP produces more predictable response times, which is becoming increasingly important for modern datacenter applications [33].

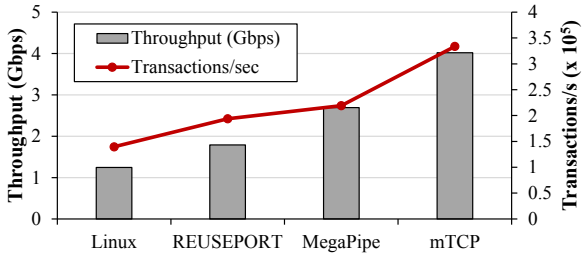


Figure 9: Performance of four versions of `lighttpd` for static file workload from SpecWeb2009.

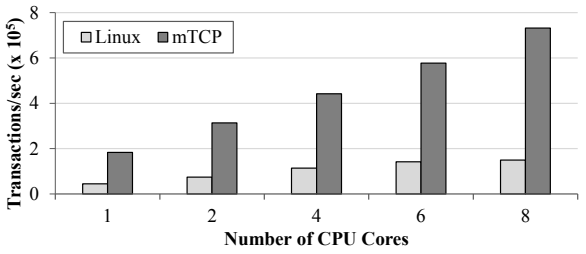


Figure 10: Performance of ab as a function of the number of cores. The file size is 64B and 8K concurrent connections are used.

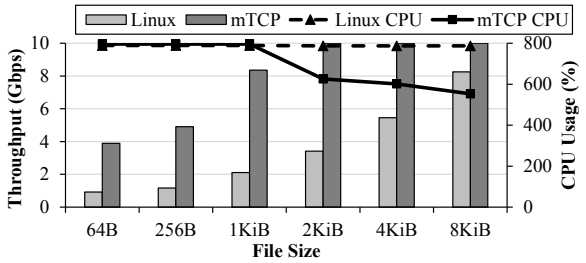


Figure 11: Performance of ab as a function of a file size. The number of cores is set to 8 with 8K concurrent connections.

5.3 Application Performance

We now demonstrate the performance improvement for existing applications under realistic workloads.

lighttpd and ab: To measure the performance of lighttpd in a realistic setting, we use the static file workload extracted from SpecWeb2009 and compare the performance of different lighttpd versions ported to use mTCP, MegaPipe, and Linux with and without `SO_REUSEPORT`. Figure 9 shows that mTCP improves the throughput of lighttpd by 3.2x, 2.2x, 1.5x over Linux, REUSEPORT, and MegaPipe, respectively. Even though the workload fits into the memory, we find that heavy system calls for VFS operations limit the performance.

We now show the performance of ab. Figure 10 shows the performance of Linux-based and mTCP-based ab when varying the number of CPU cores when fetching a 64 byte file over HTTP. The scalability of Linux is limited, since it shares the fd space across multiple threads.

Figure 10 shows the performance of ab and the corresponding CPU utilization when varying the file size from 64 bytes to 8 KiB. From 2 KiB, mTCP saturates the link.

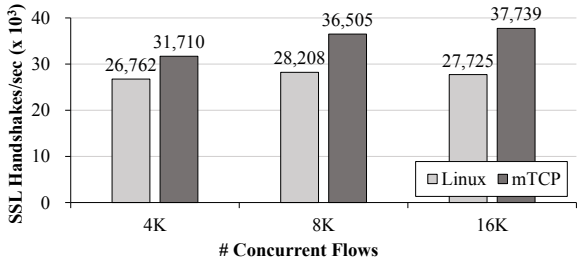


Figure 12: SSL handshake throughputs of SSLShader with a different levels of concurrency.

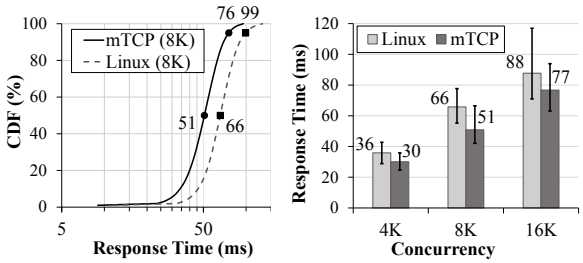


Figure 13: HTTPS response time distributions of SSLShader on Linux and mTCP stacks. We use 8K concurrent connections in the left graph, and mark median and 95th-percentile numbers.

# of copies	1	2	3	4	5	6	7
Linux (ms)	27.8	29.0	45.8	**1175.1**	-	-	-
mTCP (ms)	0.5	0.9	2.6	8.1	17.5	37.1	79.8

Table 3: Averages of extra delays (in ms) from the original response times when replaying *n* copies of the log concurrently.

	# of concurrent connections	# of new connections per second	Bandwidth (Gbps)
Mean	23,869	14,425	2.28
Min	20,608	12,755	1.79
Max	25,734	15,440	3.48

Table 4: Log statistics for WebReplay.

At the same time, mTCP's event-driven system saves CPU cycles.

When testing mTCP with long-lived connections (not shown in the figure), we find that it consumes more CPU cycles than Linux. mTCP shows a CPU utilization of 294% compared to 80% for Linux-3.10.12 when serving 8,000 concurrent connections, each transferring a 100 MiB file. This is because we did not fully utilize modern NIC features, such as TCP checksum offload, large segmentation offload (LSO), and large receive offload (LRO). However, we believe that mTCP can easily incorporate these features in the future.

SSLShader: We benchmark the performance of the SSLShader with one NVIDIA GPU (Geforce GTX 580) on our server. We use mTCP-based lighttpd as a server and ab as a client. On a separate machine, we run SSLShader as a reverse proxy to handle HTTPS transactions. SSLShader receives an HTTPS request from ab and decrypts the request. It then fetches the content from lighttpd in plaintext, encrypts the response using SSL, and sends it back to the client. We use 1024-bit RSA, 128bit-AES, and HMAC-SHA1 as the cipher suite, which is widely used in practice. To measure the performance of SSL handshakes, we have ab to fetch 1-byte objects through SSLShader while varying the number of concurrent connections.

Figure 12 shows that mTCP improves the performance over the Linux version by 18% to 33%. As the concurrency increases, the benefit of mTCP grows, since mTCP scales better with a large number of concurrent connections. Figure 13 indicates that mTCP also reduces the response times compared to the Linux version. Especially, mTCP reduces the tail in the response time distribution over large concurrent connections with a smaller variance, as is also shown in Section 5.2.

WebReplay: We demonstrate that mTCP improves the performance of a real HTTP traffic replayer. We focus on the server's performance improvement because it performs more interesting work than the client. To fully utilize the server, we use four 10 Gbps ports and connect each port to a client. The workload (HTTP requests) generated by the clients is determined by the log captured at a cellular backhaul link [45]. We replay the log for three minutes at a peak time (at 11 pm on July 7, 2012) during the measurement period. The total number of requests within the timeframe is 2.8 million with the median and average content size as 1.7 KB and 40 KB. Table 4 summarizes the workload that we replay. Unfortunately, we note that the trace we replay does not simulate the original traffic perfectly since a longer log is required to effectively simulate idle connections. Actually, the original traffic had as much as 270K concurrent connections with more than 1 million TCP connections created per minute. To simulate such a load, we run multiple copies of the same log concurrently for this experiment.

Table 3 compares the averages of extra delays from the original response times when we replay *n* copies of the log concurrently with Linux and mTCP-based WebReplayer. We find that the Linux server works fine up to the concurrent run of three copies, but the average extra delay goes up beyond 1 second at four copies. In contrast, mTCP server finishes up to seven copies while keeping the average extra delay under 100 ms. The main cause for the delay inflation in the Linux version is the increased number of concurrent TCP transactions, which draws the bottleneck in the TCP stack.

6 Related Work

We briefly discuss previous work related to mTCP.

System call and I/O batching: Frequent system calls are often the performance bottleneck in busy servers.

FlexSC [40] identifies that CPU cache pollution can waste more CPU cycles than the user/kernel mode switch itself. They batch the system calls by having user and kernel space share the syscall pages, which allows significant performance improvement for event-driven servers [41]. MegaPipe employs socket system call batching in a similar way, but it uses a standard system call interface to communicate with the kernel [28].

Batching also has been applied to packet I/O to reduce the per-packet processing overhead. PacketShader I/O engine [27] reads and writes packets in batches and greatly improves the packet I/O performance, especially for small packets. Packet I/O batching reduces the interrupt, DMA, IOMMU lookup, and dynamic memory management overheads. Similar approaches are found in other high-performance packet I/O libraries [4, 7, 39].

In contrast, mTCP eliminates socket system calls by running the TCP stack in the user level. Also, it enforces batching from packet I/O and TCP processing up to user applications. Unlike FlexSC and MegaPipe, batching in mTCP is completely transparent without requiring kernel or user code modification. Moreover, it performs batching in both directions (e.g., packet TX and RX, application to TCP and TCP to application).

Connection locality on multicore systems: TCP performance can be further optimized on multiprocessors by providing connection locality on the CPU cores [37]. By handling all operations of same connection on the same core, it can avoid inter-core contention and unnecessary cache pollution. mTCP adopts the same idea, but applies it to both flow- and packet-level processing.

User-level TCP stacks: There have been several attempts to move the entire networking stack from the kernel to the user level [22, 24, 25, 42]. These are mainly (1) to ease the customizing and debugging of new network protocols or (2) to accelerate the performance of existing protocols by tweaking some internal variables, such as the TCP congestion control parameters. They focus mostly on providing a flexible environment for user-level protocol development or for exposing some in-kernel variables safely to the user level. In contrast, our focus is on building a user-level TCP stack that provides high scalability on multicore systems.

Light-weight networking stacks: Some applications avoid using TCP entirely for performance reasons. High performance key-value systems, such as memcached [9], Pilaf [35], and MICA [34], either use RDMA or UDP-based protocols to avoid the overhead of TCP. However, these solutions typically only apply to applications running inside a datacenter. Most user-facing applications must still rely on TCP.

Multikernel: Many research efforts enhance operating system scalability for multicore systems [19, 20, 44]. Barrelfish [19] and fos [44] separate the kernel resources for each core by building an independent system that manages per-core resources. For efficient inter-core communication, they use asynchronous message passing. Corey [20] attempts to address the resource sharing problem on multicore systems by having the application explicitly declare shared and local resources across multiple cores. It enforces the default policy of having private resources for a specific core to minimize unnecessary contention. mTCP borrows the concept of per-core resource management from Barrelfish, but allows efficient sharing between application and mTCP threads with lock-free data structures.

Microkernels: The microkernel approach bears similarity with mTCP in that the operating system's services run within the user level [23, 30, 38]. Exokernel [23], for example, provides a minimal kernel and low-level interfaces for accessing hardware while providing protection. It exposes low-level hardware access directly to the user level so that applications perform their own optimizations. This is conceptually similar to mTCP's packet I/O library that directly accesses the NIC. mTCP, however, integrates flow-level and packet-level event batch processing to amortize the context switch overhead, which is often a critical bottleneck for microkernels.

7 Conclusion

mTCP is a high-performance user-level TCP stack designed for multicore systems. We find that the Linux kernel still does not efficiently use the CPU cycles in processing small packets despite recent improvements, and this severely limits the scalability of handling short TCP connections. mTCP unleashes the TCP stack from the kernel and directly delivers the benefit of high-performance packet I/O to the transport and application layer. The key enabler is transparent and bi-directional batching of packet- and flow-level events, which amortizes the context switch overhead over a batch of events. In addition, the use of lock-free data structures, cache-aware thread placement, and efficient per-core resource management contributes to mTCP's performance. Finally, our evaluation demonstrates that porting existing applications to mTCP is trivial and mTCP improves the performance of existing applications by up to 320%.

Acknowledgement

We would like to thank our shepherd George Porter and anonymous reviewers from NSDI 2014 for their valuable comments. We also thank Sangjin Han for providing the MegaPipe source code, and Sunil Pedapudi and Jaeheung Surh for proofreading the final version. This research is supported by the National Research Foundation of Korea (NRF) grant #2012R1A1A1015222 and #2013R1A1A1076024.

References

[1] Facebook. https://www.facebook.com/.

[2] Google. https://www.google.com/.

[3] How long does it take to make a context switch? http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

[4] Intel DPDK: Data Plane Development Kit. http://dpdk.org/.

[5] Intel VMDq Technology. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/vmdq-technology-paper.pdf.

[6] Libevent. http://libevent.org/.

[7] Libzero for DNA: Zero-copy flexible packet processing on top of DNA. http://www.ntop.org/products/pf_ring/libzero-for-dna/.

[8] Lighttpd. http://www.lighttpd.net/.

[9] memcached - a distributed memory object caching system. http://memcached.org.

[10] The NewReno modification to TCP's fast recovery algorithm. http://www.ietf.org/rfc/rfc2582.txt.

[11] The open group base specifications issue 6, IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html.

[12] Packet I/O Engine. http://shader.kaist.edu/packetshader/io_engine/.

[13] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf.

[14] The SO_REUSEPORT socket option. https://lwn.net/Articles/542629/.

[15] The Apache HTTP Server Project. http://httpd.apache.org/.

[16] The Apache Portable Runtime Project. http://apr.apache.org/.

[17] Transmission control protocol. http://www.ietf.org/rfc/rfc793.txt.

[18] Twitter. https://twitter.com/.

[19] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2009.

[20] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[22] D. Ely, S. Savage, and D. Wetherall. Alpine: a user-level infrastructure for network protocol development. In *Proceedings of the conference on USENIX Symposium on Internet Technologies and Systems (USIT)*, 2001.

[23] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the ACM symposium on Operating systems principles (SOSP)*, 1995.

[24] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems (TOCS)*, 20(1):49–83, Feb. 2002.

[25] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying safe user-level network services with ictcp. In *Proceedings of the conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004.

[26] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[27] S. Han, K. Jang, K. Park, and S. B. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[28] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the USENIX*

conference on Operating Systems Design and Implementation (OSDI), 2012.

[29] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[30] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. *SIGOPS Oper. Syst. Rev.*, 31(5):66–77, Oct. 1997.

[31] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proceedings of the ACM SIGCOMM conference on Internet measurement conference (IMC)*, 2011.

[32] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*, 2011.

[33] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2012.

[34] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[35] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.

[36] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *Proceedings of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[37] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the ACM european conference on Computer Systems (EuroSys)*, 2012.

[38] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the Computer Society International Conference (COMPCON)*, 1989.

[39] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the USENIX conference on Annual Technical Conference (ATC)*, 2012.

[40] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[41] L. Soares and M. Stumm. Exception-less system calls for event-driven servers. In *Proceedings of the USENIX conference on USENIX annual technical conference (ATC)*, 2011.

[42] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking (TON)*, 1(5):554–565, Oct. 1993.

[43] V. Vasudevan, D. G. Andersen, and M. Kaminsky. The case for VOS: the vector operating system. In *Proceedings of the USENIX conference on Hot topics in operating systems (HotOS)*, 2011.

[44] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, Apr. 2009.

[45] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the annual international conference on Mobile systems, applications, and services (MobiSys)*, 2013.