RESEARCH-ARTICLE

Enhanced control path for repeated TCP connections

JUNHO LEE, Korea University, Seoul, South Korea

GYEONGSIK YANG, Korea University, Seoul, South Korea

ZHIXIONG NIU, Microsoft Research, Redmond, WA, United States

PENG CHENG, Microsoft Research, Redmond, WA, United States

YONGQIANG XIONG, Microsoft Research, Redmond, WA, United States

CHUCK YOO, Korea University, Seoul, South Korea

Open Access Support provided by:

Korea University

Microsoft Research

Published: 24 August 2021

Citation in BibTeX format

APSys '21: 12th ACM SIGOPS Asia-Pacific Workshop on Systems
August 24 - 25, 2021
Hong Kong, China

Conference Sponsors:
SIGOPS

https://doi.org/10.1145/3476886.3477512
ISBN: 9781450386982

Enhanced Control Path for Repeated TCP Connections

Junho Lee
jh.lee@os.korea.ac.kr
Korea University
Seoul, South Korea

Gyeongsik Yang
ksyang@os.korea.ac.kr
Korea University
Seoul, South Korea

Zhixiong Niu
zhixiong.niu@microsoft.com
Microsoft Research
Beijing, China

Peng Cheng
pengc@microsoft.com
Microsoft Research
Beijing, China

Yongqiang Xiong
yongqiang.xiong@microsoft.com
Microsoft Research
Beijing, China

Chuck Yoo
chuckyoo@os.korea.ac.kr
Korea University
Seoul, South Korea

ABSTRACT

This paper presents FALTCON that enhances the control path for repeated TCP connections. First, we measure and find that the control path of TCP stack consumes as many CPU cycles as that of the data path, which brings up the importance of optimizing the control path. Yet, to the best of our knowledge, there has been little research effort on investigating the control path. Also, we observe that a significant portion of TCP traffic (e.g., HTTP) is not only short-lived but also repeated for a server and client pair. We design FALTCON to take advantage of the property of being repeated. Specifically, FALTCON re-designs the control path to remove the duplicate allocation of the structures and redundant operations over them. FALTCON is implemented in Linux 5.1 that has the latest and highly efficient networking stack. Furthermore, we optimize FALTCON to be lockless entirely and to work per-core. The experiment results show that FALTCON achieves a higher number of connections than Linux, up to 19%, and with much less CPU cycles up to 31%.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Communications management**; • **Networks → Transport protocols**.

APSys '21, August 24–25, 2021, Hong Kong, China

ACM ISBN 978-1-4503-8698-2/21/08…$15.00
https://doi.org/10.1145/3476886.3477512

KEYWORDS

Operating System, TCP, Short-lived TCP, Repeated TCP, Kernel TCP Stack

ACM Reference Format:
Junho Lee, Gyeongsik Yang, Zhixiong Niu, Peng Cheng, Yongqiang Xiong, and Chuck Yoo. 2021. Enhanced Control Path for Repeated TCP Connections. In *12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3476886.3477512

1 INTRODUCTION

As the number of mobile and IoT devices and their applications increases rapidly, most TCP network traffic consists of short-lived connections with a small number of packets and small messages [14]. According to the traffic analysis of the campus network [10], more than 75% of the TCP connections have an aggregated packet size of 2KB or less, and 48% consist of one or two data packets.

In this paper, we observe another characteristic of the short-lived TCP connections: a significant portion of connections has the *repeated* property. The representative example of repeated TCP traffic is HTTP, which accounts for up to 78% of cellular network traffic [14]. According to the analysis of HTTP Archive [2] and Google [1], 40% of the HTTP connections consist of single transactions (one message each direction between server and client), and a single web page between a client and server pair generates a median of 14 TCP connections, which indicates that TCP connections used for HTTP tend to be short-lived and also repeated (details in §2.2.1). We refer to these short-lived and repeated TCP connections as "repeated TCP."

We conduct a bottleneck analysis on the repeated TCP (§2.2.2). In the results, we find that connection establishment (control path) takes up 46% of TCP layer in terms of CPU cycles and 40% are used for actual packet transmission (data path). The results indicate that the control path consumes a high portion of CPU cycles as the data path in processing repeated TCP. Because TCP traffic is repeated, the control path

to establish repeated connections become a major bottleneck in the TCP stack. To the best of our knowledge, however, the control path of TCP was not the main scope in previous studies because the previous studies focused on optimizing the data path (§2.1), which necessitates the investigation of the control path to find room for enhancement.

To this end, this paper proposes FALTCON, an enhanced TCP control path for the repeated TCP. There have been two approaches for optimizing TCP: user-level and kernel-based. The user-level approach improved the data path performance of TCP by processing TCP packets and IO events at the user-level to avoid the inefficiencies in the kernel [3, 9]. However, these studies changed the semantics of packet processing and the packet IO APIs (e.g., system calls). Thus, these studies have the problem of backward compatibility on user applications [12], which requires the modification of existing user applications that rely on the kernel APIs. Furthermore, it has been reported that the user-level approach cannot share socket among threads and processes, so that it causes problems in fork and container live migration [11]. For this reason, we take the kernel-based approach to design FALTCON.

FALTCON stands on the following observation: as repeated TCP connections are generated between an identical pair of server and client, the kernel networking stack performs duplicate operations in the connection establishment. A key idea of FALTCON is to re-design the related structures and operations in the connection establishment of the TCP control path. FALTCON keeps the client and server pairs that generate the repeated TCP and maintains the relevant structures and states for connection establishment, which avoids duplicated structure allocations (§3.1, f-struct keeper) and state initializations (§3.2, f-state keeper). Also, we optimize FALTCON structures and operations to work per-core and lockless to reduce further latency or management costs (§3.3, FALTCON archive optimizations). Because FALTCON optimizes the control path, our goal is to design FALTCON that can work orthogonally with the existing studies on data path optimization.

We implement FALTCON within a recent Linux kernel (version 5.1.0)[1] alongside the existing control path so that the TCP stack maintains the backward compatibility. Thus, TCP traffic without short-lived and repeated properties can be processed through the original TCP control path. Also, existing applications run without any modification because FALTCON does not change kernel semantics. In the evaluation, we measure the CPU cycles consumed for structure allocations and state initialization of repeated connections. Also, we experiment with HTTP to show that FALTCON enhances the control path performance up to 19%, compared to the native Linux (§4).

The remainder of this paper is organized as follows. §2 explains the background, motivation, and related work of this study. §3 presents the design of FALTCON. In §4, the performance evaluations on FALTCON are presented. Finally, we conclude this paper in §5.

[1]We implement FALTCON on a recent Linux kernel, as it includes a widely-used open-sourced TCP stack. However, any other TCP stacks that allocate structures and initiate states per each TCP connection (e.g., OpenBSD) could benefit from FALTCON designs.

2 RELATED WORK AND MOTIVATION

2.1 Related Work

Various papers have tried to improve the performance on short-lived TCP connections [7, 9, 12]. We discuss the previous studies by dividing into kernel-based approach and user-level approach, which are summarized in Table 1.

Kernel-based approach. The kernel-based approach includes StackMap [15], MegaPipe [7], FastSocket [12], and Kafe [8]. Two studies, StackMap and MegaPipe optimized the data path by removing data copy between kernel and user-space (StackMap) or by batching syscalls (Megapipe). However, to take advantage of their optimizations, the applications have to be modified. FastSocket partitioned the socket table and eliminated the unnecessary overhead of VFS processing from sockets. Kafe pre-allocated skb structures and data buffers for maintaining packet metadata and payloads, respectively. It optimized packet forwarding operations while maintaining kernel semantics.

User-level approach. As the user-level based approach, DPDK [3], mTCP [9], and SocksDirect [11] are representative studies. DPDK and mTCP achieved high performance by implementing the TCP stack at the user level to avoid inefficient packet processing in the kernel. However, they have the disadvantages of breaking the backward compatibility, which requires the re-implementation of existing applications. SocksDirect [11] leveraged RDMA for high-speed. RDMA [6] is specialized hardware that offloads packet transport processing. Data transfer is (usually) offloaded in NIC without the involvement of host CPUs. SocksDirect achieved up to 20x the performance while maintaining the backward compatibility of the kernel. However, the solution cannot be used in environments where RDMA is not available.

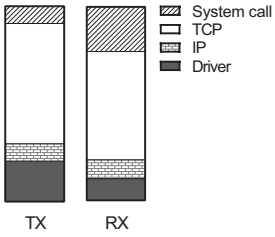
Goal. In this study, we focus on in-kernel optimization of the control path, which is complementary to the data path optimizations. We design FALTCON to provide full transparency to existing applications without changing the kernel semantics.

2.2 Motivation

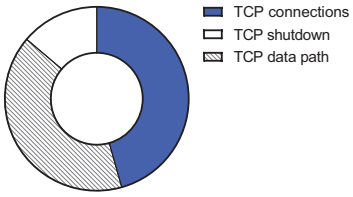
2.2.1 HTTP traffic. The repeated connections have the stateless nature of HTTP. In other words, a web server and client pair creates a connection with states independent of other

Table 1: Comparison of the features of related work and FALTCON.

	Designs					Kernel compatibility	Require specialized HW
	Approach	Hardware offloading	In-kernel optimization	Data path optimization	Control path optimization		
FATLCON	Kernel-based		✓		✓	✓	
StackMap [15]	Kernel-based		✓ (new API)	✓		✗	
MegaPipe [7]			✓ (new API)	✓		✗	
FastSocket [12]			✓ (new API)	✓		✗	
Kafe [8]			✓	✓		✓	
mTCP [9]	User-level			✓		✗	
DPDK [3]				✓		✗	✓
SocksDirect [11]		✓ (RDMA device)		✓		✓	✓

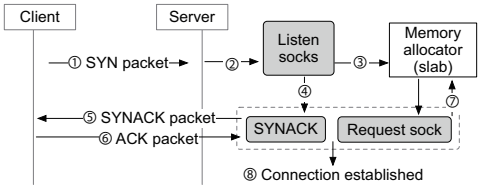


(a) CPU cycle for kernel networking.

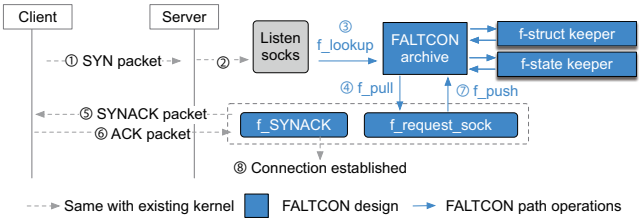


(b) TCP RX cycle in detail.

Figure 1: Bottleneck analysis in repeated TCP.



(a) Linux.



(b) FALTCON.

Figure 2: Linux and FALTCON control path.

connections, and the connection is closed when the communication has concluded. According to [1, 2], 74% of HTTP/1.1 traffic is single-transaction. HTTP/1.1 webpages generate HTTP requests a median of 17 times. Even though HTTP/1.1 [5] introduced the "keep-alive" option, it can cause performance problems in heavily loaded servers because it does not remove connection resources until the timeout. HTTP/2 further proposed multiple parallel streams that enable data transfer parallelization within a single connection. Despite such efforts, 25% of HTTP/2 connections still consist of single-transaction. HTTP/2 webpages generate HTTP requests a median of 13 times. In total HTTP traffic, the connections that carry a single transaction account for 40% of the total, with a median of 14 repeated requests made from HTTP web pages.

2.2.2 Bottleneck analysis. We analyze the bottleneck in repeated TCP. For analysis, we generate TCP connections through *lighttpd* as server and *ab* as client. The webserver runs on an Intel Xeon CPU (E5-2695 v4 @ 2.10GHz, 18 cores) with 100GB of memory and a 10 Gbps NIC (Intel 82599 chipsets). The client-side does not have any bottleneck in this experiment. Figure 1a shows the analysis results of the server's CPU cycles using perf for the entire network stack in Linux. According to Figure 1a, TCP consumes 56% and 62% of CPU cycles in TX and RX, respectively. Further, we analyze the operations of RX in detail to measure the processing cycles on the control path (Figure 1b)[2].

Figure 1b shows that TCP control path occupies 60% out of the RX cycles. TCP connections are established through the three-way handshaking of TCP connections, which consumes 46% of the TCP RX cycle. The remaining part of the control path, TCP shutdown, consumes 14%. Considering that TCP data path takes 40%, the control path is worthy of investigation, but there have been no studies yet. Therefore, this paper focuses on improving the overhead of connection establishment of repeated TCP connections.

We design FALTCON to work inside kernel with the backward compatibility. FALTCON is designed not to be a competitor to the existing studies but rather an orthogonal study that can work together.

[2]Major control path operations, such as connection establishment and shutdown, are processed in the RX path (from the server-side), so we analyze in detail of the RX path of the server-side.

3 DESIGN

We first review the control path of establishing connections. Figure 2a illustrates the TCP connection establishment. In Linux kernel, the client first sends an SYN packet to request a connection (① in Figure 2a), and the server determines the corresponding listen socket (②). Then, to handle the connection, the kernel allocates two structures (`request_sock` and `SYNACK skb`) using memory allocators like SLAB (③). Also, based on the information contained in the listen socket and the SYN packet, the states (e.g., protocol, network family, address, and routing information) are initialized (④). The server and client exchange SYNACK and ACK packets (⑤ and ⑥). At this point, the connection is established. Also, the two structures and initialized states are freed and returned to memory (⑦).

In case of repeated TCP, we observe that the structures and states are identical among repeated connections. FALTCON exploits this observation and avoids the allocation and deallocation in the control path as follows. First, we devise the FALTCON archive that consists of objects, and objects contain structures and states for repeated TCP connections. Then, we define the "keep" operation that is not to free the allocated structures and initialized states but to keep them in memory. Also, "keep" operation is called by "keeper" that maintains and manages the objects of structures and states. There are two keepers: f-struct keeper (§3.1) and f-state keeper (§3.2) to be explained later. This keep operation may look similar to caching operations, but it differs in that it uses a particular policy that maintains objects related to repeated TCP. Lastly, we define three operations for FALTCON to access objects in the FALTCON archive: `f_lookup`, `f_push`, and `f_pull` (§3.3).

Figure 2b shows the control path with FALTCON that works as follows. Servers and clients follow the standard TCP three-way handshaking procedure. After an SYN packet arrives at the server (① in Figure 2b) and is handled by the corresponding listen socket (②), FALTCON looks up the archive by `f_lookup` operation (③) to see if there is a match for the connection. If it is, FALTCON calls `f_pull` to reuse the objects of the connection in the archive (④). Otherwise, the f-struct keeper allocates a new structure, and the f-state keeper initializes the states (§3.1 and §3.2, respectively). After that, the sequence from ⑤ to ⑥ is identical to that of Linux, but instead of releasing and freeing the structures and states, FALTCON calls `f_push` to push the structure and states to the FALTCON archive (⑦). When they are no longer repeated connections, the corresponding objects are removed from the archive. Note that FALTCON applies only to repeated TCP, so non-repeated TCP traffic follows the control path of Linux. In addition, FALTCON is further optimized as lockless and per-core design (§3.3).

3.1 f-struct keeper

The f-struct keeper allocates objects for the `request_sock` and `SYNACK skb` structures as follows. For clarification, we call the TCP server and client pair identified as repeated connections as "FALTCON pair" (f-pair). Each f-pair is distinguished by an "f-key," a tuple (protocol, client IP, server IP, server port number)[3]. For each f-pair, objects kept in the FALTCON archive are `f_request_sock` and `f_SYNACK` that correspond to `request_sock` and `SYNACK skb`. The f-struct keeper also manages and maintains the objects. Speicifically, the f-struct keeper has two policies: (1) keeping policy and (2) elimination policy, which will be explained in order.

3.1.1 Keeping policy. Keeping policy determines which server and client pair is eligible to be an f-pair and how many objects are desirable to be kept in the FALTCON archive. In FALTCON, we define a threshold (k_t) to determine whether a connection is repeated. The f-struct keeper monitors incoming TCP connections. Then, if the connection between a pair of TCP server and client is established more than k_t times for a fixed length of a time (window), e.g., 5 s, the pair becomes an f-pair. Also, concurrent TCP connections frequently exists for an f-pair (§2.2.1), so FALTCON maintains multiple objects. The reason for maintaining multiple objects is that between the concurrent TCP connections, several states in `f_request_sock` and `f_SYNACK` are non-reusable (Table 2). Here, reusable state means that the values of the fields in `f_request_sock` and `f_SYNACK` do not change between concurrent connections of an f-pair. However, other fields (e.g., sequence number) get a new value for a connection even though the connection is repeated. Thus, non-reusable states indicate that each concurrent connection requires its own value, which results in multiple objects. The number of objects for each f-pair is maintained by the maximum number of concurrent connection requests within each window.

3.1.2 Elimination policy. Elimination policy removes objects in the archive that are no longer used. For elimination, we use a timer that fires periodically. When the timer fires, the f-struct keeper traverses the f-pairs in the FALTCON archive and checks whether more than one connection has been established for each f-pair. If an f-pair has had no connection within the timer interval, the objects for the f-pair are removed. FALTCON implementation configures one global timer to reduce the overhead of the elimination policy.

3.2 f-state keeper

For an f-pair, the reusable state has the same values. So, we introduce the f-state keeper that does not initialize the

[3]Note that between the same client and server, repeated connections are created from different client port numbers to known server port numbers.

Table 2: Reusable and not-reusable states in structures.

Structure	States		Share
`f_request_sock`	Reusable	Protocol family, several addresses (e.g., `sk_prot`, `ireq_family`, `sk_rcv_saddr`, `sk_daddr`, `ir_num`)	75%
	Not reusable	Packet sequence number, timestamp (e.g., `rcv_isn`, `rcv_nxt`, `ts_recent`)	25%
`f_SYNACK`	Reusable	Several packet data, destination routing (e.g., `header`, `source`, `_skb_refdst`)	50%
	Not reusable	Sequence number, timestamp (e.g., `ack_seq`, `skb_mstamp_ns`)	50%

reusable states so that FALTCON avoids repetitive initializations. For SYNACK, once the destination IP address of a client is known, the f_state keeper keeps the value for the repeated connections. Thus, the impact of f_state keeper removes redundant operations over the reusable states. Table 2 shows the states that are reusable or not reusable (change per each connection). The f-state keeper reuses states such as several addresses (e.g., IP and source port), protocol, and routing states. In contrast, the non-reusable states (e.g., timestamp) need to be updated, so they must be initialized for each connection. We instrument the states initialized by the TCP control path and get the ratio between reusable and non-reusable states. Table 2 shows that the reusable states of `f_request_sock` is 75% and `f_SYNACK` 50%.

3.3 FALTCON Archive Optimizations

To reduce the lookup speed, the FALTCON archive is designed as a hash table whose key is the hashed value of the f-key and its corresponding value is a linked list. The node of each linked list is an object for an f-pair.

The FALTCON archive is further optimized for multicore scalability and lockless designs. If this structure is shared between multicores, lock contentions to access the archive (e.g., spinlocks) delay the packet processing [13]. Also, an update of the FALTCON archive causes cache invalidation that diminishes locality [4]. Thus, we optimize FALTCON archive and related FALTCON operations as follows.

First, we devise a way to pin the FALTCON archive to per-core. If SYN and ACK packets corresponding to the same connection go to different cores, `f_pull` and `f_push` are called on different cores, which reduces the cache coherence and causes additional locks to update states of FALTCON archives. To avoid this, FALTCON distributes the packets of the same connection on the same core using RSS (receive-side scaling) of NIC.

Second, we note that in processing of SoftIRQ, a lock is required even on the per-core FALTCON archives. FATLCON runs in SoftIRQ along with the TCP stack. SoftIRQ processing is divided into the top half (kernel context) and the bottom half (user context), and the bottom half is scheduled as *ksoftirqd*. In *ksoftirqd*, when new objects are allocated in the archive, locks are required. To make our design entirely lockless, FALTCON disables local IRQ owned by the core that handles it when new objects are allocated. Note that other parts of the networking stack often disable IRQ for the same reasons (e.g., `netdev_alloc_skb`), and the time during which local IRQ is disabled is very short to the extent that it hardly affects other kernel tasks.

Lastly, we consider the operations of FALTCON, such as `f_lookup` (① in Figure 2b), `f_pull` (④), and `f_push` (⑦). These operations inevitably require synchronization toward `f_request_sock` and `f_SYNACK` structures. We utilize the "hold/release" similar to general atomic operations without busy-waiting to reduce the cost of synchronization.

4 EVALUATION

4.1 Settings

Equipment. We run a webserver on a single machine and clients on four machines. The machine used for the server is of an Intel Xeon CPU (E5-2695 v4 @ 2.10GHz, 18 cores) with 100GB memory. The four client machines are equipped with Intel Xeon CPUs—E5-2650 v3 (20 cores), E5-2650 v3 (20 cores), E5-2695 v4 (18 cores), and E5-2650 v4 (12 cores). Each machine has 62GB of memory. All machines are connected through 10 Gbps Ethernet with RSS-enabled NIC cards (Intel 82500 chipsets).

Software. To evaluate repeated TCP connections with FALTCON, a webserver (*Nginx* v 1.18.0) is set without HTTP keep-alive options. The server machine runs one *Nginx* thread per core and with the multi-accept option enabled, so the HTTP processing is fully parallelized. Each IRQ is pinned to each core, and irqbalance is disabled for per-core optimizations of FALTCON. The client-side runs *wrk*, which generates HTTP requests according to each machine's number of CPU cores. The clients create enough HTTP requests to saturate the CPU utilization of the server. Each client continuously creates repeated TCP connections, and 100 connections per core of the server are maintained throughout the experiment.

Metric. As a microbenchmark, we first measure the CPU consumption (in cycles) for connection establishment. Also, we measure the webserver's performance—in the number of connections per second (CPS) that the webserver processes.

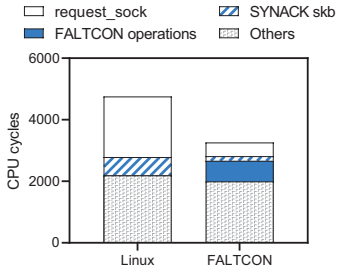
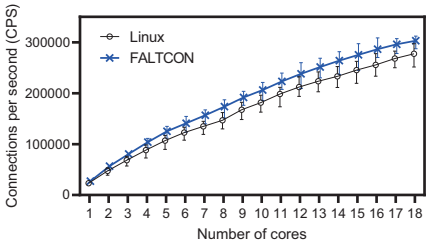
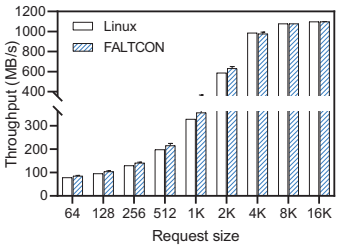


Figure 3: CPU consumption improvement.



(a) Varying number of cores.



(b) Varying sizes of requests.

Figure 4: FALTCON scalability.

We vary the number of cores and the size of requests (messages) to see FALTCON's scalability. All results are compared with Linux kernel version 5.1.

4.2 CPU Consumption Improvement

Figure 3 shows how many CPU cycles are spent in the TCP layer while the server receives SYN and sends SYNACK. The result is measured by get_cycles() in-kernel that logs the current number of CPU cycles. The legends "`request_sock`" and "`SYNACK skb`" refer to the CPU cycle consumptions of the two structures, respectively, for reserving memory space and fill states. The legend "FALTCON operations" refers to CPU cycles consumed for FALCON Archive, such as `f_lookup`, `f_push`, `f_pull`, and keep. Also, "others" includes CPU cycles for TCP control path processing, such as checksum verification, socket lookup from the listen socket table, and TCP header decoding. Compared to Linux, the total CPU consumption of FALTCON is reduced by up to 31%. Specifically, FALTCON reduces the CPU consumptions for `request_sock`, `SYNACK skb`, and others by up to 77%, 75%, and 9%, respectively. The improvement in CPU cycles, especially for the `request_sock` and `SYNACK skb`, comes from eliminating duplicate operations in the control path of repeated TCP connections.

4.3 Scalability on CPU Cores and Request Sizes

Figure 4 illustrates the scalability of FALTCON with the number of CPU cores and request sizes. Each experiment is conducted 30 times, and the average values are shown.

CPU cores. Figure 4a shows scalability related to the number of CPU cores. In this experiment, *wrk* requests 64 B messages to *Nginx* per connection and measures CPS that indicates HTTP/1.1 connections it completes in a second. As a result, the CPS of FALTCON is higher than Linux by 14%, ranging from 9% (18 cores) to 19% (2 cores). Also, FALTCON outperforms Linux over all the number of cores consistently.

Request sizes. Figure 4b shows the throughput according to request size. The server machine utilizes all 18 cores. FALTCON's throughput improves by 8% for request sizes from 64B to 2KB. FALTCON has higher throughput than Linux because it saves CPU cycles in the control path, and the saved CPU cycles are used to process more packets. Due to the space limit, we omit the detailed data of CPU cycles. For request sizes from 4KB, the throughput of FALTCON and Linux are similar because the network link is saturated.

5 CONCLUSION

This paper investigates the TCP control path that consumes as many CPU cycles as the data path. We present FALTCON that optimizes the three-way handshaking in the control path for the repeated connections. Our goal is to design FALTCON without breaking the backward compatibility, and so FALTCON is implemented in Linux 5.1. Through the HTTP experiments, we show that FALTCON achieves a higher number of connections per second than Linux, by up to 19%. Also, FALTCON reduces CPU cycles by up to 31%, indicating that the repeated characteristics of TCP traffic can benefit from FALTCON.

ACKNOWLEDGMENTS

We thank Suwan Kim and Bong-yeol Yu for their discussions to improve the quality of the paper. This research was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF-2019H1D8A2105513) and also by Basic Science Research Program through the NRF funded by the Ministry of Education (NRF-2021R1A6A1A13044830). The corresponding author of this paper is Chuck Yoo.

REFERENCES

[1] 2019. Introduction to HTTP/2. https://developers.google.com/web/fundamentals/performance/http2/
[2] 2020. The 2020 Web almanac by HTTP archive. https://almanac.httparchive.org/en/2020/http2/.
[3] 2020. Intel DPDK: Data Plane Development Kit. http://dpdk.org/.
[4] Thomas E Anderson. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (1990), 6–16.

[5] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1999. RFC2616: Hypertext Transfer Protocol–HTTP/1.1.

[6] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.

[7] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: a new programming interface for scalable network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 135–148.

[8] Cheol-Ho Hong, Kyungwoon Lee, Jaehyun Hwang, Hyunchan Park, and Chuck Yoo. 2018. Kafe: Can OS kernels forward packets fast enough for software routers? *IEEE/ACM Transactions on Networking* 26, 6 (2018), 2734–2747.

[9] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 489–502.

[10] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. 2021. Flow length and size distributions in campus Internet traffic. *Computer Communications* 167 (2021), 15–30.

[11] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. SocksDirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*. 90–103.

[12] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable kernel TCP design and implementation for short-lived connections. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 339–352.

[13] Mircea Negrean, Simon Schliecker, and Rolf Ernst. 2009. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 524–529.

[14] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. 2013. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. 319–332.

[15] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 43–56.