M.S. THESIS

# Optimized Memcached on User-Space TCP Stack

# User-Space TCP Stack을 이용한 Memcached의 최적화

BY

PAN XIAOWAN

JANUARY 2018

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

# Abstract

## Optimized Memcached on User-Space TCP Stack

PAN XIAOWAN
Department of Computer Science and Engineering
College of Engineering
Seoul National University

Memcached is a wildly used multi-thread, distributed Key-Value caching solution in today's many web applications and services to provide high query performance and reduce tail latency. Optimizing memcached has always been a hot topic in both academia and industry from various perspective. In this paper, we describe a solution for memcached that utilizes user space TCP stack on top of intel DPDK to improve its performance on multi-core system. This solution enables the system to process key-value queries bypassing the linux kernel to greatly reduce network overhead especially in situation where the system needs to exchange a huge amount of small messages with clients. We implemented the system based on the best optimized version of memcached, MemC3, and used mTCP with Intel DPDK as the user-space TCP stack. Our experimental results

showed that MemC3 and DPDK can be greatly combined, and the throughput of the system can be increased by 50% even when it scales to multiple cores. Apart from this, we can also see the tail latency can be significantly reduced by our solution.

**Keywords:** Key-Value Store, DPDK, User-space stack, Multi-core

**Student Number:** 2015-23304

# Contents

# List of Figures

# Chapter 1

# Introduction

Memcached[1] is an open source key-value cache wildly used in web applications and services by many companies, such as Facebook, Twitter, Youtube, etc. It is usually used in front of the database to accelerate the query response to client, as storing items in main memory can eliminate expensive disk accesses. In key-value systems, each item is stored as a pair of key and value in main memory. An item data is retrieved by given a key, calculating the item's location by using hash functions, and then returning the value to clients. Memcached item mainly has two features: first, it usually has a size less than hundreds of bytes; second, most of the queries tend to be get/read queries according to statistics[2]. However, current OS kernel cannot process that huge amount of connections when a lot of queries swarm into the memcached server. The in-kernel TCP/IP stack cannot handle with this situation or even utilize multi-core system facility, thus has become the bottleneck of the network.

To overcome this problem, Intel DPDK provides a solution that processing network packets directly in user-space rather than doing it through the kernel, which greatly reduces the network processing overhead. The problem is

that Intel DPDK doesn't provide any detailed function or solution in network protocol, so it cannot be directly port to existed applications like memcached which needs TCP/IP stack to transfer message between application identifiable format and NIC identifiable format. There are some key-value storages that have already applied UDP on DPDK solution to build their own

systems, such as MICA[3], Mega-KV[4]. By porting existed DPDK library functions , we can directly get the packet payload according to UDP packet format without doing much modifications in programs. Despite of the fact that we can use UDP protocol with DPDK to simplify the TCP/IP stack part in applications, UDP protocol has its own limitations than TCP[5] protocol does that it can not maintain connection state or provide reliable transport service. Aside from that, there some implementations of user-space TCP/IP stacks based on DPDK. Among those works, mTCP is an open-source, high-performance implementation especially working for multicore systems.

This paper presents a design and implementation of memcached based on user-space TCP/IP stack. Although previous work has proposed a similar idea, it was not able to build on multi-core system, and the evaluation result showed limited improvement[6]. Based on this, our implementation did some adjustment. We shift our implementation to MemC3[7], an evolved version of memcached, also known as the most well-optimized memcached. MemC3 improved memcached in both memory efficiency and throughput by utilizing two techniques ,

optimistic cuckoo hashing, and optimistic locking, which can especially enhance the performance in read-intensive, highly concurrent workload. Also we found that mTCP could better fit in MemC3 version, with less modification on the data structure and system architecture, but more performance improved compared with the stock memcached.

The rest of the paper is organized into the following sections: Chapter 2 presents more detailed information about the background and motivations, including system architecture of stock memcached and MemC3, as well as Intel DPDK technique and its wide application in key-value systems. Chapter 3 shows our design and implementation of optimized memcached on user-space TCP/IP stack. Chapter 4 shows the evaluation result of our implementation, verifying that our proposed approach works effectively in increasing system throughput and reducing response latency on multicore system. Chapter 5 finally concludes our work in this paper, and also indicates some limitations in our system and further research to do in the future.

# Chapter 2

# Motivations and Background

In this chapter, we focus on the motivation and background of our work from two perspectives. We will first introduce memcached's system architecture, and existed optimization work on it. Then we will introduce the detail contribution of Intel DPDK as it is widely applied in key-value system in term of accelerating packet processing.

## 2.1 Optimization on memcached

When memcached was introduced in the first place, it was designed for single-core system to serve as a key-value cache layer for web service. Then with the wide prevalence of multi-core processor, memcached also keeps evolved to fit the system architecture to utilize multiple threads or processors. While memcached is keeping upgrading, it doesn't change the core of design: (1) using hash function to locate each item, and items with same key are organized in a linked-list,(2) using slab allocator for memory management and preallocateing memory in various-sized chunk, (3) using Least Recently Used (LRU) list to determine which item to evict when the cache is full, by

organizing items of same slab into a single linked-list, and doing item eviction while traversing the linked-list according to each item's reference count value. In order to enable memcached to work on multicore systems, the first evolved version of memcached (1.6) used a single global cache lock on the entire hash table and LRU to realize concurrent data accessing. But obviously it failed to achieve an ideally, linearly increased throughput when the system worker thread scales. Because all the worker threads share the same lock, it will cause contention when multiple threads try to concurrently access the same data structure. Later, some optimization work was done to eliminate this global cache lock bottleneck[8]. The contributions of this paper mainly lie in two aspects: (1) changing the original hash table locking mechanism to allow for parallel access. More specifically, a fine-grained set of locks (versus the global hash table lock) is used to ensure more concurrent operations on the hash table. (2) replacing LRU with Bag LRU — a parallel data structure that can take advantage of the modified hash table. As a result, this modified memcached exhibits linear scalability, and overcomes the limitations of the previous global lock version.

Since memcached is widely used as key-value cache layer for many web service such as Facebook, Twitter, Youtube, etc, the real-world workload can be concluded according to statistics gathered by these companies. According to the statistics, we can conclude the real-world workload features, thus giving us

implies and directions on the future optimization research. For instance, according to research result published by Facebook, workloads in real-world has the following characteristics: (1) queries are small objects dominated. Most keys are smaller than 32 bytes and most values are smaller than a few hundred bytes; (2) queries are read intensive. Workload in Facebook case has a GET/ SET ratio of 30: 1 and some important applications even have a much higher GET/SET ratio up to 500: 1. These features can bring some limitations or bottleneck to memcached system performance. Since additional data structure of each item tends to be bigger than key and value themselves, it will cost more memory space. Especially when a huge amount of items are stored in memory, it is not efficient for memory utilization because additional information takes up more memory space. The second feature implies that even most queries are GETs, the system is still not optimized and lock overhead still significantly limits the throughput scalability. Based on these characteristics, MemC3 dedicates to build a new memcached that can boost the performance under this workload[7]. MemC3's approach includes several contributions:

(1) Replacing original hash table with a novel hashing scheme called optimistic cuckoo hashing.

(2) Replacing LRU eviction policy with a compact CLOCK-based eviction algorithm that requires only 1 bit extra space per cache entry while supporting concurrent cache operations.

(3) Using optimistic locking scheme instead of original mutex lock that eliminates lock overhead and synchronization problems on GET path while ensuring consistency.

By doing evaluation experiment on both MemC3 and stock memcached, the former one was verified to provide higher throughput but requires significantly less memory and computation. Therefore, MemC3 has been generally considered the most well-performed memcached version so far.

In more detail, Figure 1 illustrates the system architecture of MemC3, and documents the process flow when a query sent by a client arrives at the MemC3 server side. Usually there are several memcached servers in the cluster. Each memcached server is responsible for its own part to caching data, and clients usually know how the data is distributed on each memcached server. Client generates queries (SET/GET/DELETE) through UDP or TCP protocol, and sends the request to a certain server. As we can see in the figure from top to the bottom. When the memcached server initializes, it uses a main thread as listener thread to listen to all the requests from the clients, and dispatches several worker threads to serve each connection to do the exact set/get/delete operation on behalf of the client. The incoming events are all notified to memcached threads through Libevent[9] backend, by registering the event type and event handler at the event base. Note that the listener thread communicates with worker threads through pipes. Once a connection request arrives, the listener

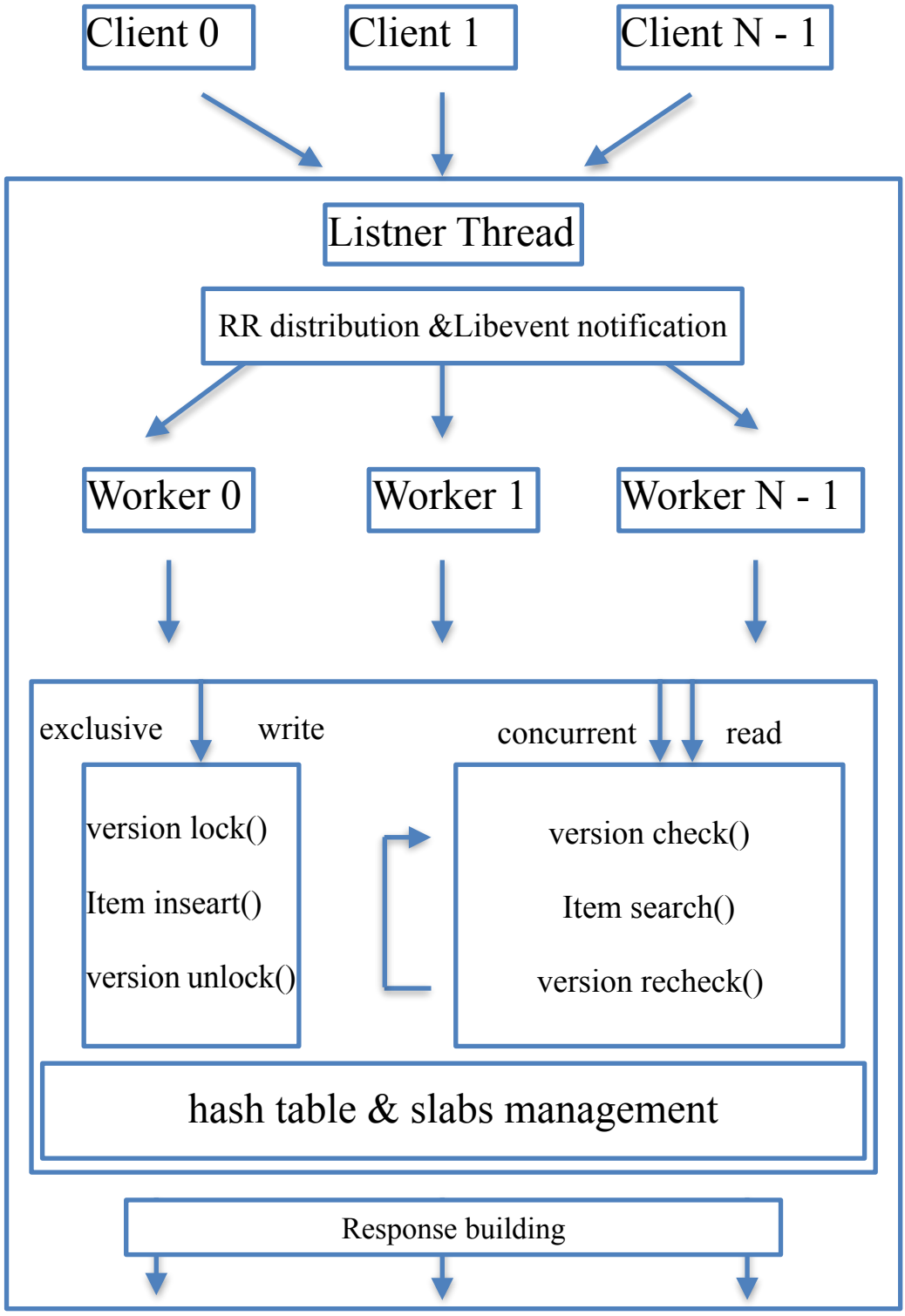


Figure 1 — MemC3 system architecture

thread will create a connection item, and push a specific character to one worker thread. The chosen worker thread will immediately check the character's content and then know it should take over that connection item to handle the request. All the requests are processed in drive_machine function.

The basic process flow in MemC3 keeps constant with original memcached. Main difference as follows:

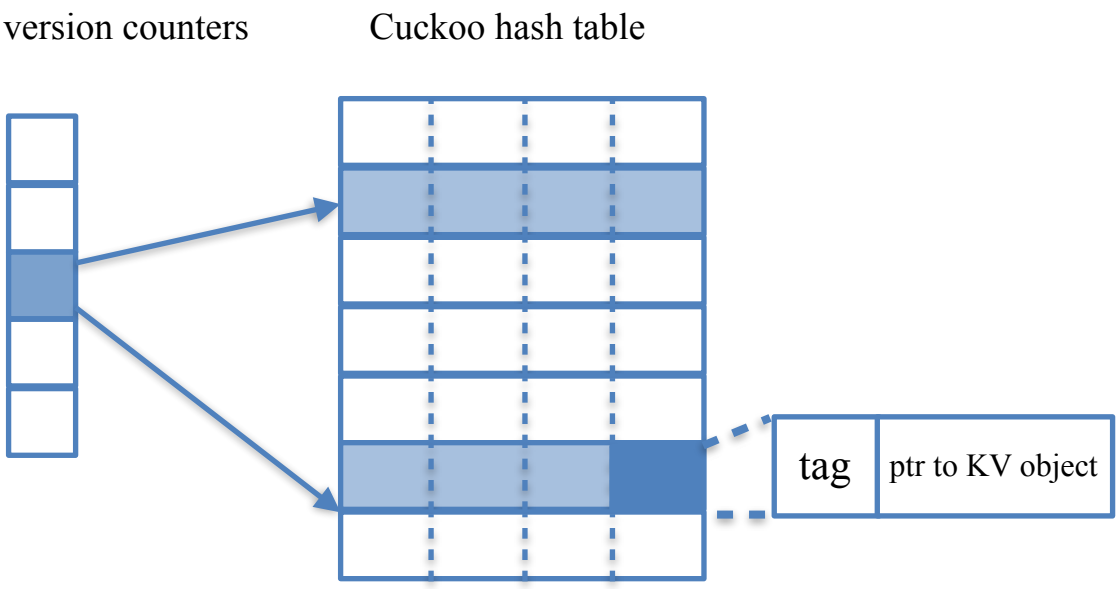


Figure 2 — MemC3 cuckoo hash table

For hash function, as shown in Figure 2, the hash table consists of an array of buckets, each having 4 slots. Each slot contains a *pointer* to the key-value object and a short summary of the key called a tag. So every time when we search an

item, we only need to compare the tag without referring the KV object memory. Each key is mapped to two random buckets, and we should find one bucket to store the item. More detail is described in the original paper.

```
SET(key, value)
begin
    lock();
    ptr = Alloc();
    If ptr == NULL then
        ptr = Evict();
    memcpy key,value to ptr;
    Insert (key, ptr) to hashtable;
    unlock();
```

```
GET(key)
begin
    while true do
        keyversion1 = ReadCounter(key);
        ptr = Lookup(key);
        If ptr == NULL then return NULL;
        fetch data in ptr;
        keyversion2 = ReadCounter(key);
        if keyversion1 & 1 or
            keyversion1 != keyversion2 then
        continue;
        updateCLOCK(key);
        return response;
```

Figure 3 — algorithm of optimistic lock

Locking primitive is illustrated in Figure 3. Generally, MemC3 eliminates locks for read operation by replacing the lock with counter version to compare it before and after getting the item value. However, write lock is still inevitable as it is necessary to ensure data consistency.

## 2.2 DPDK in Fast Packet Processing

Intel Data Plane Development Kit (DPDK) is a set of libraries and drivers for fast packet processing[10]. It is designed to run mostly in Linux user space to reduce the heavy overhead in kernel space. The main libraries include: multicore framework, huge page memory, lock-free ring buffers, poll-mode drivers for networking, crypto and eventdev. Porting DPDK libraries in the project can dramatically accelerate receiving and sending packets with minimum number of CPU cycles but extremely high QPS. Note that DPDK is not a networking stack and does not provide L3 functions, so we cannot directly use the libraries in the project to replace the existed socket functions. Generally, DPDK is used in L2/L3 forwarding rather than directly being applied to user space applications.

Although DPDK does not provide TCP/IP stack, we can still use it and it has already been utilized in some key-value system. CPHash[11], MICA[3], Mega-KV[4] both use DPDK to direct packets to individual cores and avoid network stack overhead.

Network I/O is considered one of the most expensive processing steps for in-memory key-value storages. TCP processing alone may consume 70% of CPU time on a many-core optimized key-value store[11]. Especially in some case where millions of short TCP flood happens, the linux kernel stack cannot even handle with this situation thus lead to dramatic performance decreasing .

To overcome this overhead, CPHash, MICA, and Mega-KV use DPDK direct NIC access to transfer packet data between NICs and the server software, and use UDP for transmission. On the one hand, UDP protocol has been widely used for read requests case in key-value storage for low latency and low overhead. On the other hand, since UDP packet structure is simple and does not require complicated loss recovery and flow/congestion control that TCP provides.

With the tools provided by DPDK, we can use NIC multi-queue support to allocate a dedicated RX and TX queue to each core. Cores do not have synchronization or contention issue with each other. Flow-level core affinity can be achieved by using Receive-Side Scaling(RSS) that sends packets to individual cores by hashing the packet header 5-tuple to identify which RX queue to target, and Flow Directory(FDir) that can use different parts of the packet header plus a user-supplied table to map header values to RX queues. DPDK also provides burst packet I/O to transfer multiple packets each time when it fetches packets from the RX queue. Burst packet I/O reduces per-packet cost of accessing the processing the packets, so the average latency can be much reduced in this approach.

Figure 4 illustrates an UDP packet structure. After fetching an UDP packets from RX queue and using DPDK library function rte_pktmbuf_mtod to get the start address of the payload, we calculate the key-value data's offset so that we

can get the SET/GET request message and process for the next step. One of the benefits of this approach is we can reuse the packet buffer latter when we construct the response packet later, only by flipping the source and destination IP addresses and ports in the header of the packet, and filling the payload part with return information to the clients.

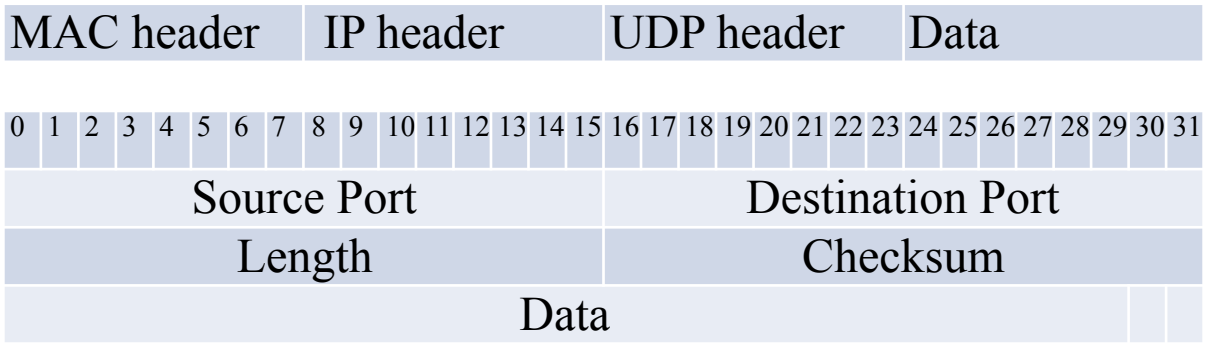


Figure 4 — UDP packet structure

This UDP with DPDK solution is quite simple and clear, especially useful for read operations. We still believe that TCP protocol can ensure a higher transmission quality, as it provides flow/congestion control and retransmission policy that are significant for write/SET operation or situation when we don't have ideal transmission environment. Therefore, user-space TCP/IP stack with DPDK is necessary to build a high performance networking. Among related research, mTCP[12] is a open-source, high-performance user-space TCP stack for multicore systems.

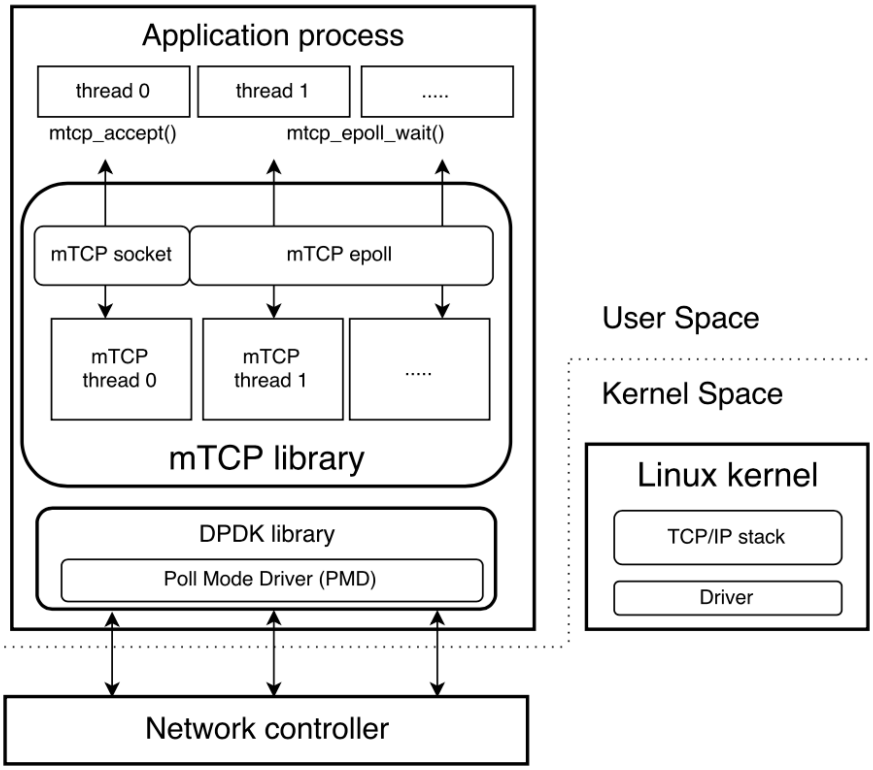


Figure 5 — mTCP and Intel DPDK

Figure 5 illustrates architecture of mTCP on DPDK library. mTCP uses per-core accept queue, connection locality, batched function calls, and batched packet I/O to enhance the network processing ability and reduce the overhead that linux kernel introduces. There is a pair of application thread and mTCP thread bind on each core. The application thread is responsible for handling user application logic, and mTCP thread is responsible for handling the events notified. Each mTCP thread executes a polling routine, constantly getting

packets from its RX queue, parsing the packets, notifying upper application threads, and sending packets to its TX queue. Since DPDK does not provide TCP/IP stack directly, the author of mTCP composed its own version of TCP stack in a state machine manner. For thread communication, each application thread and mTCP thread pair communicates by the pipe or signal, so there is a context switch between these two threads. The context switch overhead can be reduced when the packet size is mall and concurrency level is high. Most important is that mTCP provides BSD-like socket and epoll-like event-driven interface, so migrating existed event-driven applications does not require much modification. Previous migrating work on SSLShader and lighttpd has shown application performance improvement of 33% and 320%, respectively.

# Chapter 3

# Design and Implementation

## 3.1 thread model

As we present in Figure 1, original MemC3 uses single listener, multi-worker thread model. All the threads share a global connection array. The listener thread is responsible for listening the incoming connection requests. When a connection request arrives, the listener thread creates a connection item, and distributes the item to a worker thread in round-robin manner. The worker thread will be notified of the event, and call event handler to get the item and process the following request.

In our design, we replace the single listener thread with multiple equivalent worker threads. Each one keeps its own connection arrays, serving both connection listening and request handling. As mentioned before, mTCP binds one application thread and one mTCP thread on a same core. The mTCP thread polls the burst packet I/O on its designated RX/TX queue of NIC. Figure 6 illustrates our proposed thread model.

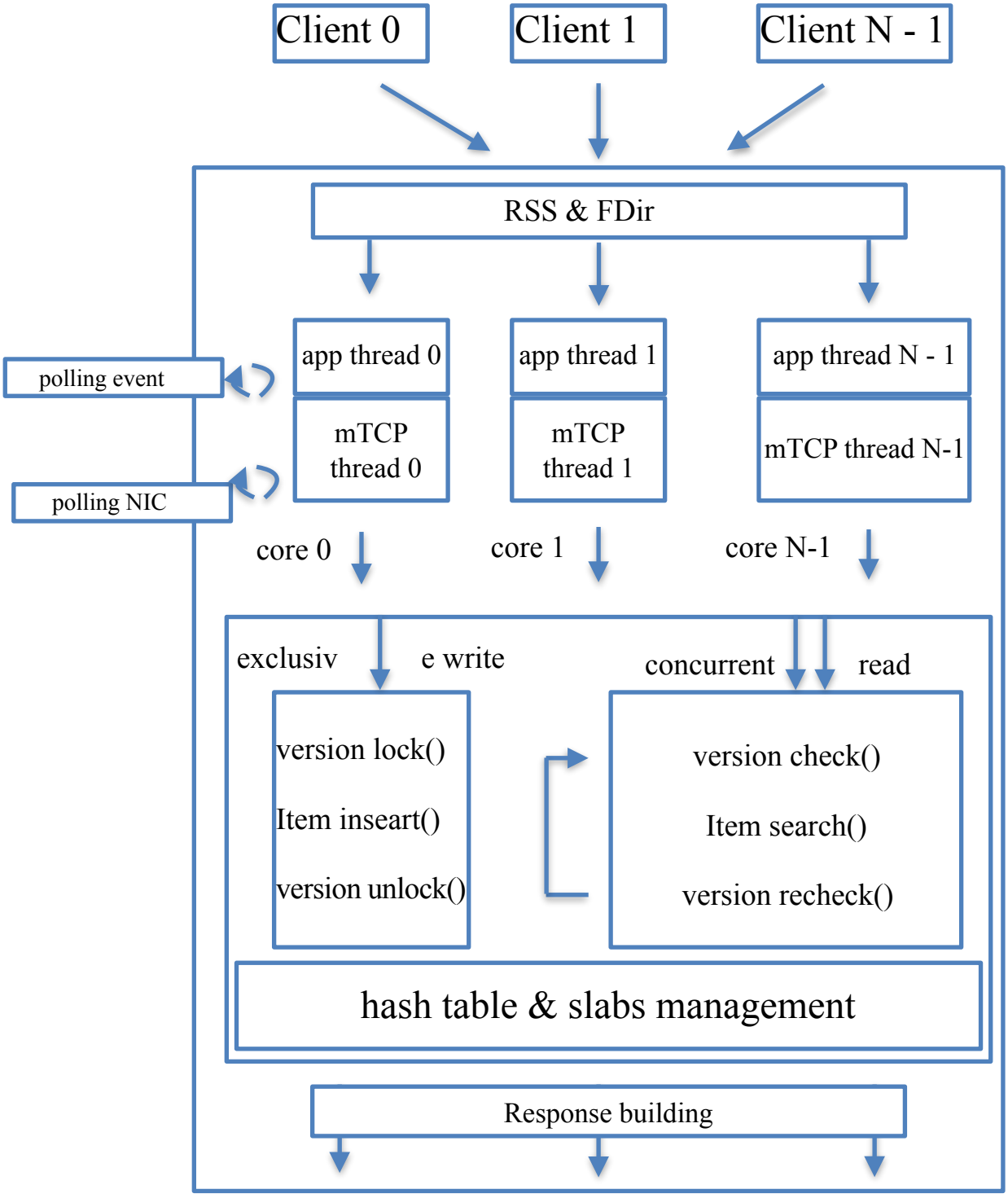


Figure 6 — proposed thread model

## 3.2 socket and epoll APIs

Original MemC3 uses BSD socket API and libevent as event handling backend. In this way, listener and worker threads should register event type, event handler, and add these to event base in advance. BSD socket APIs can be directly replaced by mTCP socket APIs since they provide almost same function with the original one. The problem is the I/O multiplexing API part. Previous paper suggests two approaches. The first one is straightforward that to replace libevent API with mTCP epoll API. The second one is to implement an mTCP epoll back-end inside libevent. The experiment result showed the former one outperforms the latter one, so in our implementation we take the first approach. Since we no longer use libevent as epoll back-end, all the event related functions and data structures have to be modified or totally removed so it makes our work troublesome.

First, we should remove all the pipes in the original version. This part is easier because of our proposed multiple worker thread model where each thread works independently and pipes are not needed for message passing. Note that this remove is not achievable in stock memcached because pipe is still required to pass pause/restart signal to worker threads when hash table expanding happens. Second, we should combine original listener thread's function with worker thread's part of work altogether on each single application thread. Each

application maintains its own connection array, and loops on detecting arrived events and then handling each event respectively. If the event file descriptor equals listening socket, calling accepting function, creating a new socket id, and adding it to event pool. If the event file descriptor equals a connection socket id, to handle the event in the drive machine. Figure 6 illustrates the event processing control flow.

Aside from the application thread that serves the application logic, there is an another mTCP worker thread on the same core to do packet I/O's part of work. Each mTCP worker thread runs the following routine: (1) fetch packet from its RX queue in burst manner (32 or 64 packets). (2) Then process each packet first by analyze the packet type according to its header. Only IPV4 packet can be handled in the current version. The packet then is processed in a state machine according to the connection state stored in local mTCP connection data structure kept on that core. If the packet should be handle by the upper user application layer, an event will be registered in event queue shared by the application thread and mTCP thread. (3) Next, check if there is any sending event finished by the application thread. If so, calling burst sending function to add the packets to TX queue. According to real time experiment

evaluation, we found there exists lock overhead on original mTCP design because many locks are used for synchronizing the application threads and mTCP threads. This overhead can be greatly reduced by using fine-grained lock or removed unnecessary one.

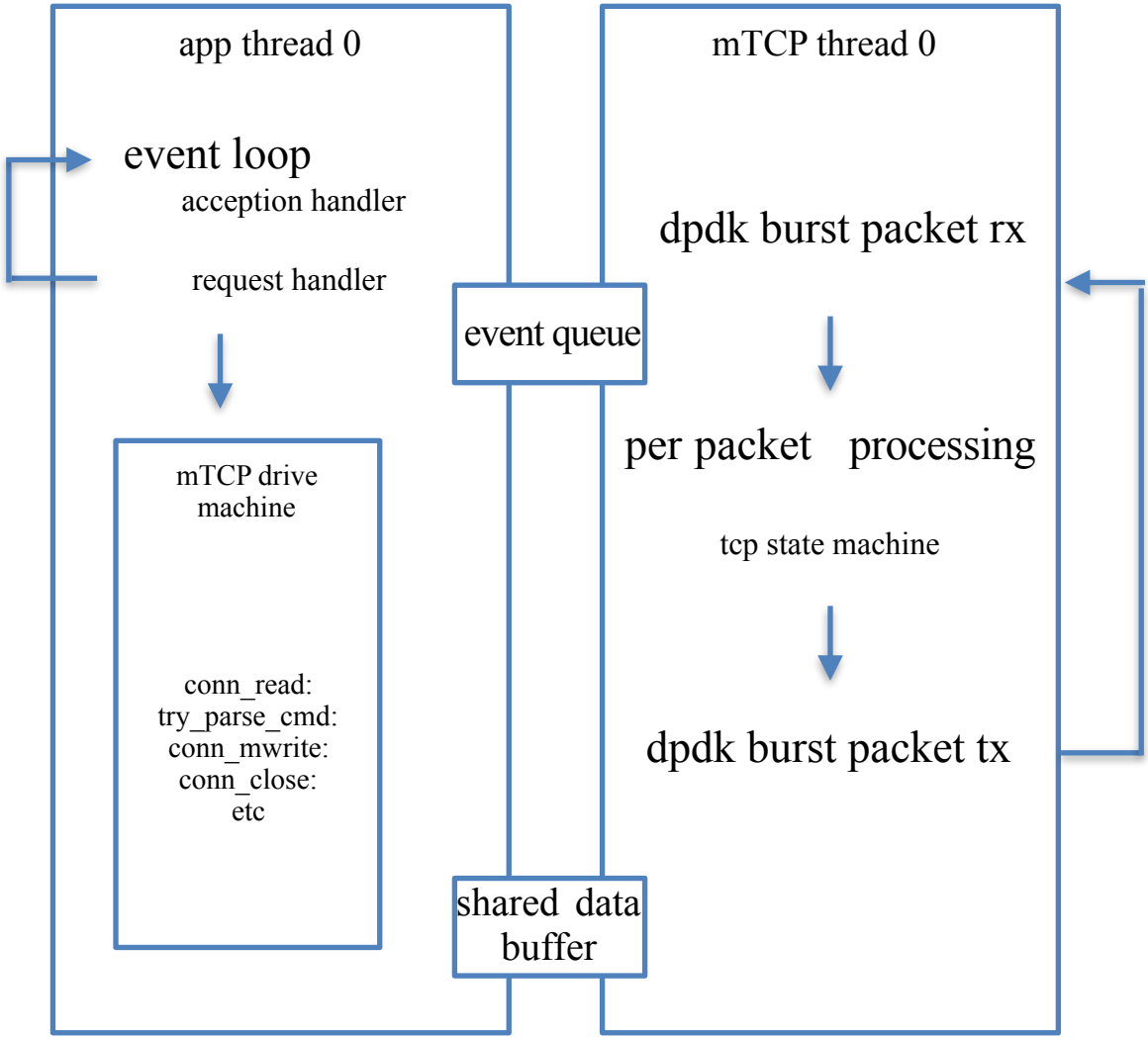


Figure 7 — event processing control flow

# Chapter 4

# Evaluation

After coding and debugging the new MemC3 with user-space TCP stack feature, we setuped test environment and tested the system performance.

We have conducted experiments on three machines with same hardware and software configuration. We set one machine as server, which was installed with our newly built system, and the other two machines as client to generate workload. Each machine consisted of 4 Intel(R) Core(TM) i7-4790 CPU 3.60GHz processors, 32GB DRAM, Intel X520 10 Gigabit Ethernet NIC. Hyper Threading was enabled. Each machine run Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-66-generic x86_64), DPDK 16.11, and mTCP (commit hash: 0a1d3d0, Apr 4, 2017). Two clients run Mcblaster [13] as benchmark. Mcblaster was designed to drive a synthetic cache lookup workload and measure performance of the memcached instance. For the first step, we generated 1 million keys with 64 Byte value to the server. Then we generated set/get request simultaneously on two client machines.With same workload, we gradually increased the worker thread number, and also increased set/get rate on the client side. We chose the rate that was less than 1 ms average latency as the best throughput of the server.

./mcblaster –p 11211 -t 16 -z 64 –k 1000000 –r <rate> –W 0 –d 60 <memcached host>

Figure 8 shows the maximum throughput with an average RTT less than 1ms as core counts increase. This figure shows experiment with all queries were set queries. Because mTCP does not support binding more than one thread on a physical core, we can only see the maximum result on 4 worker threads. Our modified version can have about 50% performance enhancement compared with

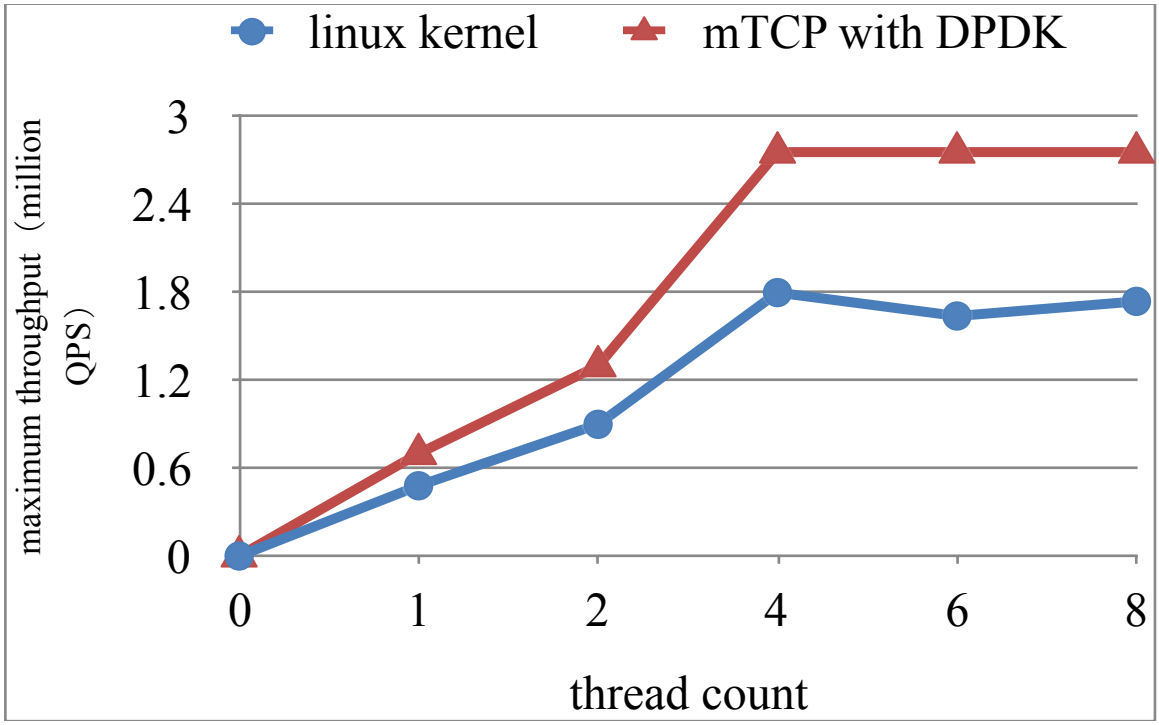


Figure 8 — Maximum throughput with avg. RTT < 1ms

the original version with 4 cores used. We also adjusted set/get ratio to 95:5, a more common workload in real world application, result showed similar throughput and performance enhancement with the former test.

We also compared the latency distribution of two systems in Figure 9. We can see that DPDK can greatly improve the latency performance, for both average latency and maximum latency. Even with workload when original MemC3 suffered from severe latency lag, our modified version could still finish more than 90% of the queries within a relative short time. We believe this is because DPDK can process packet directly in user space, and burst packet I/O can amortize the per packet overhead.

In order to see the system performance on a bigger machine (with more cores), we also tested on two 24 core machines with 10G NIC directly connected. But we divided one machine evenly into two parts, each has 12 cores and they work as a memcached server and a client, respectively. And another machine was split into 4 clients. Figure 10 shows the experiment result that, compared with the smaller machine (4 cpu cores), our application leads to an average 45% performance improvement w.r.t the original kernel stack. But the figure also

shows that the performance doesn't scale ideally or linearly when the number of core increases.

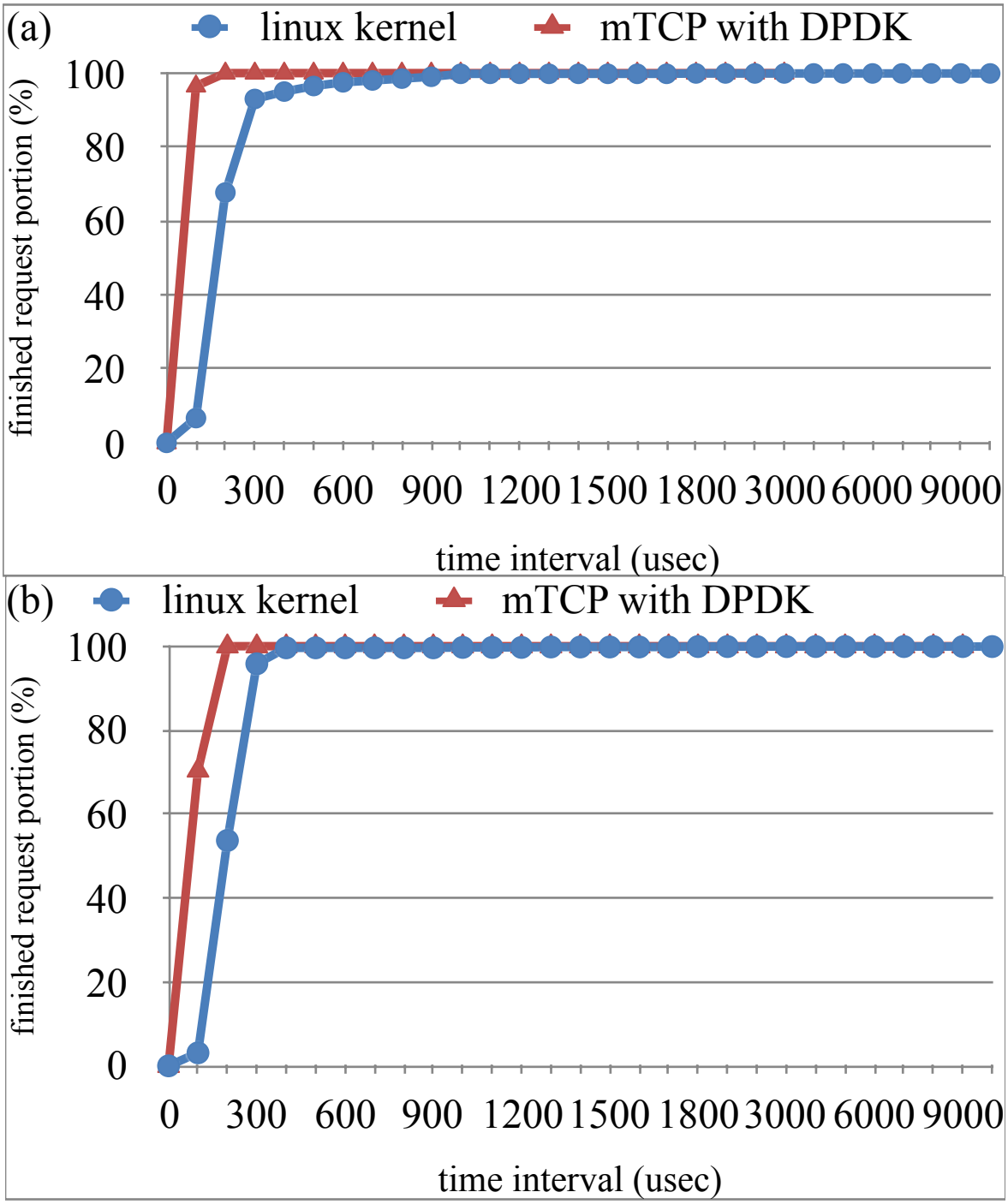


Figure 9 — latency distribution (a) thread count = 1 (b) thread count = 4

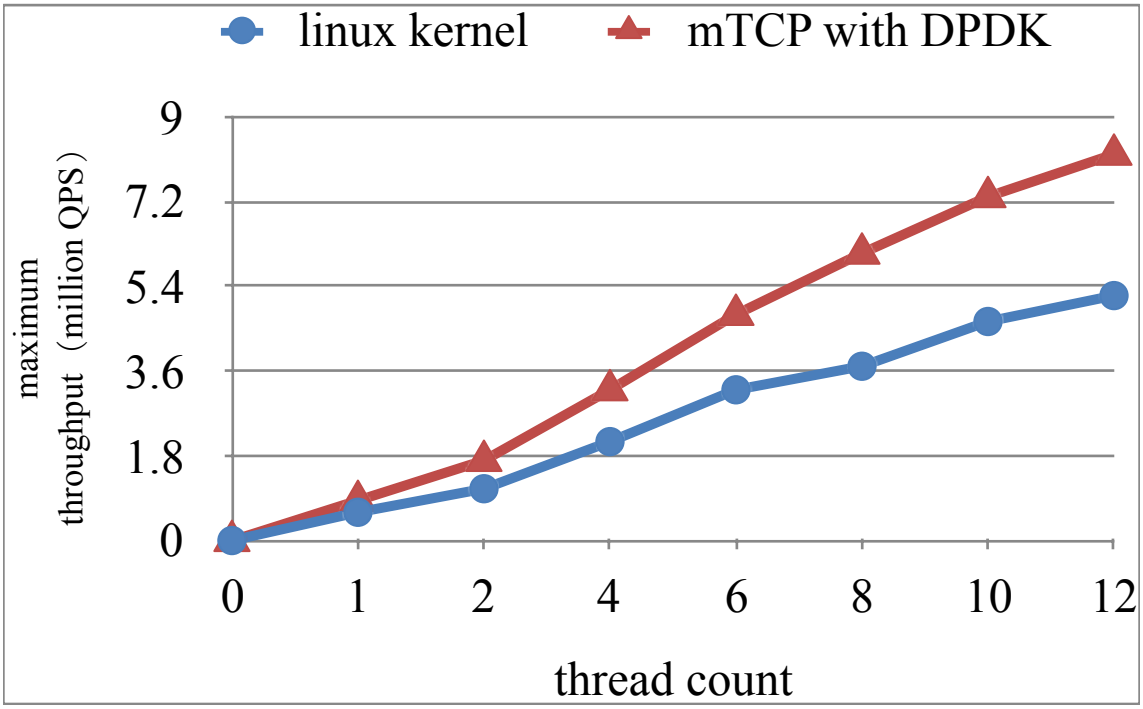


Figure 10 — Maximum throughput with avg. RTT < 1ms on 24 core machine

# Chapter 5

# Conclusion and Future Work

This paper proposed the design of optimized memcached with user-space TCP stack based on mTCP and Intel DPDK. Intel DPDK is a set of libraries that provides fast packet processing ability. And mTCP is a high-performance user-level TCP stack designed for multicore systems. By using these two technologies, we optimized existed best-performed version of memcached, MemC3 to eliminate network overhead. We modified the system thread model and event handling mechanism to fit mTCP architecture and then to be able to maximize the throughput gain introduced by DPDK. Our experiment result shows that the optimized memcached can achieve 45-50% performance improved compared with the original one. Latency performance also can be greatly improved by using burst packet I/O and user-space packet processing.

One potential problem is that, while the workload skew level is increasing, the performance of our system can degrade because cache miss may become more common. To overcome this problem, cache partition is an ideal solution and has already been verified in some key-value storage systems[11].

However, since mTCP also has the core-affinity feature, and packets or connections are hashed to each core by 5-tuple according to general case, partitioning key-value item on each cache is not easy to implement. Further research still need to be done in the future.

# Bibliography

[1] Memcached. https://memcached.org/

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, 2012.

[3] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In NSDI, pages 429–444, 2014.

[4] K. Zhang, K. Wang, Y. Yuan, L. Guo, R.B. Lee, and X.D. Zhang . Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. In *Proceedings of the* VLDB, 2015.

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

[6] S. Thongprasit, V. Visoottiviseth, and R, Takano. Toward Fast and Scalable Key-Value Stores Based on User Space TCP/IP Stack. In *Proceedings of the AINTEC '15*, 2015.

[7] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In NSDI, pages 371–384, 2013.

[8] A. Wiggins, and J. Langston. Enhancing the Scalability of Memcached. Intel document, 2012.

[9] Libevent, http://libevent.org/.

[10] Intel DPDK. http://dpdk.org/.

[11] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: A cache-partitioned hash table. In PPoPP, pages 319–320, 2012.

[12] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In NSDI, 2014.

[13] Mcblaster, https://github.com/fbmarc/facebook-memcached-old/tree/master/test/mcblaster/.

# 초록

멀티스레드와 분산 key-value 스토리지인 memcached는 클라이 언트의 쿼리에 대해 빠르게 응답하고 tail latency도 많이 줄일 수 있는 장점이 있기 때문에 현재 많은 웹 애플리케이션과 서비스에 서 사용되고 있다. 본 논문에서는 memcached에 multi-core 시스 템의 환경에서 성능이 향상시킬 수 있기를 위하여 intel DPDK를 기반으로 한 user-space tcp stack을 이용하여 최적화시켰다. 이 런 방법을 이용해서 서버와 클라이언트를 통신할 때, 특히 small 메시지가 위주인 경우에서, 리눅스 커널 영역을 통과 하지 않아서 네트워크 쪽에 오버헤드가 많이 감소될 수 있다. 본 논문은 intel DPDK를 기반의 mTCP가 user-space tcp stack으로 쓰이고 가 장 최적화된 memcached인 MemC3을 바탕으로 하여 구현하였다. 실험결과는 MemC3과 DPDK가 잘 맞춰서 멀티 코어를 사용하는 경우이라도 50% 성능적인 향상을 유지할 수 있다는 결론을 제시 하였다.

주요어: Key-Value 스토리지, DPDK, user-space tcp stack, 멀티 코어

학번: 2015-23304