High Performance Memcached

Lijing Chen

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

This is an example of `infthesis` style. The file `skeleton.tex` generates this document and can be used to get a "skeleton" for your thesis. The abstract should summarise your report and fit in the space on the first page. You may, of course, use any other software to write your report, as long as you follow the same style. That means: producing a title page as given here, and including a table of contents and bibliography.

Acknowledgements

Acknowledgements go here.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Software object caches, such as Memcached, are crucial to achieving high throughput and low response latency in a datacenter setting. A well-known problem with Memcached deployments is the high overhead incurred by the kernel's TCP/IP stack in processing of individual requests. This project will evaluate the performance of a vanilla Memcached setup, followed by an assessment of the effect that user-level networking and TCP offload have on Memcached.

1.2 Project Aim

This project is consisted of following parts.

Baseline Tuning We are going to first tune the original memcached and use the result as a baseline for following testing. The components we are going to tune are thread number, process number and IRQ affinity.

User-level network Evaluation Evaluate to what extent applying user-level networking will influence the performance of memcached.

TCP Offload Evaluation Evaluate the influence of multiple offloading techniques.

Chapter 2

Background

2.1 Memcached

Memcached is a ”high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load.” Despite the official description aimed at dynamic web applications, Memcached is also used as a generic key value store to locate servers and services.

2.1.1 Memcached protocol

Memcached support two transport-layer protocols: TCP and UDP. They have some small difference. The basic protocol we are going to use in our experiments are described as Table 2.1. Note that there are many other command provided by memcached, but we are only going to test the most basic ones as listed. Comparing with TCP, the UDP protocol added an additional binary header for controlling. The binary header is described in Table 2.2.

Operation	Description
get [key]	Get the value of given key
set [key] [flags] [length] value	Set the value of given key
delete [key]	delete a key provided by the argument.

Table 2.1: Basic Memcached protocol

Position byte	Content	Description
0-1	Request ID	The id of the request
2-3	Sequence number	The id of the request
4-5	Total number of datagrams	The id of the request
6-7	Reserved	Must be 0

Table 2.2: Memcached UDP frame header

2.1.2 Memcached parameters

2.2 User-level networking

One performance bottleneck of modern high-performance applications is the kernel TCP/IP stack. There is a trend in industry to replace kernel TCP/IP stack with user-level TCP/IP stack.

2.2.1 Data Plane Development Kit

The Data Plane Development Kit is ”a set of libraries and drivers for fast packet processing”[3]. It runs mostly in user-space, and it serves as a platform for developer to develop high performance networking applications.

DPDK provides developers with a framework, which contains a set of libraries for different hardware and software environments. DPDK encapsulates the low level environmental detail by creating Environment Abstraction Layer (EAL). Developers can program with a general interface, and link the library compiled for each specific environment, instead of coding with specific API from different hardware, operating systems etc.

2.2.2 Seastar

DPDK doesn’t contain a TCP/IP stack. In order to migrate TCP/IP based applications to use DPDK, developers need to implement their own TCP/IP stack using DPDK provided API.

Seastar[5] is a framework that provided us with a ready-to-use user-level TCP/IP stack based on DPDK.

2.3 TCP Offload

TCP Offloading is another technique quite widely used in modern NIC cards. In order to alleviate the load of CPU, NIC manufacturers provided different level of offloading

features in NIC, this means some of the operations like checksum, fragmentation can be processed by NIC directly instead of host CPU.

2.3.1 Full offload vs Partial offload

There are two type of TCP Offloading. Full offload means we offload all kernel TCP/IP stack to our NIC. Partial offload means we only offload some part of TCP/IP stack to our NIC, like checksum, segmentation etc.

TOE implementations also can be differentiated by the amount of processing that is offloaded to the network adapter. In situations where TCP connections are stable and packet drops infrequent, the highest amount of TCP processing is spent in data transmission and reception. Offloading just the processing related to transmission and reception is referred to as partial offloading. A partial, or data path, TOE implementation eliminates the host CPU overhead created by transmission and reception. However, the partial offloading method improves performance only in situations where TCP connections are created and held for a long time and errors and lost packets are infrequent. Partial offloading relies on the host stack to handle controlthat is, connection setupas well as exceptions. A partial TOE implementation does not handle the following:

2.3.2 TCP/IP checksum offload

TCP/IP checksum offload let the network adapter to do the calculation task of verifying the checksum of packet received, which is done by CPU originally. This technique can reduce the CPU utilization.

The TCP/IP checksum offload technique moves the calculation of the TCP and IP checksum packets from the host CPU to the network adapter. For the TCP checksum, the transport layer on the host calculates the TCP pseudo-header checksum and places this value in the checksum field, thus enabling the network adapter to calculate the correct TCP checksum without touching the IP header. However, this approach yields only a modest reduction in CPU utilization.

2.3.3 Large send offload

Large send offload (LSO), also known as TCP segmentation offload (TSO), frees the OS from the task of segmenting the applications transmit data into MTU-size chunks. Using LSO, TCP can transmit a chunk of data larger than the MTU size to the network adapter. The adapter driver then divides the data into MTU-size chunks and uses the prototype TCP and IP headers of the send buffer to create TCP/IP headers for each packet in preparation for transmission.

LSO is an extremely useful technology to scale performance across multiple Gigabit Ethernet links, although it does so under certain conditions. The LSO technique is most efficient when transferring large messages. Also, because LSO is a stateless offload, it

yields performance benefits only for traffic being sent; it offers no improvements for traffic being received. Although LSO can reduce CPU utilization by approximately half, this benefit can be realized only if the receivers TCP window size is set to 64 KB. LSO has little effect on interrupt processing because it is a transmit-only offload.

Methods such as TCP/IP checksum offload and LSO provide limited performance gains or are advantageous only under certain conditions. For example, LSO is less effective when transmitting several smaller-sized packages. Also, in environments where packets are frequently dropped and connections lost, connection setup and maintenance consume a significant proportion of the hosts processing power. Methods like LSO would produce minimal performance improvements in such environments.

2.4 Performance Analysis

2.4.1 Linux perf

Perf is a tool provided by linux to Perf began as a tool for using the performance counters subsystem in Linux, and has had various enhancements to add tracing capabilities.

2.4.2 FlameGraph

Chapter 3

Methodology

3.1 Quality of Service

Before we start to do experiments, it's important to set up a goal for our benchmarking. For this project, a sufficient quality of service (QoS) will be under 1 millisecond for the 99th percentile request latency.

3.2 Hardware

The Server we are using is described in Table 4.1.

3.2.1 Server

Component Name	Device
CPU	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (40 cores)
Network Interface Card	Mellanox Connect-X 4

Table 3.1: Hardware Table

The Network Interface Card we are using is (mlx5) [1].

3.2.1.1 Network topology

For DPDK tests, we directly connect the NIC in two servers using network cable due to device limitation. The reason we use this topology is the NIC only supports Infiniband ports, but we don't have a switch that supports Infiniband. But then don't make a large difference and it's sufficient for us to do our experiments.

3.3 Benchmark

3.3.1 Open-loop vs Closed loop

There are two kinds of load testers: open-loop and closed-loop. Open-loop load testers send a new request only when the previous one has returned (only one outstanding request), while closed-loop load testers can send a new one even the previous one is pending (allow multiple outstanding requests).

3.3.2 Treadmill

Treadmill [6] is a open-source load tester for memcached developed by Facebook. It's a typical open-loop load tester. It only supports TCP for memcached. For our experiments involving UDP, we implemented our own version of treadmill for UDP.

3.3.3 Memaslap

Memaslap is a load tester provide by libmemcached [2]. It's a closed loop load tester.

Chapter 4

Baseline Tuning

In this chapter, we are going to tune the stock memcached by changing built-in parameters and system configurations to set up a baseline for following chapters to compare with.

4.1 Multithreading

In this section, we will evaluate how different number of threads can tail latency. Memcached supports multithreading by default, and it used spin lock for synchronization from all threads. We increase the throughput linearly find out the maximum throughput memcached can achieve when restricting the 99 percentile latency less than 1 millisecond.

Intuitively and empirically, the best outcome we expected is when the number of memcached threads is the same as server CPU cores, as suggested by Leverich and Kozyrakis[4],

4.1.1 Setup

The way we change the thread number is changing the argument `-t` passed to memcached. The detailed description of this parameter is mentioned in Section 2.1.2.

4.1.2 Result

We can see the result in Figure 4.1. According to the Figure 4.1, we can see that the curve representing 4 threads (red curve) is the last one to hit the 1ms latency line, which is consistent with our previous empirical prediction. But we can also see from the figure, the 4 threads line isn't always the one with smallest latency, actually in the section from 160k throughput to 200k throughput, it's defeated by the curves representing 5 threads and 6 threads. But those two lines ("Thread 6" and "Thread 5") are getting

Component Name	Device
CPU	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (40 cores)
Network Interface Card	Mellanox Connect-X 4

Table 4.1: Hardware Table

higher latency after all 200k latency. One possible reason of this phenomenon can be CPU utilization. The curves are linearly increasing which means they are still not so close to maximum throughput, since the CPU is still under-utilized, more threads can increase some performance even they added some context switching overhead.

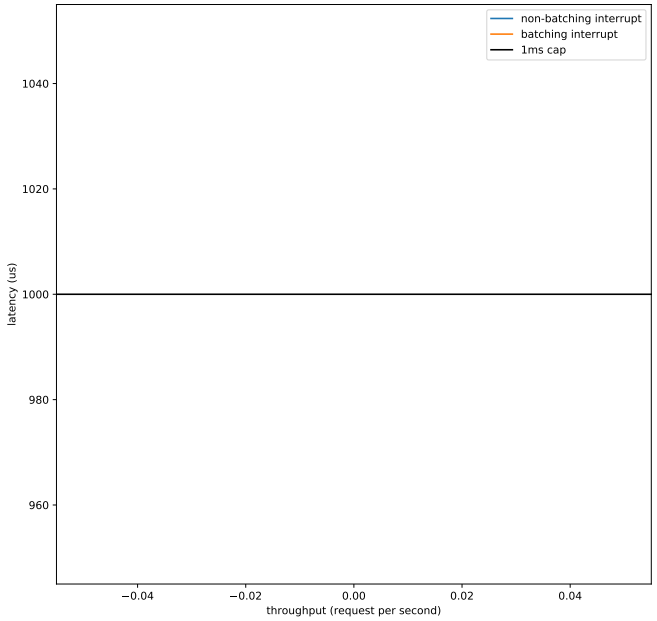


Figure 4.1: Multithreading

4.2 Interrupt batching

Interrupt batching is a technique for improving the performance of packet processing under heavy workload. The basic idea of interrupt batching is that instead of raising a interrupt for every packet received, batch multiple packets in one interrupt. This can significantly reduce network and CPU resources because message batching results in fewer packet to process, effectively reducing the overhead of processing a packet.

4.2.1 Setup

We can use `ethtool` to change interrupt batching option of our NIC. The value is recommended by IBM(add cite here). The command we used is as follows.

```
$ ethtool -C  p2p1 rx-usecs 5 rx-frames 5 tx-usecs 5 rx-usecs 5
```

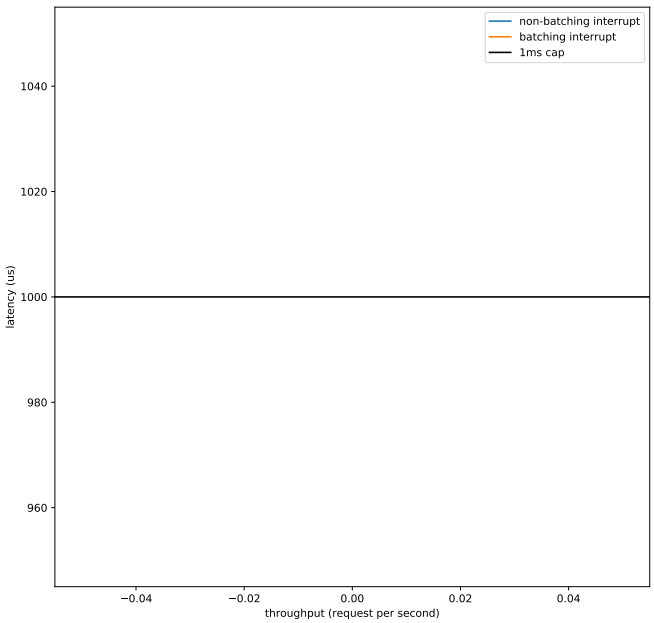


Figure 4.2: Batching vs non-batching interrupts

4.2.2 Result

The result is shown in Figure 4.2. We can see that the curve representing batching interrupts can achieve a better throughput under 1ms cap than non-batching interrupts. So batching interrupts does help for improving tail latency.

4.3 Multiprocessing

In this section, we are going to use mcrouter to test the performance of multiprocessing.

4.4 Setup

We first need to setup multiple memcahed instances.

4.4.1 Result

Chapter 5

Contention Analysis

5.1 Core contention

5.2 SMT Thread contention

Chapter 6

User-level Networking Optimization

6.1 DPDK

6.1.1 DPDK Setup

In this section, we are going to go over how DPDK is set up.

Setup hugepage mapping Hugepages is a mechanism that allows the Linux kernel to utilize the multiple page size capabilities of modern hardware architectures. Linux uses pages as the basic unit of memory, where physical memory is partitioned and accessed using the basic page unit. The default page size is 4096 Bytes in the x86 architecture. Hugepages allows large amounts of memory to be utilized with a reduced overhead.

Compile and insert DPDK kernel modules First we need to do `make config` to tell the compiler which architecture we are targeting. We need to generate binaries working on `x86_64-native-linuxapp-gcc`. The command is as follows.

```
$ make config T=x86_64-native-linuxapp-gcc
```

Some NIC drivers are not enabled by default, including the one we are going to use (mlx5). In this case, we need to manually enable it in `config/common_base` file.

Then we can use `make` command to compile DPDK. After successfully compiled DPDK, switch to output folder and insert kernel module: `igb_uio.ko`.

```
$ insmod igb_uio.ko
```

Configure NIC First we need to unload the NIC from system kernel driver using `ifconfig`.

```
$ ifconfig p2p1 down
```

Then we can use the tool provided by DPDK to bind NIC to kernel driver. `dpdk-devbind.py` This is common step for most NIC, but for our NIC specifically, this step is not necessary.

6.2 Experiment

In this section, we are going evaluate the performance between DPDK implementation and stock memcached.

6.2.1 Single Core performance

We are going to compare the latency performance DPDK memcached with stock memcached, each with only one core. The Figure 6.1 shows the result.

The curve of stock memcached faded at about 200,000 rps, which means that is the maximum throughput stock memcached can achieve. We can see DPDK version can achieve a much higher maximum throughput. Also, when the throughput is the same, the DPDK one also has lower latency than stock one.

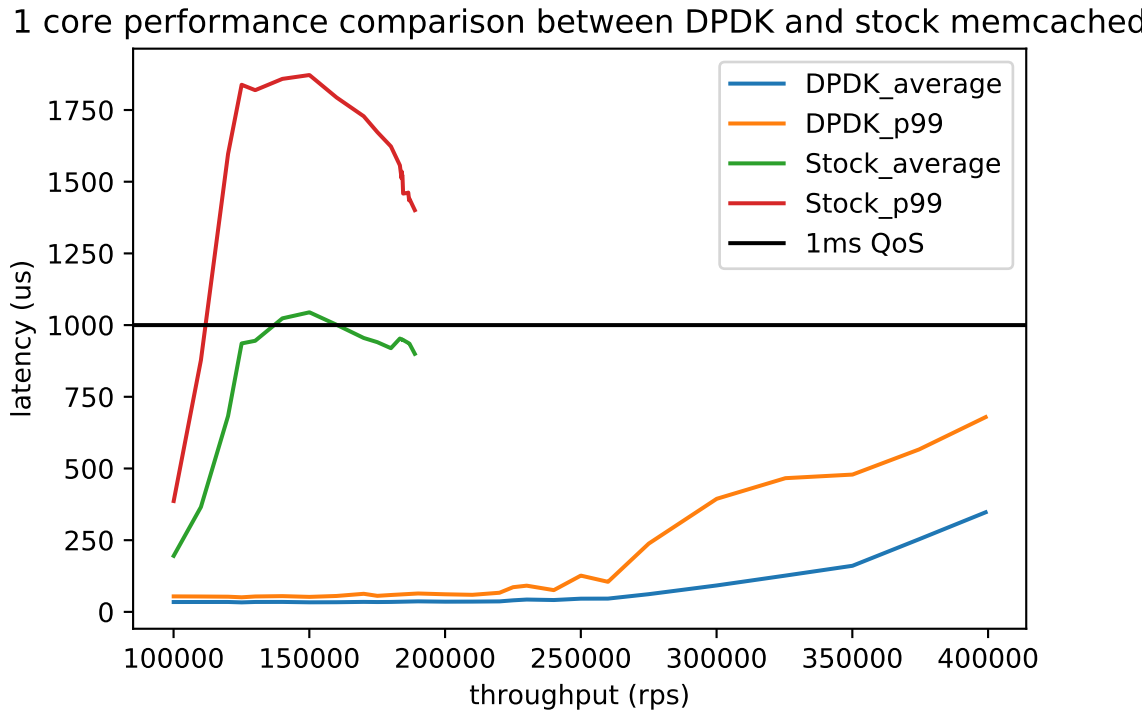
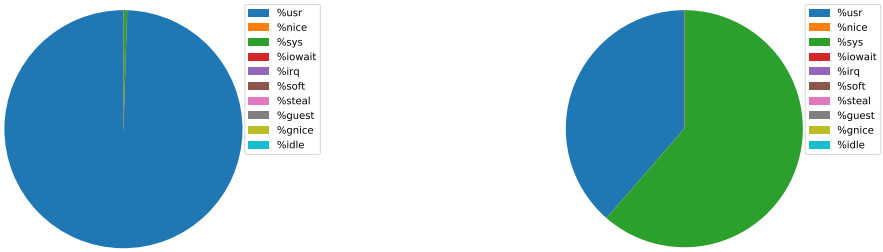


Figure 6.1: DPDK



(a) Memcached (DPDK) busy time CPU utitliza-tion

(b) Memcached (DPDK) idle time CPU utitliza-tion

Figure 6.2: DPDK CPU utilization

6.2.2 Multiple Core performance

6.3 Analysis

6.3.1 CPU utilization analysis

Different event One major difference between DPDK and stock is DPDK made use of its poll mode driver. Different from original interrupt based Linux kernel network stack, poll mode driver use busy waiting to avoid using interrupts, which can significantly reduce the latency. When using the poll mode driver, the CPU utilization will always be 100%.

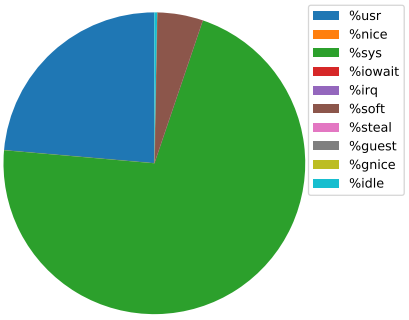


Figure 6.3: Memcached (stock) busy time CPU utitlization

Chapter 7

TCP Offloading

7.1 Setup TCP Offloading

First, we use **ethtool** to check device offloading. We can show the support for TCP Offloading by executing following command:

```
# ethtool -k p2p2
```

7.2 Checksum Offload

7.3 Large Send Offload

Bibliography

[1] Hiroki Arimura. Learning acyclic first-order horn sentences from entailment. In *Proc. of the 8th Intl. Conf. on Algorithmic Learning Theory, ALT '97*, pages 432–445, 1997.

[2] Chen-Chung Chang and H. Jerome Keisler. *Model Theory*. North-Holland, third edition, 1990.

[3] Intel. Intel Data Plane Development Kit. http://www.intel.com/go/dpdk, 2016.

[4] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 4:1–4:14, 2014.

[5] Scylla. Seastar Framework . http://www.seastar-project.org/, 2016.

[6] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 456–468, 2016.