Individual Diploma Thesis

MEMCACHED PERFORMANCE AND POWER CHARACTERIZATION

Nanour Kazarian

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

December 2022

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

MEMCACHED PERFORMANCE AND POWER CHARACTHERIZATION

Nanour Kazarian

Supervisor

Yiannakis Sazeides

The Individual Diploma Thesis was submitted for partial fulfilment of the requirements for obtaining a degree in Informatics of the Department of Informatics of the University of Cyprus

December 2022

Acknowledgements

I would like to thank my supervisor Mr Yiannakis Sazeides for suggesting this topic to me, also for his patience, guidance and support, as well as Mr Haris Volos and Mrs Georgia Antoniou, for their guidance throughout this project.

Moreover, I would like to thank my family, Husband and friends for their everyday support and encouragement.

Summary

Nowadays, two of the most important problems of Servers is the power consumption and the performance and the process of finding techniques that will lead to the reduction of power consumption and to improve the performance. Because reducing power consumption will not always assure that we will have the same performance as before**.** Today, most of the giant tech companies have embraced Microservices architecture one of the most Famous microservice application is used is the Memcached which is a widely used in-memory caching solution in large-scale searching scenarios. The most pivotal performance metric in Memcached is latency, which is affected by various factors. These Moderns servers must be good at performance but at the same time to not consume much power because consuming more power it means to have more cost also not be able to work in higher frequencies.

This Thesis is based on Memcached to understand its Characterization by exploring different configuration of the machine that it is running on. The three base configurations was the client payload, SMT and also different idle states (C-states) configurations. This Research finds out that having the SMT enabled has worst latencies than the SMT disabled from the perspective of performance, but in the other hand from the perspective of Power having SMT on gave us lower power consumption than the SMT off. Having C-state enabled gave us the worst performance at some point for payloads which had idle time, but in the other hand having C-state Enabled always is the best config for power saving. In Contrast having high Payloads C-state Enabled and Disabled where almost the same at performance level, So that choosing a c-state enabled at high payloads will achieve both performance and power saving goals.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Today one of the important issues of Data centres is the power and performance relation, because Data centres industry accounts for around 4% of global electricity consumption which means it is important to reduce the power consumption of these machines to reduce pollution, reduce operating cost but very important question what will be the trade off in performance of these centres which is also an important matter for end users. In my thesis, the main topic is about Memcached which a widely used in-memory caching solution in large-scale searching scenarios. Today Memcached is used by many other systems, including YouTube, Reddit, Facebook, Pinterest, Twitter…etc. So, checking and researching its power and performance characteristics is very trending topic nowadays.

1.2 Previous Work[1]

Many services leverage Memcached as a caching layer between a web server tier and back-end databases. Such applications are demanding; therefore, a series of changes are needed for fully taking advantage of the caching system. There have been many studies to understand and research the different performance and power characteristics of Memcached In this research we will be concentrating on the Performance and power consumption of Memcached with different configurations and it is different than previous research is that we checked also Other idle State Configurations which wasn't tested previously also checking a combination of configurations which was SMT and C-states.

1.3 Contributions

In the beginning it was very important to understand modern servers Hardware architecture and their power characterizations. Then, understand and study the software applications used by the servers, especially Memcached. Furthermore, researching and studied some past papers related to Memcached to have a good background. And then finding out new configurations that would be interesting to test the performance and power results of Memcached. Also, finding a way to calculate Memcached Server side Average and Tail Latencies which was not available previously.

1.4 Thesis Structure

Chapter 2:

We give a short review of the various power management elements and procedures utilized in modern processors, also explaining the problem definition of this thesis.

Chapter 3:

Introduction to Memcached, data structure, parameters used how to run Memcached, how to set, get, update keys in Memcached and the statistics that we can retrieve.

Chapter 4:

This chapter will define the Workload Generator Mutilate (mcperf), the different parameters used and their meanings also some examples on how to run a workload generator on Memcached.

Chapter 5:

This chapter will describe the Code added to Memcached to measure server-side tail and average latency. In addition to the validation used and the trade-off of this measurement to the Memcached.

Chapter 6:

The combinations of the experiment's scenarios held to monitor different performance metrics and statistics with different client-side configurations. And the methodology of the experiments.

Chapter 7:

The Results graph, Observations, and discussions.

Chapter 8:

General conclusions, describing some lessons learned and also stating the future work needed.

Chapter 2

Background and Problem Definition

2.1 Modern Servers' Software Architecture

The design and organization of parts into systems, which serve as the building blocks of software, is referred to as software architecture. The structural build of the software program and how its components interact are described by its software architecture. An architectural pattern is the idea that establishes the software organization schema for these software systems.

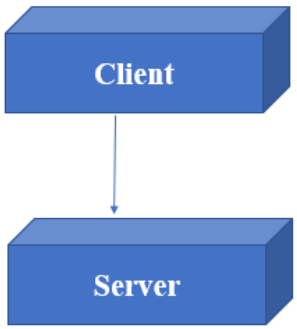


Figure 1: Client Server Architecture

In the client-server architecture patterns, there are two main components: The client, which is the service requester, and the server, which is the service provider. Although both client and server may be located within the same system, they often communicate over a network on separate hardware.

In this Architecture pattern the client component initiates certain interactions with the server to generate the services needed. While the client components have ports that describe the needed services, the servers have ports that describe the services they provide. Both components are linked by request/reply to connectors. The client-server model is related to the peer-to-peer architecture pattern and is often described as a subcategory of this pattern.

2.2 Introduction to Modern Servers Architecture and Hardware[2]

The first servers were mainframes, which date back to the 1950s and 1960s. However, the history of servers as we know them now began in the 1990s with the development of web and rack-mounted servers. Nowadays, servers can be found in a variety of settings, from tiny businesses to major corporations, and they offer a variety of capabilities.

The motherboard, CPU, random access memory (RAM), and storage are the main elements of a server's hardware architecture. At the centre of the server is the motherboard, which acts as a bridge for attaching external devices and connecting system components. The two primary varieties of motherboards are Advanced Technology Extended and Low-Profile Extension and Mini Information Technology Extended motherboards catering to the needs of smaller form factors. The processor, or central processing unit (CPU), resides on the motherboard. CPU components include the arithmetic logic unit, floating point unit, registers, and cache memory. A server might also contain a graphics processing unit (GPU), which can support applications such as machine learning and simulations. And the arrival of tensor processing units and neural processing units offers additional levels of processor specialization. Servers are designed for high performance but is important to have power. Today to Modern servers have been added many new technologies added to save power some of these technologies are the following:

2.2.1 Idle Power States

2.2.1.1 C-States [3]

CPU Cores can lower their power consumption when they are not in use by enabling power saving states. Various C-states are supported by contemporary CPUs; for example, In Intel's Skylake which will be the machine that we will be using for all the experiments; Its architecture provides the following four: C0, C1, C1E, C6. Following a table which includes in each state what reduces take place and the approximate time to wake. The C-states are controlled by the hardware, and it can be enabled partially disabled (disabling some stages) and fully disabled. (Figure 2)

C0 – Active Mode: Code is executed

C1 – Auto Halt

C1E – Auto halt, low frequency, low voltage

C6 – Save core states before shutdown and PLL off

2.2.1.2 P-States[3]

During the execution of code, the operating system and CPU can optimize power consumption through different p-states (performance states). Depending on the requirements, a CPU is operated at different frequencies. P0 is the highest frequency (with the highest voltage). As of the Skylake architecture, the operating system can leave the control of the P-states to the CPU (Speed Shift Technology, Hardware P-states).

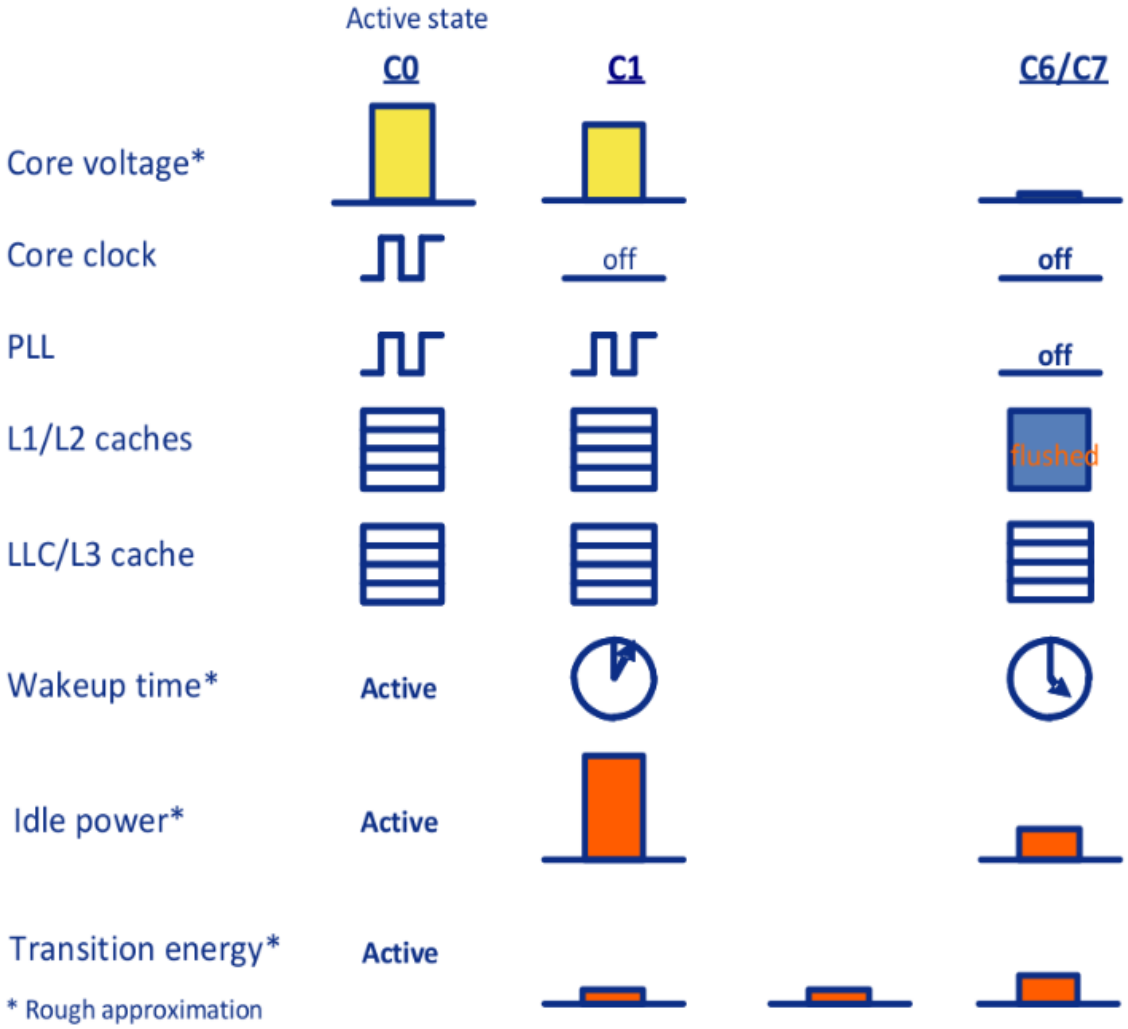


Figure 2: Overview of the different C-states and PC-states [3]

2.2.2 Simultaneous Multithreading (SMT)

SMT is a new technology which is used to improve efficiency of modern CPU with multithreading. It permits multiple independent threads to be executed different tasks for better using the resources by modern hardware architectures. This provides thread level multitasking. If we enable SMT we will be having 2 threads per core and when it is disabled, we will be having single thread per core. Enabling SMT doesn't promise always to increase in performance because in the case of the shared Resources are bottleneck, we can see a decrease in the performance.

2.3 Performance, Latencies , Power Metrics

Following a table (Table 1) with the metrics and the units used in the experiments

Metric	**Units**
Power Consumption	W (Watt)
Latency (AVG.,Tail,..)	us (Microsecond)
Instructions	Instruction(user +kernel)
Branch Mispredict,L1 icache, L1 dcache, iTLB, dTLB ,L2 ,L3 Misses	MPKI (Misses Per Kilo Instructions (Metric/Instructions)*1000)

Table 1: Metrics Table

The Instructions Branch Mispredict L1 icache ,dcache L2 ,L3 dTLB and iTLB misses will be collected from perf stat which gathers performance metric statistics.

The power Consumption Residencies of different C-states will be gathered from kernel CPU idle file.

Power consumption is total energy divided by execution time.

The average latency is the total request latencies divided by total request numbers)

The tail latency is the 99^{th} tail latency.

2.4 Problem Definition.

The main problem of this thesis is to know how different hardware configuration will have effect on modern Server applications both in performance and power. Specifically, How C-states configurations and SMT configuration effect on a server's performance and power consumption while running Memcached having a function different payload. The interaction between C-state and SMT Is a new idea that it is not researched previously. Is having SMT on with more threads will give us a better performance.

Chapter 3

Memcached[4]

3.1 Introduction

It is important to make a brief introduction on Microservices Architecture which is an architectural pattern that arranges an application as a collection of loosely coupled, fine-grained services, communicating through lightweight protocols. One of its objectives is for teams to create and offer their services independently of one another. One of the most Famous memory-caching systems used by Microservices is Memcached. To give Memcached a definition 'Memcached is a general-purpose distributed memory-caching system'. It is often used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source must be read. Memcached is free and open-source software, licensed under the Revised BSD license. Memcached runs on Unix-like operating systems (Linux and macOS) and on Microsoft Windows. It depends on the libevent library.

Memcached was first developed by Brad Fitzpatrick for his website LiveJournal, on May 22, 2003. It was originally written in Perl, then later rewritten in C by Anatoly Vorobey, then employed by LiveJournal. Memcached is now used by many other systems, including YouTube, Reddit, Facebook, Pinterest, Twitter, Wikipedia, and Method Studios.

3.2 Memcached Software architecture and data structure

The architecture of the system is client-server. Clients fill out and query a key-value associative array that the servers keep track of on their behalf. Values can only be up to one megabyte in size and keys can be up to 250 bytes long. Clients use client-side libraries to contact the servers which, by default, expose their service at port 11211.Both TCP and UDP are supported. All servers are known to every client, but they are not in contact with one

another. In order a client to set or read a value for a specific first, it computes the hash of that key. Then the server computes a second hash in order to determine where to store the value. Memcached uses strings and integers in its data structure. Hence, everything you save can either be one or the other. With integers, the only data manipulation you can do is adding or subtracting them. If you need to save arrays or objects, you will have to serialize them first and then save them. Memcached uses various hashing functions that are available in the library libmemcached. The pseudocode for the function to hash a key is as following:

uint32_t memcached_generate_hash_value(
***const* char* *key*, size_t *key_length*,**
memcached_hash_t*hash_algorithm*)

while: memcached_hash_thash_algorithm is an enum from the list below: Table 2

enum memcached_hash_t
enumerator MEMCACHED_HASH_DEFAULT
enumerator MEMCACHED_HASH_MD5
enumerator MEMCACHED_HASH_CRC
enumerator MEMCACHED_HASH_FNV1_64
enumerator MEMCACHED_HASH_FNV1A_64
enumerator MEMCACHED_HASH_FNV1_32
enumerator MEMCACHED_HASH_FNV1A_32
enumerator MEMCACHED_HASH_HSIEH
enumerator MEMCACHED_HASH_MURMUR
enumerator MEMCACHED_HASH_JENKINS
enumerator MEMCACHED_HASH_MURMUR3
enumerator MEMCACHED_HASH_CUSTOM

Table 2: Memcached Hashing algorithm enum List

The Hash used in my experiments is JENIKS HASH which is calculated by the following Hashing algorithm:

```
uint32_t jenkins(const uint8_t* key, size_t length) {
  size_t i = 0;
  uint32_t hash = 0;
  while (i != length) {
    hash += key[i++];
    hash += hash << 10;
```

```
    hash ^= hash >> 6;
  }
  hash += hash << 3;
  hash ^= hash >> 11;
  hash += hash << 15;
  return hash;
}
```

3.3 Functionality of Memcached

Memcached have only some basic functions GET, SET to Retrieve data we use get to update or write a new key value we use SET Request.The examples show how we get a key when we don't have a Memcached and the same when we have.

Note that all functions are pseudocode

Get Method: In this function it defers in Memcached we should first check Memcached and when it is not available in Memcached we query from the database.and then we add the key to memcached.

Regular Query without Memcached

```
function get_foo(int key)
     data = db_select
("SELECT * FROM keys WHERE key = ?",
key)
     return data
```

The same Code with Memcached

```
function get_foo(int key)
     /* first try the cache */
     data = memcached_fetch("keys:"
+ key)
     if not data
          /* not found : request
database */
          data = db_select("SELECT *
FROM keys WHERE key = ?", key)
          /* then store in cache
until next get */
          memcached_add("keys:"      +
key, data)
     end

     return data
```

Set Method:In this function both are similar with small difference in memcached in which we should add the key to memcached after updating it.

Regular Query without Memcached	The same Code with Memcached
function update_foo(int userid, string dbUpdateString) **/* first update database */** **result = db_execute(dbUpdateString)**	function update_foo(int userid, string dbUpdateString) /* first update database */ result = db_execute(dbUpdateString) if result /* database update successful : fetch data to be stored in cache */ data = db_select("SELECT * FROM users WHERE userid = ?", userid) /* then store in cache until next get */ memcached_set("userrow:" + userid, data)

Example:

In the figure 3 We can see the Workflow of Memcached Get Method. First, a client sends a Request to Memcached if the key is available (HIT) Memcached returns the value to the client when the key value is not available in Memcached (MISS) Memcached queries the key from the DB and then returns the value to the client.

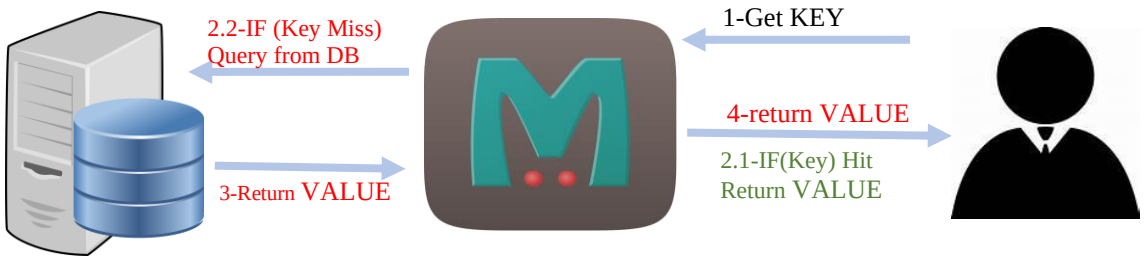


Figure 3: Memcached Functionality

3.4 Memcached Parameters

To run Memcached We have to Download it from the Repository

github.com/memcached/memcached

There are various parameters to configure Memcached, but the most important ones are the following Table 3:

Parameter	Values	Default	Description
-u User	<username>	-	To specify the user for executing
-m memory(in MB)	2-no limit	64MB	Memory limit
-p port			TCP port used
-U port			UDP port used
-c connections		1024	Maximum number of simultaneous connections to the Memcached service
-t threads		4	The number of threads to use when processing incoming requests
-d			Run Memcached as a demon

Table 3 : Memcached Parameters

Chapter 4

Workload Generator [5]

4.1 Introduction

The workload generator we are using is mcperf which is a Memcached load generator designed for high request rates, good tail-latency measurements, and realistic request stream generation.

mcperf reports the latency (average, minimum, and various percentiles) for get and set commands, as well as achieved QPS and network goodput. A separate thread is also keeping track of client CPU usage on the master. A warning will be issued if the master client CPU usage goes above 95%. In that case, it is recommended to add more machines as agents.To achieve high request rate, you must configure mcperf to use multiple threads, multiple connections, connection pipelining, or remote agents.

4.2 Parameters

In Table 4 We have various of the parameters used to run the workload generator and their description and default values

Parameter	Description
-s, --server=STRING	specify multiple servers.
-q, --qps=INT	Target aggregate QPS. 0 = peak QPS. (default=`0')
-t, --time=INT	Maximum time to run (seconds). (default=`5')
-K, --keysize=STRING	Length of Memcached keys (distribution).
-V, --valuesize=STRING	Length of Memcached values (distribution).

-r, --records=INT	Number of Memcached records to use. If multiple Memcached servers are given, this number is divided by the number of servers. (default=`10000')
-u, --update=FLOAT	Ratio of set:get commands. (default=`0.0')
-T, --threads=INT	Number of threads to spawn. (default=`1')
-c, --connections=INT	Connections to establish per server
-i, --iadist=STRING	Inter-arrival distribution (distribution). Note: The distribution will automatically be adjusted to match the QPS given by --qps. (default=`exponential')
--noload	Skip database loading.
--loadonly	Load database and then exit.
-A, --agentmode	Run client in agent mode.
-a, --agent=host	this client. (default=`1')
-C,--measure_connections=INT	Master client connections per server, overrides --connections.
-Q, --measure_qps=INT	Explicitly set master client QPS, spread across threads and connections.

Table 4: Mcperf Parameters

Some options take a 'distribution' as an argument.

To recreate the Facebook "ETC" request stream the following hard-coded distributions are also provided:[9]

fb_value = a hard-coded discrete and GPareto PDF of value sizes

fb_key = "gev:30.7984,8.20449,0.078688", key-size distribution

fb_ia = "pareto:0.0,16.0292,0.154971", inter-arrival time dist.

4.2 Example

For example if we want to generate a workload by 1 master and 4 agent clients ,QPS 10k using 40 threads and each thread 40 connections and we want to specify the requests to be only get Requests and the values size and key sizes and the interarrival time using the hard coded fb parameters and Having a Target Total Queries to send 2.4M which means we have to run it for 240 second,also we want to set the total memcached records to use to 1M records.

We should assign the following parameters with the following values(Table 5):

Parameter	Value
-s	Name of the node that the memcached server is running
-q	10000
-t	240
-r	1000000
-c	40
--noload	(the parameter as its own without a value)
-T	40
-a	Agent1
-a	Agent2
-a	Agent3
-a	Agent4
--keysize	fb_key
--valuesize	fb_value
--iadist	fb_ia

Table 5: Example workload generator parameters

Chapter 5

Memcached Extension to measure Server-Side Latency [7]

5.1 Memcached Server-Side Average and Tail Latency Problem

To calculate Memcached Server Side Latency, in previous works the method used was taking the sum of two results that are calculated by rusage_usr and rusage_sys commands and divide them by total request numbers. But these two commands didn't pass the validations made on them. The validation was to calculate according to instruction count Cycles per instructions the execution time of a specific program and then check whether these two commands will be able to give us the correct results. On Figure 4 we can see that when c-state is enabled the Time(us) calculated by these commands has a big difference that the calculated time which has to be equal to Calculated Exec Time.

Figure 4: Validating Rusage_usr Command

This was a problem because we didn't have a valid capture to server-side latency. The workaround was to have our own metrics in Memcached code, in order to measure execution time for each single request and in the end print in the statistics the results.

5.2 The Solution

To Find where to add the metrics, it was important to look at the code and see the workflow. Each Request of Memcached passes many stages Figure 6 The conn_new_cmd initializes a new connection and the conn waiting is a waiting stage that's why we don't include them in our measure the rest of the stages are included in the measurements. (In figure 6 the stages included in the yellow box)

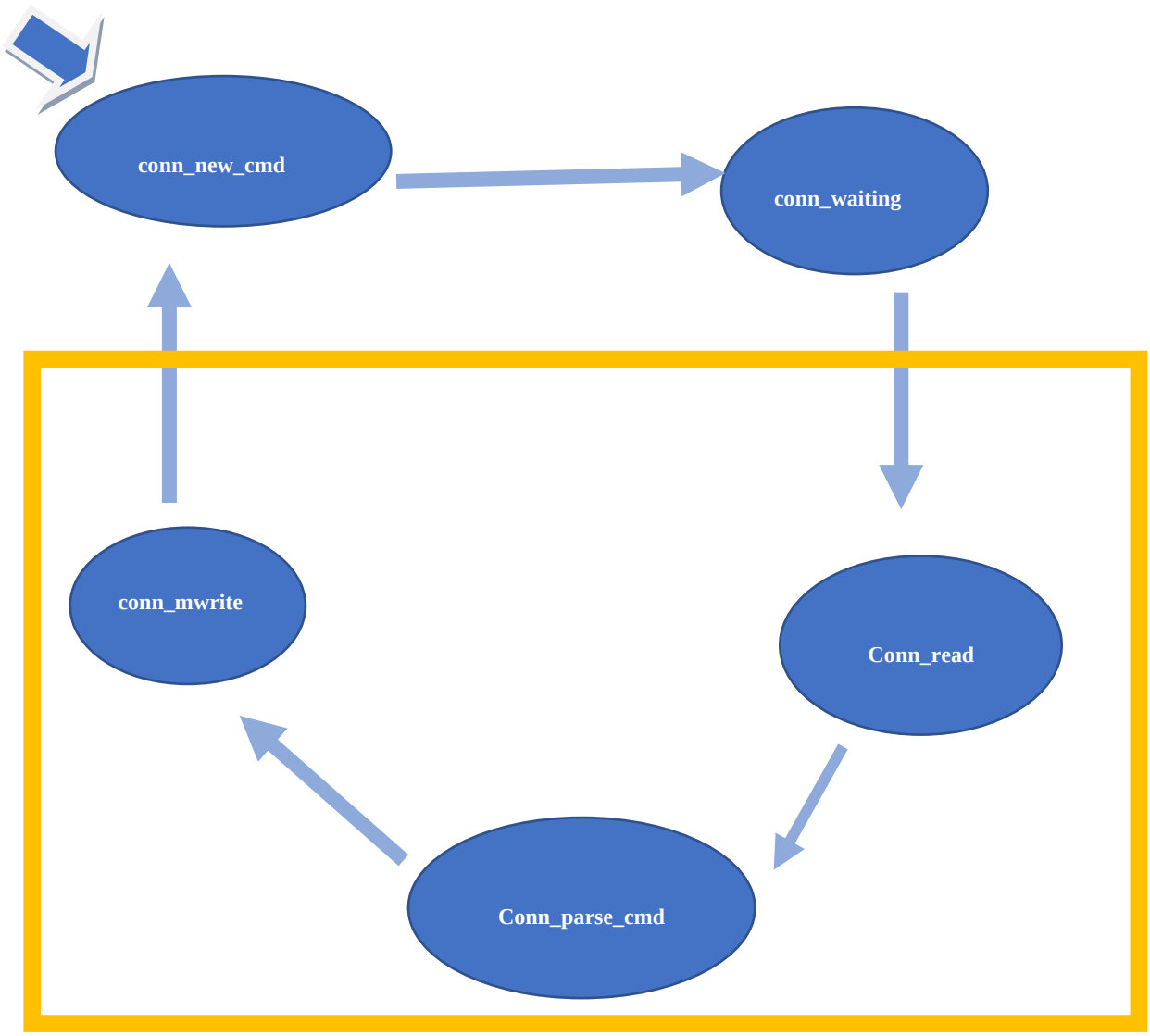


Figure 6: Memcached Request Stages

In order to calculate average Server Side Latency Basically we Sum all the Request Execution Time and divide them by the total Requests served.

In Order to Calculate the Server-Side Tail 99th Latency we used the following algorithm:

1. Array [10000] **Each cell index represents the latency in us**
2. When I calculate a single request execution time(t0) which will be in sec

   Will increase the counter in the corresponding Array index's cell

   Array [t0 * 10^6] ++ ; totalRequests++;

3. When finishes we have totalRequests=K

And Array has values let's assume are:

i	0	1	…	100	101	…	10000
Array[i]	0	2	…	23	33	…	0

To have the 99th Tail latency of the K total requests is

e = K * 0.99

then we iterate on the array to find out the e'th element:

```
bucket=0;

for (i=0;i<10000;i++)
{bucket+=array[i]

if(bucket >= e){
return i;}

return 0;

}
```

here are the codes that is modified in memcached highlighted the metrics added in a function that all the requests execute it which is the drive_machine.

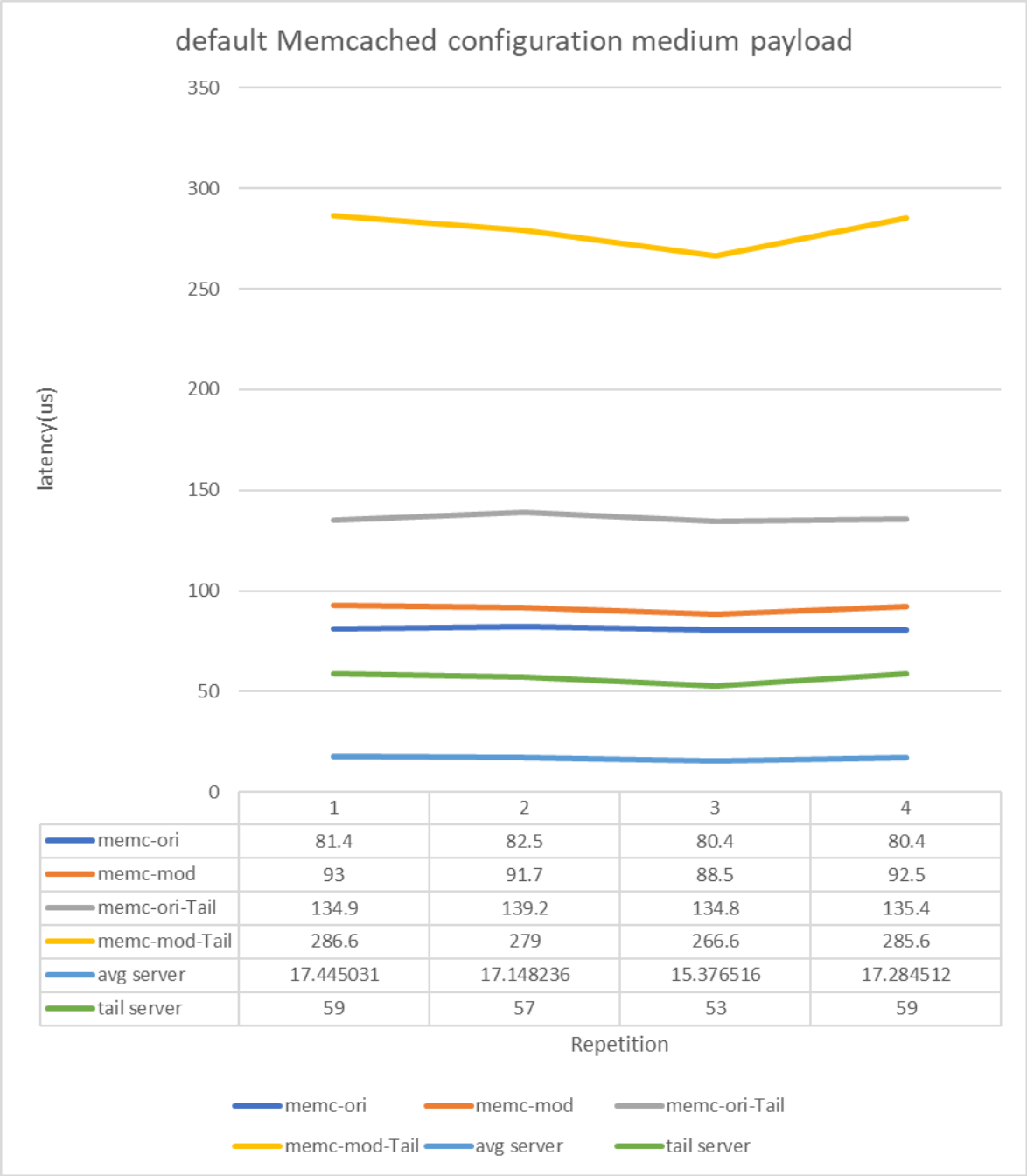
```
static void drive_machine(conn *c)
{
    …...
while (!stop)
    {if (settings.latencies > 0)
        {if (c->state == 3)
            {
         pthread_mutex_lock(&calc_lock);
         clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
         c->reqProcessStartTime = ts.tv_sec + (double)ts.tv_nsec /1000000000;
         pthread_mutex_unlock(&calc_lock);
            }
            else if (c->state == 9)
            {
            pthread_mutex_lock(&calc_lock);
           clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
           c->reqReqFinishTime = ts.tv_sec + (double)ts.tv_nsec / 1000000000;
           pthread_mutex_unlock(&calc_lock);

         if (c->reqReqFinishTime > 0.0  && c->reqProcessStartTime > 0.0)
                {
int indexProcess = (int)(((c->reqReqFinishTime) - (c->reqProcessStartTime)) *
1000000);

      if ( indexProcess > 0 && indexProcess<3000)
                    {
                   pthread_mutex_lock(&calc_lock);
 totalProcess += (double)((c->reqReqFinishTime) - (c->reqProcessStartTime));
                        ProcessCount++;
                        latencyProcessArray[indexProcess] += 1;
                        pthread_mutex_unlock(&calc_lock);
                    }
                }
            }
        }…….//then there are switch case code for the stages }
```

In the code above I am checking the stage that the request is according to that I start the measurements. As we can see to avoid race conditions locks are added which may have an overhead in the results.

5.3 Validation of Modified Code

To validate the modified version of Memcached in which we calculate the memcached Avg and Tail Latencies. It was important to compare the same exact workload with both Memcached versions and compare the results of the workload generator single client, QPS 1k read only requests, key value and interarrival Facebook parameters.

In Figure 7 we can see a graph which is the result of 4 executions for both modified and original Memcached results from workload generator we can observe that in both cases original and modified the client average(memc-ori and memc-mod) is very close but we can see an overhead on clients tail latency (memc-ori-Tail and memc-mod-Tail) that's because of the locks added on the code thus Memcached is a multithreaded program in order to use shared total and calculations it was necessary to add some extra locks to overcome the race condition of the shared variables. Another important validation is that the avg server is smaller than the tail server which are the average server latency and server 99^{th} latency calculated and also both of them are smaller than the corresponding client-side latency



	1	2	3	4
memc-ori	81.4	82.5	80.4	80.4
memc-mod	93	91.7	88.5	92.5
memc-ori-Tail	134.9	139.2	134.8	135.4
memc-mod-Tail	286.6	279	266.6	285.6
avg server	17.445031	17.148236	15.376516	17.284512
tail server	59	57	53	59

Figure 7: Validating Memcached Modified code

Chapter 6

Methodology of Experiments

6.1 Methodology

The experiments took place on six nodes all of them Intel-Skylake. A node for Memcached Server, a node for master client and 4 client agent nodes.

The Memcached Server machine had some Fix Configurations specified in the Table 4.

All the experiments included the following steps first configure the machine with the configuration of the specific experiment for example, C-state disabled and SMT on.

Fixed Configurations for server machine will be	
Machine	Intel Skylake (Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz) **Cores: 10 per socket #Sockets: 2** **thread per core: 2 (SMT ON)** **threads per core: 1 (SMT OFF)**
Intel P-States	off
Turbo mode	Off
Dvfs Governor	performance
Uncore frequency	2.0GHz
Idle governor	
Task set server to logical core worker threads	

Table 4: Fixed Memcached Server Machine configurations

The Following are the steps of each experiment:

1-We Start Memcached Server With the following parameters:

- Max memory: 16384MB
- Threads: SMT off 10, SMT on 20
- Simultaneous connection: 32768

2-We run the workload generator to load the Memcached server with values (1M unique keys) using Facebook parameters for key value and interarrival

3- We run warm up workload generator for 1 sec with 1M QPS.

4- Then we start our measurements on the server Machine, and we Start the workload generator with the specific configuration of the experiment

5- In the end, after finishes workload generator its execution, we terminate Memcached server.

Each Experiment is repeated 9 times and in the end after validating the Standard deviation is small an average of the results is calculated.

6.2 Experiment List

The different variable configurations were a combination of SMT and C-states Table 5:

	SMT	*C-states*
1	ON	ENABLED
2	ON	C1E ONLY (C6 DISABLED)
3	ON	C1 ONLY (C1E AND C6 DISABLED)
4	ON	DISABLED
5	OFF	ENABLED
6	OFF	C1E ONLY (C6 DISABLED)
7	OFF	C1 ONLY (C1E AND C6 DISABLED)
8	OFF	DISABLED

Table 5: Combination of the experiments' configurations

The workload generator we used 4 agent clients, thread number 40 and each thread 40 connection → Total Connection 160 And only ***read*** requests which means because we initialize the server in the beginning with the same key value distributions all the requests will be hits .Having Target Total Requests to Send 2.4M Req. without having any drop queries the key size values and inter arrival time we are using the Facebook parameters.
we used 3 different QPS and execution time parameters:

1- QPS 10k for 240s
2- QPS 100k for 24s
3- QPS 200k for 12s

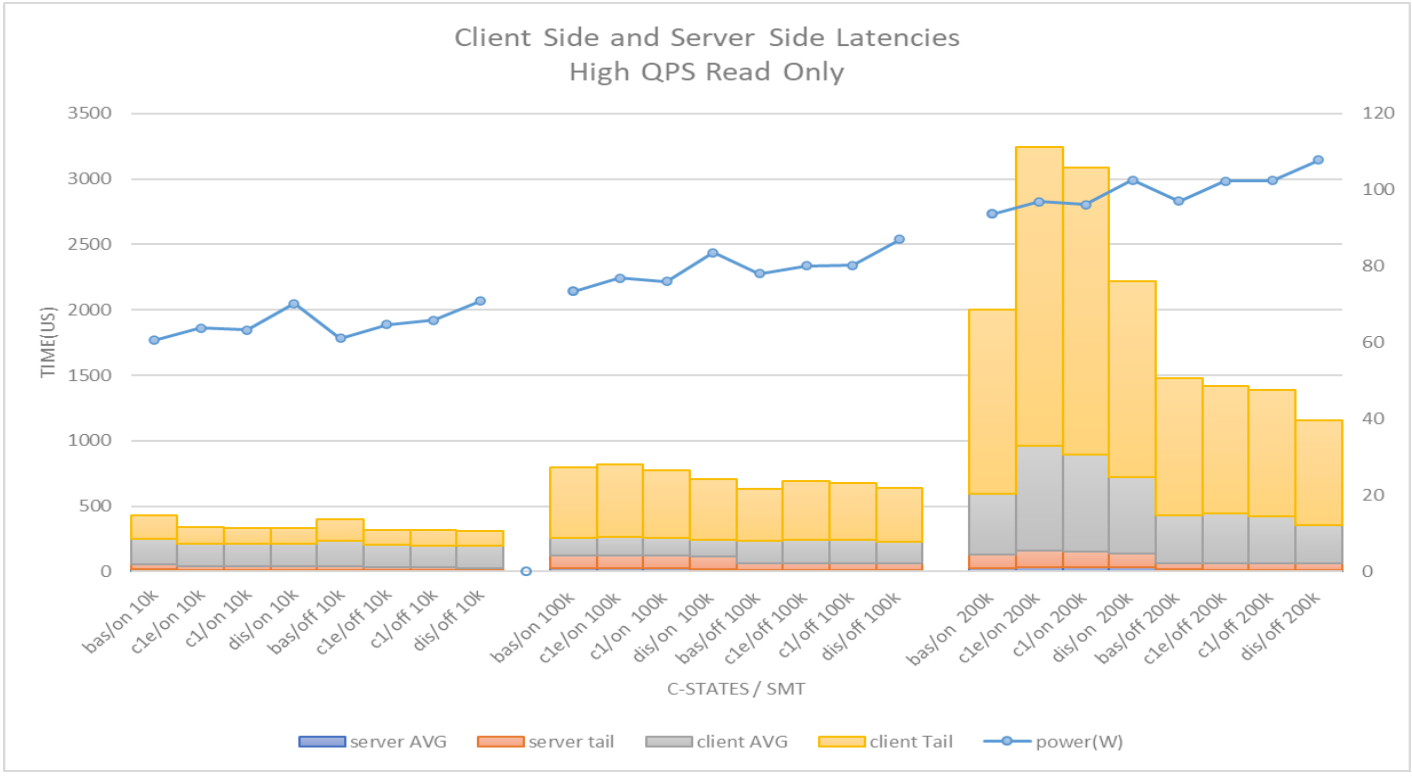
I have collected the power of CPU from the kernel packages and dram statistics taking 2 timestamps the begging and the end and the difference will give us each statistics at its own the packages and dram and dividing the sum of these statistics by execution time will give us the power.
The Server-side Average and Tail latency from the modified Memcached statistics.
The client-side latency and Tail from mcperf statistics.
The C-states Residency from CPU's kernel idle file.
The Branch Mispredict, L1 icache misses ,L1 dcache misses ,L2 misses .L3 misses dTLb misses iTLB misses from perf stat.

Chapter 7

Results

7.1 Results

After Executing the Experiments stated in the Methodology chapter, we collected the following data, after checking that the standard deviation is small an average of the results is calculated.Each result bar is an average of 9 results and each stack's measurements starts from 0.



Graph 1: Latencies collected from the experiments

7.1.1 Graphs Description

All the results respect a general format and shows a result (for example Latencies) for different QPS 10k ,100k and 200k starting from left to right 10k ,100k and 200k and configurations (Left to right) SMT on C-states enabled ,SMT on C-state c1e only (c6 disabled),SMT on C-state c1 only (c1e and c6 disabled) and then SMT on C-states Disabled and next to them the same configurations for c-states but in this case having SMT off.

The encodings of the x-axis are the following:

	Bas/on	C1e/on	C1/on	Dis/on	Bas/off	C1e/off	C1/off	Dis/off
C-state	Enable	C6 disabled	C1e and c6 disabled	Disabled	Enabled	C6 disabled	C1e and c6 disabled	disabled
SMT	On	On	On	On	Off	Off	Off	Off

Graph 1 :we can see the server-Side Average and Tail Latency also the client-side average and Tail Latency. For the 3 different QPS for all the configurations.

Also, on a secondary axis we can see the corresponding power consumption in Watts.

Graph 2: we can see a zoom in to results of 10k .

Graph 3:is a zoom in to the Server Tail and average Latencies for all the QPS.

Graph 4 : we can see the instruction Count results, which are the result from perf stat the sum of user and kernel instructions count.

Graph 5: we can see the C-states Stage Residencies which are collected from the kernel archives of each core cpuidle.

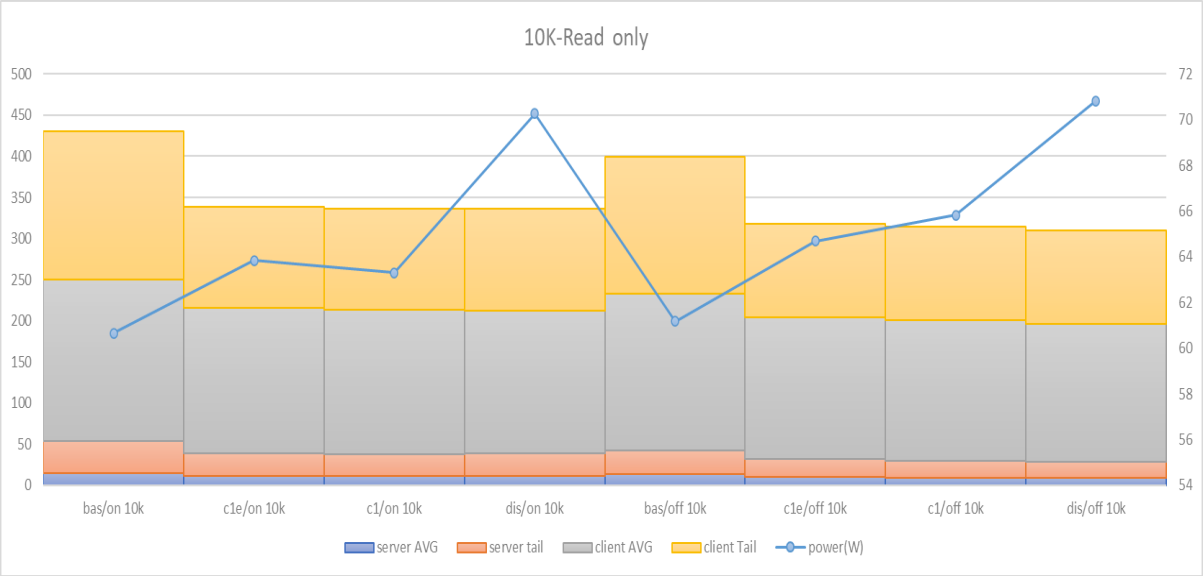
Graph 6: We can see branch mispredict L2 cache and L3 cache misses and results are Misses per kilo instructions(MPKI) which are collected from perf stat.

Graph 7:We can see the Results of iTLB and dTLB cache misses MPKI which are collected from perf stat.

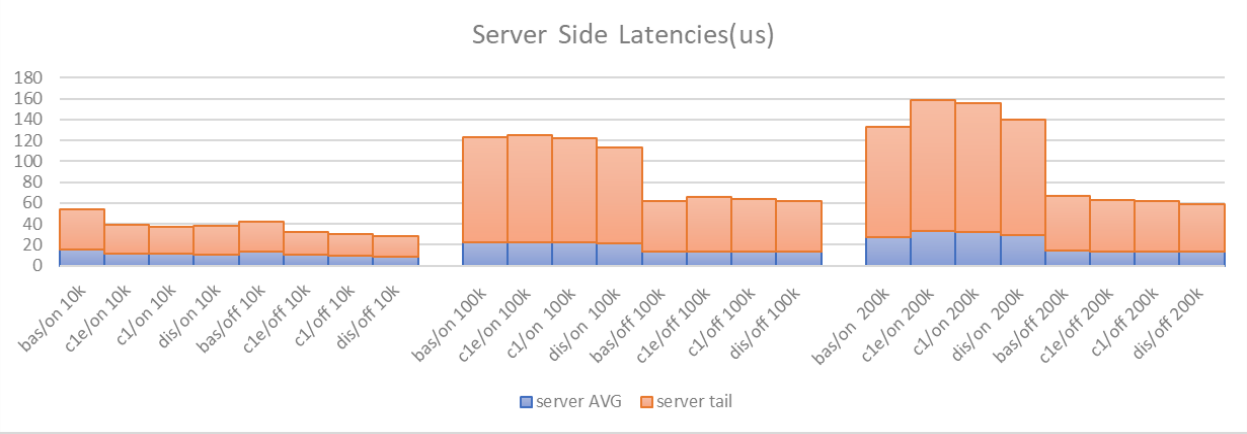
Graph 8: We can see the L1 icache and dcache misses which are collected from perf stat.

7.1.2 Validations:

We can see in Graph 1 that server-side average latency is smaller than server-side tail latency, also both are smaller than client-side average and Tail Latencies. Also each c-state configuration is validated, in other words we can see Graph 5 Which is the Residencies for each configuration we can see that configuration c1e doesn't enter c6 state at all, also c1 configuration doesn't enter c1e and c6.



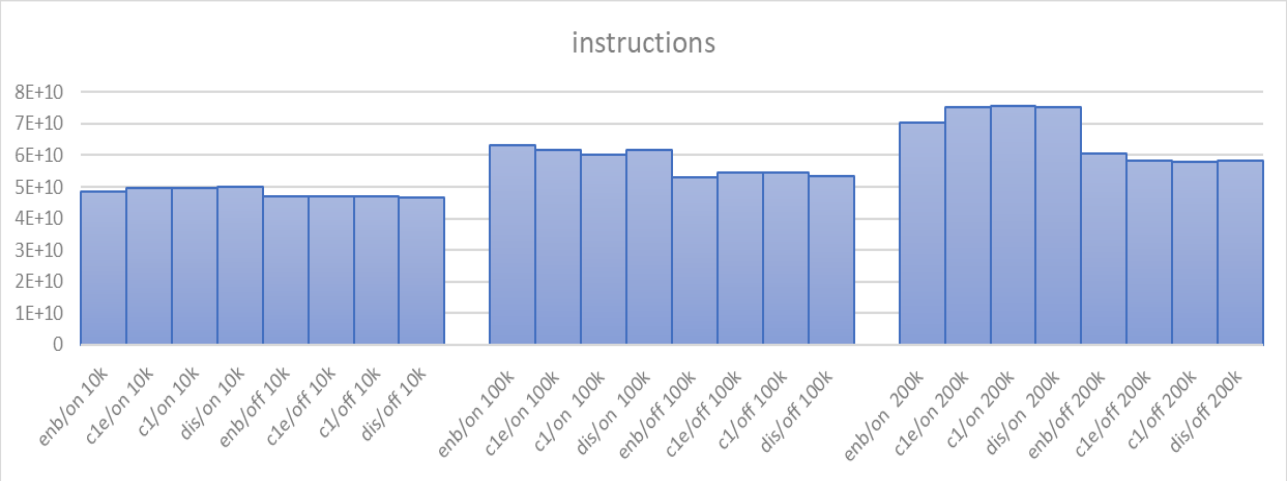
Graph 2: 10k Results zoom in



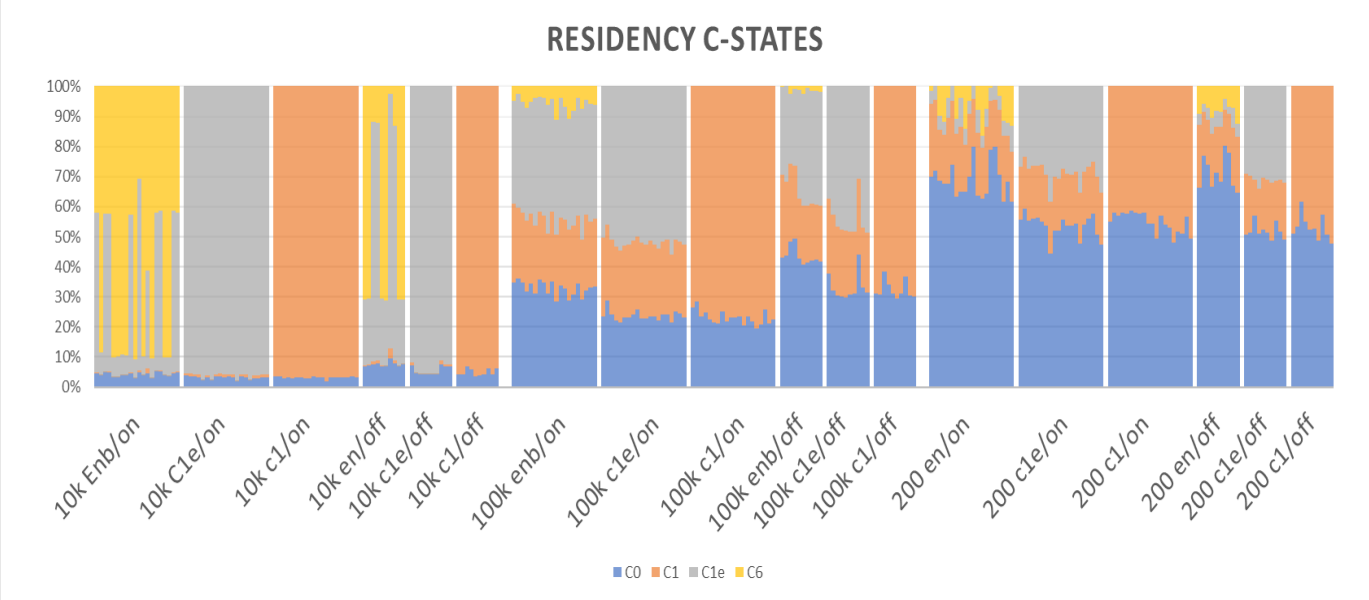
Graph 3: Server-Side Average and Tail Latency zoom in

Important Note:

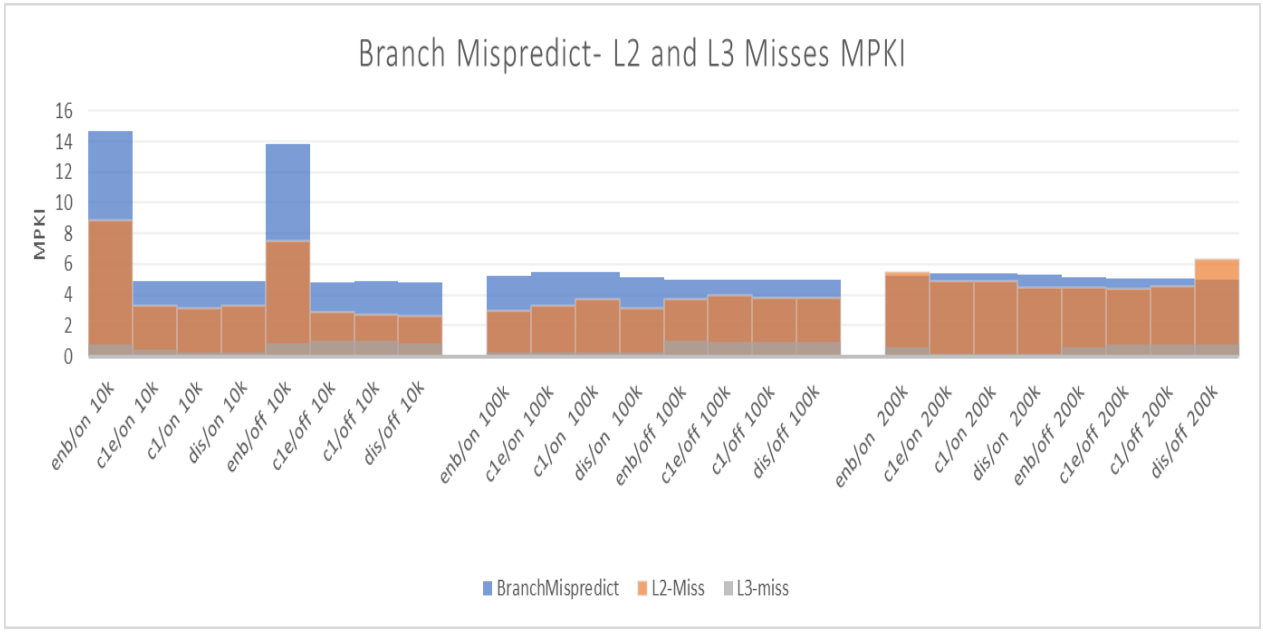
We can clearly see that from Graph 1 it is obvious that almost all the bars are close to each other except in 200k SMT/On which has a very strange result a very big tail latency because in this case it may have drop queries or maybe overutilisation of the server which due to those strange results, that's why I will exclude this result part from my discussion.



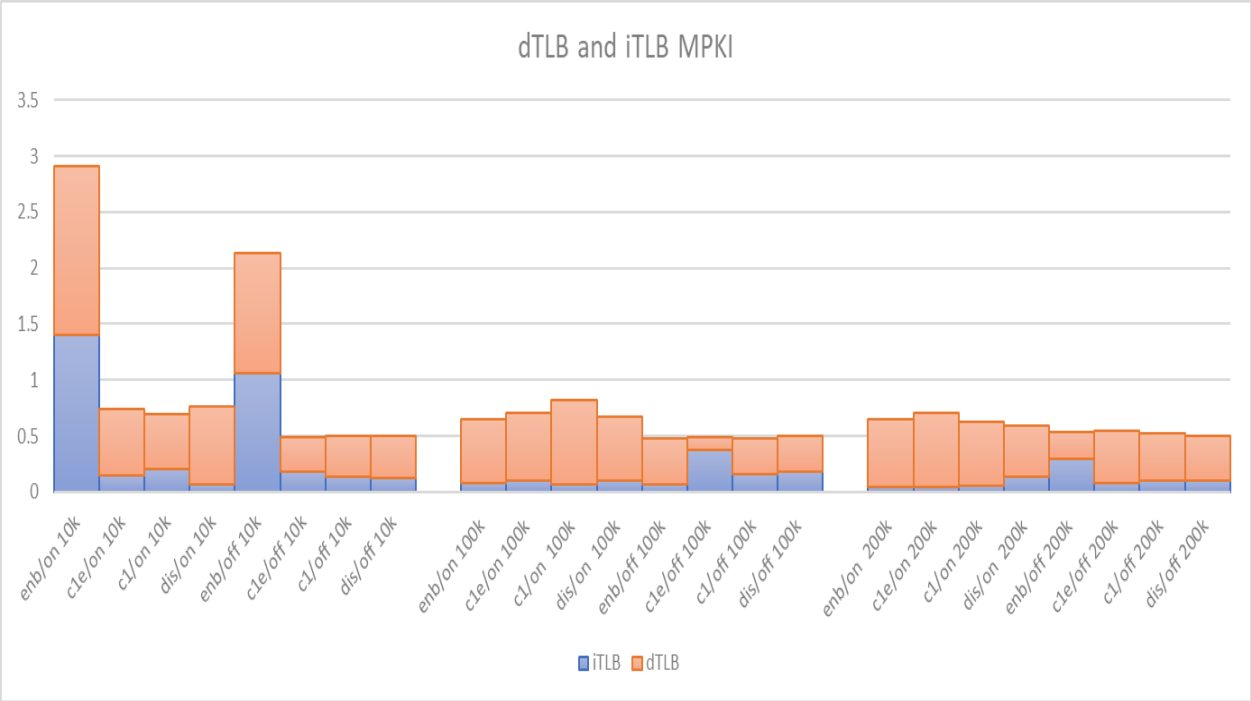
Graph 4: Instructions(user+kernel)



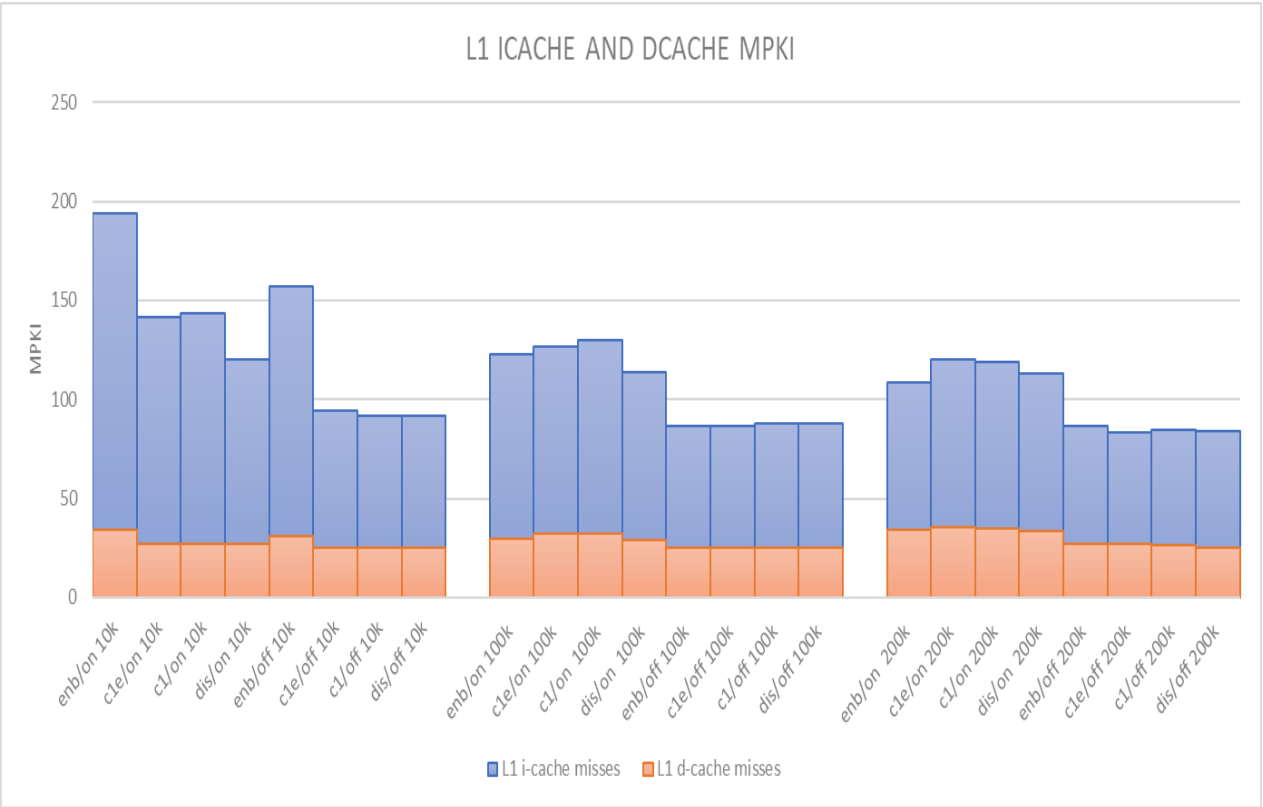
Graph 5: C-State Residencies



Graph 6-Branch Mispredict L2 and L3 Misses



Graph 7: iTLB and dTLB Misses



Graph 8: L1-icache and dcache Misses

7.1.3 Result Observations and Discussion

7.1.3.1 General Lines:

- ***The Total Latency (Client /Server Avg/Tail) are getting bigger while the QPS are getting larger. (Graph 1)***

For example, all the Latencies of 10k are smaller than 100k and both are smaller than the 200k because while having more QPS we have more Queuing according to Little's Law. Having to Execute 10k in 1 sec is different than having to execute 100k or 200k also we can't ignore the overhead added by the modified Memcached code which uses locks which means having more QPS increases the probability to have 2 requests arrive together.

- ***The Power Consumptions are getting bigger while the QPS are getting big (Graph 1)***

The power consumption in 10k is smaller than 100k and 200k.Because having more QPS means that we will have more work to do this leads to consume more power. For example if we compare the power consumption of 10k dis/Off is less than 100k dis/off and the previous both are smaller than 200k dis/off

7.1.3.2 SMT impact:

- ***Comparing the Latencies for the 10k and 100k sections both SMT on and off (Graph 1)***

It is clear that each C-state configuration when SMT is off the latency is smaller than its corresponding c-state configuration when SMT is on, It is because when we have SMT on we have 20 worker threads, while we have 10 worker threads on SMT off. This is because of the Overhead added from the modified Memcached which has additional locks on thread level which means having more threads leads to have a bigger probability to have 2 or maybe 3 threads to wait on queue on the same lock which means to have additional latency.

7.1.3.3 C-states impact

- ***The client and server Average and Tail Latencies in 10k section is worst when we have c-state enabled for both SMT on and off compared to the rest c-state configurations of the same SMT configuration (comparing the enb/on with the rest */on and the enb/off with the rest */off) (Graph 1 and 3)***

This can explain to us the Residencies because we can see that the residency of c0 is almost <10%-20% also the residency of c6 is almost 60-70% so this means it will add an overhead every time it enters c6 and wakes up back that's why we can see such a difference in the results. Also, The performance metrics Graph 6-7-8 we can see that in the specific configurations 10k enb/on and enb/off the misses are higher compared to the other configurations of the 10k experiment because when it enters c6 it flushes and clears the caches.

- ***The power Consumption In the C-state enabled is the smaller compared to the one in which c-state is disabled for all the QPS results. (Graph 1 and 3)***

That's because when we have the c-state enable we are in power save mode which leads to consume less power. Comparing to C-state disabled configuration which doesn't have any power saving mode on it and always in active mode.

- ***The Results of 200k and 100k we can realise that almost all the client and server latencies for different c-states are the same comparing to 10k results. (Graph 1 and 3)***

According to the Residencies for both QPS experiments we realise that server active mode is very high because there is not much idle time that's why it has similar results in all C-states configurations.

- ***The Results of C1e and C1 c-state is almost close in both performance and Power Consumption (Graph 1)***

  The difference is only in power consumption, but it doesn't have that much difference because in c1e it has more power saving than c1.

In the end to give a winner name, First we have to decide which is more important the winner in power consumption (the configuration with less power consumption) or the winner in performance (the configuration with the smaller latency). As per the winner of Best Power for QPS <100k the configuration that wins is the SMT/on and C-state Enabled.

The winner of Best Performance for QPS <100k the configuration that wins is the SMT/off C-state Disabled.

If we want to choose between SMT on or off always SMT off gave us better performance than the corresponding SMT on but also it was worst in power consumption.

If Things were predictable and we could know that we usually have >100k we could choose the best performance and best power consumption configuration which is the SMT off and C-state enabled.

Chapter 8

Conclusion

8.1 Conclusion

To conclude this research covers only specific 2 variable configurations in a pool of configurations that we could compare which are SMT and different C-states but in the end, it is very important to choose the correct configuration according to the needs of the application. For example, if the aim is reducing power consumption and having a moderate performance, we could choose a configuration according to the results of our experiment. Or if the aim is having best performance regardless of power consumption, we could choose the one that achieves it in the end in my opinion, It is good to choose something in the middle for example moderate performance and medium power consumption like this we could achieve both goals. Because in the end the changes are not that huge difference us differences in performance but reducing the power could have a positive effect also in the cost and also it is better for the environment.

8.2 Lessons Learned

In the end of this thesis it is important to point out that I've learned a lot about this topic but one of the most important things I leaned and believe which will give me a very good motivations is when I faced the problem with validating the Memcached server side latency with some techniques used by previous researchers and finding out that this method is not correct and I need to find out a workaround in very small time it was challenging because I needed to think out of the box a new method in the same time to validate it also. This was the best lesson I learned is to not believe anything I read, first check it and then if it is not correct do not give up and try to find a quick workaround.

8.2 Future Work

As I pointed in the Conclusion this research only covers 2 of many configuration switches and from client side only the Read only Requests. For Future work it was also very interesting to know the impact of these application throughout these configurations having a client Read and Write requests. Here it is good to point that during my research I have done various kind of experiments than the ones presented which is the Read Only 10k 100k and 200k QPS. For example Read and Write 10k 100k 200k also both configurations with low payload 100 and 1k QPS. Unfortunately, because of the shortage of time I couldn't complete the analysis of that results also. Another important modifications could be to update the Memcached modified code in order to decrease the using of lock or at least check whether the locks are the reason of some of overhead by comparing the Memcached server using single thread and multiple thread.

Bibliography

1. *Modeling and Analyzing Latency in the Memcached system*, Wenxue Cheng , Fengyuan Ren , Wanchun Jiang , Tong Zhang Tsinghua National Laboratory for Information Science and Technology, Beijing, China Department of Computer Science and Technology, Tsinghua University, Beijing, China School of Information Science and Engineering, Central South University, Changsha, China March 27, 2017

https://nns.cs.tsinghua.edu.cn/personal/chengwx/public_html/paper/modeling-memcached-techreport.pdf

2. *Data Centre Hardware and Architecture*

https://www.techtarget.com/searchdatacenter/Server-hardware-guide-to-architecture-products-and-management

3. *C-states and P-states*

https://www.thomas-krenn.com/en/wiki/Processor_P-states_and_C-states

4. *Memcached*

https://memcached.org/
https://en.wikipedia.org/wiki/Memcached

5. *Mutilate or Memcached Workload generator*

http://www-cs-students.stanford.edu/~leverich/papers/2014.mutilate.eurosys.pdf
https://github.com/leverich/mutilate

6. *Reconciling High Server Utilization and Sub-millisecond Quality-of-Service Jacob Leverich Christos Kozyrakis Computer Science Department, Stanford University, April 13–16, 2014.*

7. *Memcached Source code Analysis*

https://holmeshe.me/understanding-memcached-source-code-II/

8. *Cloud Lab*

https://www.cloudlab.us/

9.*Memcache-perf Experiment Runner*

https://github.com/hvolos/mcperf

10 *Notes of EPL420 Computer Architecture, University of Cyprus,2021*
And EPL 499 Data Centre Computing, University of Cyprus,2022.

Appendix A -Some useful commands

```
   1- Install Memcached using the command
sudo apt update
sudo apt install memcached libmemcached-tools

   2- Check if it is installed properly
memcached --version

   3- Check if it is running
sudo systemctl status memcached.service

   4- Stop Memcached
sudo systemctl stop memcached.service

   5- Install perf stat
sudo apt-get install linux-tools-common linux-tools-generic linux-
tools-`uname -r`

   6- Install soc watch
wget https://registrationcenter-
download.intel.com/akdlm/irc_nas/18447/l_oneapi_vtune_p_2022.1.0.98_
offline.sh
sudo sh ./l_oneapi_vtune_p_2022.1.0.98_offline.sh

   7-  Disable p-state
   sudo nano /etc/default/grub
   GRUB_CMDLINE_LINUX_DEFAULT="intel_pstate=disable"
   sudo update-grub
   reboot

   check pstates
   cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_driver

   8- Disable C-state
    sudo nano /etc/default/grub
   GRUB_CMDLINE_LINUX_DEFAULT="intel_pstate=disable
intel_idle.max_cstate=0 idle=poll"
   sudo update-grub
   reboot

   9- Turn-off turbo_mode
  git clone
https://github.com/hvolos/mcperf/blob/cea0bdfab936192758b9cc505abf85
d0dc60442b/turbo-boost.sh
   sudo apt-get install msr-tools
   Chmod +x turbo_boost.sh
   ./turbo_boost.sh disable
```

10-Turn off SMT
Firt to check use command
cat /sys/devices/system/cpu/smt/active

to disable SMT
echo off | sudo tee /sys/devices/system/cpu/smt/control

11-Client setup
git clone https://github.com/shaygalon/memcache-perf.git
sudo apt update
sudo apt-get install -y libevent-dev libzmq3-dev
cd memcache-perf/
git checkout 0afbe9b
make

HELPFUL HINTS:

to run Memcached:

```
sudo perf stat -e {…} /usr/bin/memcached -m 16384 -t 10 -c 32768 -p 11211 -u memcache -l 10.10.1.1 -P /var/run/memcached/memcached.pid > result &
```

to run mcperf:

```
./memcache-perf/mcperf -s node1 --noload -T 40 -q 10000 -t 120 -r 1000000 --iadist=fb_ia --keysize=fb_key --valuesize=fb_value &
```

All the script for the experiments with the results could be found in the git repository(public)
https://github.com/nanor1994/mcperf

which is a forked and modified version of the original Repository:
https://github.com/hvolos/mcperf