Memcached vs Redis: Benchmarking In-memory Object Caches

Milan Pavlik

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

2016

Abstract

0.1 Abstract

Object caches are an integral part of a scalable architecture. They are heavily used in web services and allow a service to exploit temporal trends in the services usage as well as increase throughput while decreasing latency of a service.

In this study we consider two popular object caches, Memcached and Redis, and perform a head to tail comparison of their performance on a common feature set - *get*, *set*. Both object caches feature different architectural decisions, Memcached is multi threaded while Redis is single threaded.

Memcached has received a large amount of published research while Redis has not been studies as extensively. We utilize the research on Memcached and apply it, where possible, to Redis. In our analysis, we first focus on fully benchmarking Memcached in order to establish a performance baseline of the system and allow us to validate the results with the literature. Subsequently, we extend our analysis to Redis. Finally, a head to tail comparison of the caches is made. Throughout the study, we aim to answer the following questions:

1. How does performance of Redis compare to Memcached on a common feature set?
2. Can a simpler, single threaded, architecture of an application compete with a multi-threaded design?

Throughout the study, we utilize benchmarking of the object caches as means of data gathering and analysis. We focus on key metrics of object caches - throughput and 99th percentile latency. Additionally, in order for an object cache to be useful, it must deliver a specific quality of service. In our study, we impose a quality of service constraint of 99th percentile latency under 1 ms on both caches. We focus on a wide range of benchmarks in this study including performance out of the box, multiple threads and/or instances as well as analyze the impact of object size on overall performance.

In our analysis, we find that Memcached delivers higher overall performance on a common feature set. However, Redis performance is not dissimilar to Memcached. It achieves an overall 8% lower level throughput. As a result, we find that a simple application design of Redis can perform as well as more complex, multi-threaded, design.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Boris Grot, for his invaluable guidance, patience and experience provided during my dissertation.

I would like to thank my parents, sister and friends for their continuous support, encouragement and willingness to listen.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In recent years, we have seen a shift from supercomputer computing to commodity computing. As a result, horizontal scalability has increased, however, complexity increased too. With distributed, it is no longer sufficient to embed caching in an individual application as future requests for data may be directed to other machines in the deployment. And this is where object caches come in.

An object cache is a dedicated application responsible for memoization of expensive computation results. For example, a popular person on a social media site may receive a large number of requests for the number of connections. It would be wasteful to traverse the social graph each time this data is requested and compute the number of connections. Instead, we execute the computation once and store the result in our object cache. Subsequent requests retrieve the results from the cache and avoid incurring the cost of the computation.

Frequently, object caches will change state throughout time. As such, the the object cache attempts to achieve temporal locality and captures the currently most popular requests. This is essential to increasing throughput and decreasing latency in a distributed system.

The motivation behind this study on object caches is their irreplaceability in modern scalable system architectures, their relative design simplicity and high effectiveness.

In this study, we focus on two popular object caches widely used in the industry - Memcached and Redis.

Memcached is a multi threaded object cache introduced in 2003 and it has been studied extensively since then. Researches have targeted various aspects of the memcached system in order to increase performance of the system. For example, memory allocation techniques have been explored as well as direct access to the NIC [12]. Other approaches aimed at removal of locks in Memcached's critical section have been explored too [30].

Unlike Memcached, Redis - a single threaded object cache released in 2009, has not received the same level of attention in published research. Redis features a different architecture design with focus on simplicity through the absence of threads and therefore locks.

Both caches are widely used in the industry, Facebook likely has the largest production deployment of Memcached with more than 800 servers [22], however, other companies such as Twitter [5], Google [24] and Amazon [1] use Memcached. Redis, on the other hand is used by Twitter [9], GitHub [17], Pinterest [8] or StackOverflow [15] among others.

Both caches feature a different architectural design decisions as well as provide different APIs. Memcached is simple with the main operations being *get*, *set* while Redis features a much richer API and supports additional data structures such as lists and sets. In this study, we perform a head to tail comparison of the two caches on a common feature set - *set*, *get*.

Additionally, by comparing the two caches on a common feature set, we can extend our conclusions on the design decisions made in development of each object cache. Can a single threaded application perform better than a multi-threaded one?

1.2 Memory Object Caches

Firstly, the purpose of a memory object cache is to use the machine's available RAM for key-value storage. The implication of a *memory object cache* is that data is only stored in memory and should not be offloaded on the hard drive in order to not incur hard drive retrieval latency.

The most common usage of an object cache is as part of a web service deployment. The cache is utilized to memoize the result of an expensive computation avoid the cost of computation for subsequent requests. Frequently, large web service deployments will utilize object caching heavily in order to decrease cost and scale better.

Secondly, an *object* cache implies that the cache itself is not concerned with the type of data (binary, text) stored within. As a result, memory object caches are multi-purpose caches capable of storage of any data type within size restrictions imposed by the cache. This is an abstraction, however, web services often deal with multiple data formats as well as data sizes.

Finally, memory object caches can be deployed as single purpose servers or also co-located with another deployment. Consequently, general purpose object caches often provide multiple protocols for accessing the cache - socket communication or TCP over the network. Both caches in question - Memcached and Redis - support both deployment strategies. Our primary focus will be on networked protocols used to access the cache.

1.2.1 Desired qualities

Firstly, an object cache should support a simple interface providing the following operations - *get*, *set* and *delete* to retrieve, store and invalidate an entry respectively.

Secondly, a general purpose object cache should have the capability to store items of arbitrary format and size provided the size satisfies the upper bound size constraints imposed by the cache. Making no distinction between the type of data is a fundamental generalization of an object cache and allows a greater degree of interoperability.

Thirdly, a cache should support operation atomicity in order to prevent data corruption resulting from multiple simultaneous writes.

Furthermore, cache operations should be performed efficiently, ideally in constant time and the cache should be capable of enforcing a consistent eviction policy in the case of memory bounds are exceeded.

Finally, a general purpose object cache should be capable of handling a large number of requests per second while maintaining a fair and as low as possible quality of service for all connected clients.

1.2.2 Design and Implementations

The design and implementation of a general purpose cache system is heavily influenced by the desired qualities of a cache.

Firstly, high performance requirement and the need for storage of entries of varying size generally requires the cache system to implement custom memory management models. As a result, a mapping data structure with key hashing is used to efficiently locate entries in the cache.

Secondly, in the case of *Memcached*, multi-threaded approach is utilized in order to improve performance. Conversely to Memcached, *Redis* is implemented as a single threaded application and focuses primarily on a fast execution loop rather than parallel computation.

1.2.3 Performance metrics

Firstly, the primary metrics reflecting performance of an in memory object cache are *mean latency*, *99th percentile latency* and *throughput*. Both latency statistics are reflective of the quality of service the cache is delivering to it's clients. Throughput is indicative of the overall load the cache is capable of supporting, however, throughput is tightly related to latency and on it's own is not indicative of the real cache performance under quality constraints.

Secondly, being a high performance application with potentially network, understanding the proportion of CPU time spent inside the cache application compared to time

spent processing network requests and handling operating system calls becomes important. Having an insight into the CPU time breakdown allows us to better understand bottlenecks of the application.

Finally, the *hit* and *miss* rate of the cache can be used as a metric, particularly when evaluating a cache eviction policy, however, the hit and miss rate is tightly correlated with the type of application and the application context and therefore it is not a suitable metric for evaluating performance alone.

1.3 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." [7] 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].

1.3.1 Memcached API

Memcached provides a simple communication protocol. It implements the following core operations:

- `get key1 [key2..N]` - Retrieve one or more values for given keys,
- `set key value [flag] [expiration] [size]` - Insert *key* into the cache with a *value*. Overwrites current item.
- `delete key` - Delete a given key.

Memcached further implements additional useful operations such as `incr/decr` which increments or decrements a value and `append/prepend` which append or prepend a given key.

1.3.2 Implementation

Firstly, Memcached is implemented as a multi-threaded application. "Memcache instance started with n threads will spawn n + 1 threads of which the first n are worker threads and the last is a maintenance thread used for hash table expansion under high load factor." [6]

Secondly, in order to provide performance as well as portability, memcached is implemented on top of *libevent* [18]. "The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also support callbacks due to signals or regular timeouts." [18]

Thirdly, Memcached provides guarantees on the order of actions performed. Therefore, consecutive writes of the same key will result in the last incoming request being the retained by memcached. Consequently, all actions performed are internally atomic.

As a result, memcached employs a locking mechanism in order to be able to guarantee order of writes as well as execute concurrently. Internally, the process of handling a request is as follows:

1. Requests are received by the Network Interface Controller (NIC) and queued
2. *Libevent* receives the request and delivers it to the memcached application
3. A worker thread receives a request, parses it and determines the command required
4. The *key* in the request is used to calculate a hash value to access the memory location in *O(1)*
5. Cache lock is acquired *(entering critical section)*
6. Command is processed and LRU policy is enforced
7. Cache lock is released *(leaving critical section)*
8. Response is constructed and transmitted [30]

We can observe that steps *1-4* and *8* can be parallelized without the need for resource locking. However, the critical section in steps *5-7* is executed with the acquisition of a global lock. Therefore, at this stage execute is not being performed in parallel.

1.3.3 Memcached Configuration

Memcached provides a convenient command line configuration options to tweak the performance of memcached through various parameters, commonly used options include:

-d	runs application in daemon mode
-p port	binds application to a TCP port (18080 by default)
-U port	binds application to a UDP port (18080 by default)
-m memory	defines how much memory to allocate to memcached (default 64)
-c conns	maximum number of simultaneous connections (default 1024)
-t threads	Number of threads (default 4)
-R num	Number of requests per connection (default 20)
-B protocol	Protocol support. One of ascii—binary—auto (default auto)

1.3.4 Usage in Production

Memcached has gained significant popularity in the industry. Facebook uses "more than 800 servers supplying over 28 terabytes of memory" [22] to their users. From publicly available data, Facebook is considered to have the largest deployment of Memcached. However, other tech companies are betting on Memcached to provide high scalability and speed.

Twitter uses Memcached for caching tweets on a user timeline [5]. Twitter uses Twemcache, a Memcached fork, which "has been heavily modified to make to suitable for the large scale production environment at Twitter." [28]

Aside from direct usage in a software stack, both Google [24] and Amazon [1] offer Memcached as web service in their web platform.

Hard numbers are hard to come by but Memcached is used heavily across the industry and shows maturity of application as well as the level trust large companies have in Memcached.

1.4 Redis

"Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries." [19] Redis was first released in 2009 and as such is younger than Memcached. It is frequently favored by developers due to it's rich feature set and good performance.

1.4.1 Redis API

Redis includes a rich text based API with a large number of commands. For the purposes of clarity, we will list only a subset of the commands.

- `get key` - Retrieve a value for a given key,
- `mget key [key ...]` - Retrieve multiple keys simultaneously,
- `set key value [expiry] [NX|XX]` - Insert *key* into the cache with a *value* and an optional expiration period. If NX, the key set if it does not already exist. If XX, key is set if already exists.
- `append key value` - Append *value* to *key*. Effectively creates a list.
- `del key [key ...]` - Delete a given key or multiple keys.

In this paper, we are primary concerned with *get* and *set*.

1.4.2 Redis Implementation

Redis is a single threaded application built on top of *libevent*. The single threaded nature of Redis is a conscious design decision and aims to alleviate development complexity arising from parallel programming.

Being single threaded allows Redis to perform an extremely fast event loop. The steps performed by a Redis cache are as follows [26]:

1. Requests are received by the Network Interface Controller (NIC) and queued
2. *Libevent* receives the request and delivers it to Redis
3. Request is parsed
4. Hash of *key* is calculated
5. Value is retrieved by the hash of the key
6. Response is constructed and transmitted

Therefore, there are no locks required as all processing occurs sequentially.

1.4.3 Redis Configuration

Redis is configured through the use of a *redis.conf* [20] file. The following list summaries the configuration options used in this paper.

- *daemonize* run in daemon mode (default yes)
- *port* binds application to a TCP port (default 6379)
- *maxmemory* The maximum amount of memory allocated to Redis (default 100MB)
- *maxmemory-policy* Eviction policy when max memory is reached, one of *volatile-lru, allkeys-lru, volatile-random, allkeys-random, volatile-ttl, noeviction*. We use *allkeys-lru* throughout this paper.

1.4.4 Usage in Production

Redis does not lack in popularity in the industry. Twitter utilizes Redis for real time delivery [9]. GitHub uses Redis "as a persistent key/value store for the routing information and a variety of other data." [17]. Pinterest stores a followers graph inside Redis, shared by user IDs [8]. StackOverflow uses Redis for aggressive content caching [15].

From the above, it is apparent that Redis is popular within the industry with highly popular web services relying on it to deliver performance.

Chapter 2

Methodology

In order to effectively benchmark the performance of both types of caches in question, it is essential to be able to stress the cache server sufficiently to experience queuing delay and saturate the server. This study is concerned with the performance of the cache server rather than performance of the underlying network and therefore it is essential to utilize a sufficient number of clients in order to saturate the server while maintaining low congestion on the underlying network.

The benchmarking methodology is heavily influenced by similar studies and benchmarks in the literature. This allows for a comparison of observed results and allows for a better correlation with related research.

2.1 Quality of Service

Firstly, it is important the desired quality of service we are looking to benchmark for. Frequently, distributed systems are designed to work in parallel, each component responsible for a piece of computation which is then ultimately assembled into a larger piece of response before being shipped to the client. For example, an e-commerce store may choose to compute suggested products as well as brand new products separately only to assemble individual responses into an HTML page. Therefore, the slowest of all individual components will determine the overall time required to render a response.

Let us define the quality of service (QoS) target of this study. For our benchmarking purposes, a sufficient QoS will be the *99th percentile* tail latency of a system under 1 *millisecond*. This is a reasonable target as the mean latency will generally (based on latency distribution) be significantly smaller. Furthermore, it is a similar latency target used in related research [11].

2.2 Hardware

Performance benchmarks executed in this study will be run on 8 distinct machines with the following configuration: *6 core Intel(R) Xeon(R) CPU E5-2603 v3 @ 1.60GHz*, *8 GB RAM* and *1Gb/s Network Interface Controller* (NIC).

All the hosts are connected to a *Pica8 P-3297* switch with 48 1Gbps ports arranged in a star topology. A single host is used to run an object cache system while the remaining seven are used to generate workloads against the server.

2.3 Workload generation

Workload for the cache server is generated using Memtier Benchmark developed by Redis Labs [10]. Memtier has been chosen as the benchmark for this study due to it's high level of configurability as well as ability to benchmark both *Memcached* and *Redis*. Utilizing the same benchmark client for both caches allows for a decreased variability in results when a comparison is made.

In order to create a more realistic simulation of a given workload, 7 servers all running *memtier* simultaneously are used. A simple parallel ssh utility is used to start, stop and collect statistics from the load generating clients.

The workload generated by Memtier is driven by the configuration specified. The keys and values are drawn from a configured distribution dynamically at runtime. All comparable benchmarks presented in this thesis configure the same initial seed for comparable benchmarks in order to minimize stochastic behavior.

2.4 Benchmark

In the context of this thesis, a benchmark is a set of workloads executed against the cache host. Statistics are collected from the cache host as well as the clients in order to draw conclusions.

Firstly, a benchmark consists of a warm up stage. The cache is being loaded with initial data in order to prevent a large number of cache misses and skewed results.

Secondly, a configured workload is generated and issued against the cache host from multiple benchmarking hosts simultaneously.

Thirdly, the workload is repeated 2 more times in order to decrease the impact of stochastic events in the benchmark.

Finally, statistics are collected, individual benchmark runs are averaged and the results are processed.

2.4.1 Memtier

Memtier benchmark is "a command line utility developed by Redis Labs for load generation and benchmarking NoSQL key-value databases" [10]. It provides a high level of configurability allowing for example to specify patterns of *sets* and *gets* as well as generation of key-value pairs according to various distributions, including Gaussian and pseudo-random.

Memtier is a threaded application built on top of `libevent` [18], allowing the user to configure the number of threads as well as the number of connections per each thread which can be used to control the server load. Additionally, memtier collects benchmark statistics including latency distribution, throughput and mean latency. The statistics reported are used to draw conclusions on the performance under a given load.

Memtier execution model is based on the number of threads and connections configured. For each thread `t`, there are `c` connections created. The execution pattern within each thread is as follows:

1. Initiate `c` connections
2. For each connection
   (a) Make a request to the cache server
   (b) Provide a *libevent* callback to handle response outside of the main event loop

By offloading response handling to a callback inside libevent, memtier is able to process a large number of requests without blocking the main event loop until a response from the network request is returned while maintaining the ability to collect statistics effectively.

Connections created with the target server are only destroyed at the end of the benchmark. This is a realistic scenario as in a large distributed environment the cache clients will maintain open connections to the cache to reduce the overhead of establishing a connection.

Memtier provides a comprehensive set of configuration options to customize Memtier behavior and tailor the load. Table 2.1 outlines the relevant configuration options. The complete set of configuration options is available on RedisLabs [21].

2.4.2 Open-loop vs Closed-loop

A load tester can be constructed with different architecture in mind. The main two types of load testers are *open-loop* and *closed-loop*. Closed-loop load testers frequently construct and send a new request only when the previous request has received a response. On the other hand, open-loop principle aims to send requests in timed intervals regardless of the response from the previous requests.

Configuration option	Explanation	Default Value
-s	Server Address	localhost
-p	Port number	6379
-P	Protocol - redis, memcache_text, memcache_binary	redis
-c	Number of Clients per Thread	50
-t	Number of Threads	4
–data-size	The size of the object to send in bytes	32
–random-data	Data should be randomized	false
–key-minimum	The minimum value of keys to generate	0
–key-maximum	The maximum value of keys to generate	10 million

Table 2.1: Memtier Configuration Options

The consequence of a closed-loop load tester is potentially reduced queuing on the server side and therefore observed latency distribution may be lower than when server side queuing is observed.

Memtier falls in the category of closed loop testers when considering a single thread of memtier. However, memtier threads are independent of each other and therefore requests for another connection are made even if the previous request has not responded. Furthermore, by running memtier on multiple hosts simultaneously, the closed loop implications are alleviated and the server observes queuing delay in the network stack.

Chapter 3

Memcached

The purpose of this chapter is to benchmark and evaluate Memcached performance. Firstly, we will focus on Memcached performance "out of the box". Secondly, we will examine Memcached scalability in respect to threads which will provide us with an optimization baseline. It is worth noting that majority of Memcached user will end up using the default configuration, perhaps with increased threading level. Subsequently, we will explore the impact of assigning individual threads to CPU cores as well as the impact interrupt processing has on Memcached. Furthermore, we will explore a multi-instance setup as well as the impact object size has on Memcached performance. Finally, we will turn our attention to the distribution of cache keys.

Throughout this chapter, we will focus primarily on latency, 99th percentile latency and throughput. Where relevant, we will explore additional attributes. Unless otherwise stated, all performance tuning is done to meet a Quality of Service (QoS) constraint of 99th percentile latency under 1 millisecond.

3.1 Shiny Fresh Memcached

Firstly, let us focus on Memcached performance "out of the box", that is, Memcached with a default configuration. When we tear down the wrapping paper of a Memcached distribution, we are presented with two important configuration options - a) The port number memcached will listen on and b) the amount of memory we allocate to Memcached which determines the total capacity of the cache. For the purposes of this paper, we use port number 11120. The amount of memory we allocate to memcached is dependent on the total amount of memory available on the host machine as well as any additional workload on the host. In our case, we are the sole workload with a total of 8GB memory available to us. Throughout this paper, we choose to allocate 6 GB of memory to Memcached, leaving 2 GB for the operating system or remaining unused. Table 3.1 outlines the configuration options including relevant defaults. It is worth noting that in the default configuration Memcached runs with 4 threads.

Given the configuration outlined in Table 3.1, we can proceed and launch Memcached

Configuration Option	Explanation	Value
-d	Run in Daemon Mode	true
-p	Port number	11120
-t	Number of Threads	4 (default)
-m	Memory Allocated	6144 (6GB)

Table 3.1: Memcached Configuration Options.

on the server with the following command:

```
memcached -d -p 11120 -m 6144
```

Secondly, we configure the clients responsible for generating cache workload. In order to determine a saturation point of the serve cache, we increase the workload exerted by the clients linearly. Initially, we consider a workload provided by 3 threads and 1 connection per each workload generating server. Subsequently, the number of connections is increased linearly until a saturation point is found or QoS requirements are no longer satisfied. For the benchmark, and indeed for the rest of the paper unless otherwise stated, we consider an object size of 64 bytes. With object sizes of 64 bytes, we aim to generate a sufficiently large dataset in order to exceed the memory capacity provisioned for Memcached. In this case, we define the key space to be between 1 and 100 million, yielding a dataset 6.4GB large. Table 3.2 outlines the configuration options used.

Configuration Option	Explanation	Value
-s	Server	nsl200 (server hostname)
-p	Port number	11120
-c	Number of Connections	[1..10]
-t	Number of Threads	3
–key-minimum	Smallest key	1
–key-maximum	Largest key	100 000 000
–random-data	Generate Random Data	true
–data-size	The size of data in bytes	64

Table 3.2: Memtier Configuration Options

The memtier_benchmark (Memtier) can be launched with the following command:

```
memtier -s <server> -p 11120 -c <connections> -t 3
  --random-data
  --key-minimum=1
  --key-maximum=100000000
  --random-data
  --data-size=64
```

The application start commands are provided for clarity and will be omitted in subsequent benchmarks as they can be directly constructed from the configuration tables.

3.1.1 Latency, Throughput and Number of Connections

Firstly, we are interested in the relationship between throughput, latency and the number of connections. The relationship is shown in Figure 3.1. Latency, both mean and 99th percentile, are plotted on the left vertical axis, the number of operations per second is plotted on the right vertical axis and the number of connections used is on the horizontal axis.

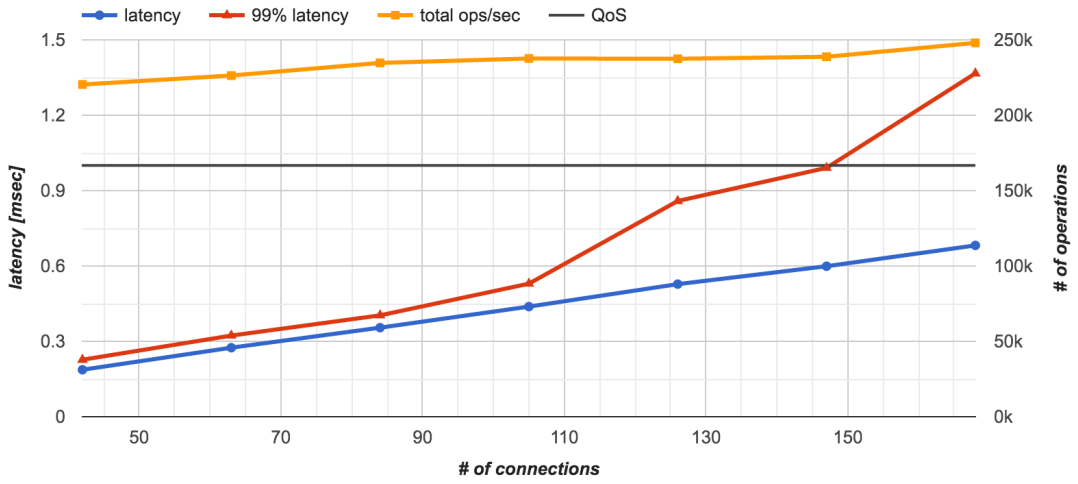

Figure 3.1: Latency & Throughput vs Number of Connections

As the number of connection increases, so does mean latency. There is a linear relationship between the mean latency and the number of connections. This is an expected result, as the load increases linearly, we expect the mean latency to increase linearly too. An increase in the number of connections by 21 results in increased mean latency by approximately 0.09ms.

The 99th percentile latency increases linearly with the number of connections until we reach 105 connections. A further increase in the number of connections results in a disproportionately greater increase in tail latency. The maximum number of connections satisfying the QoS occurs at 147 connections and tail latency at 0.99ms. A further increase in load results in a steeper increase in tail latency. We can observe as the load increases, the tail latency diverges from the mean latency and increases at a faster pace.

The number of operations per second increases linearly with the number of connections and reaches a peak of 238k requests per second at 147 connections (within QoS). The total throughput appears to be largely unaffected by the steep increase in tail latency in this benchmark.

3.1.2 CPU Utilization

Secondly, we consider the effect of the workload on the Memcached server in terms of CPU Utilization. The CPU utilization is monitored through the *mpstat* [14] utility

which reports the percentage of CPU utilization broken down into multiple categories. The following table [14] summarizes the responsibilities of each category.

%usr Show the percentage of CPU utilization that occurred while executing at the user level (application).

%sys Show the percentage of CPU utilization that occurred while executing at the system level (kernel). Note that this does not include time spent servicing hardware and software interrupts.

%iowait Show the percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request.

%irq Show the percentage of time spent by the CPU or CPUs to service hardware interrupts.

%soft Show the percentage of time spent by the CPU or CPUs to service software interrupts.

%idle Show the percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request.

For the context of this paper, *%usr* corresponds directly to the CPU utilization used by Memcached as it is the only application running on the server.

Furthermore, *%soft* represents the software interrupt issued by *libevent* when a new file descriptor is available for processing, that is, a new request is available to be processed or a response is ready to be handed over to the network stack.

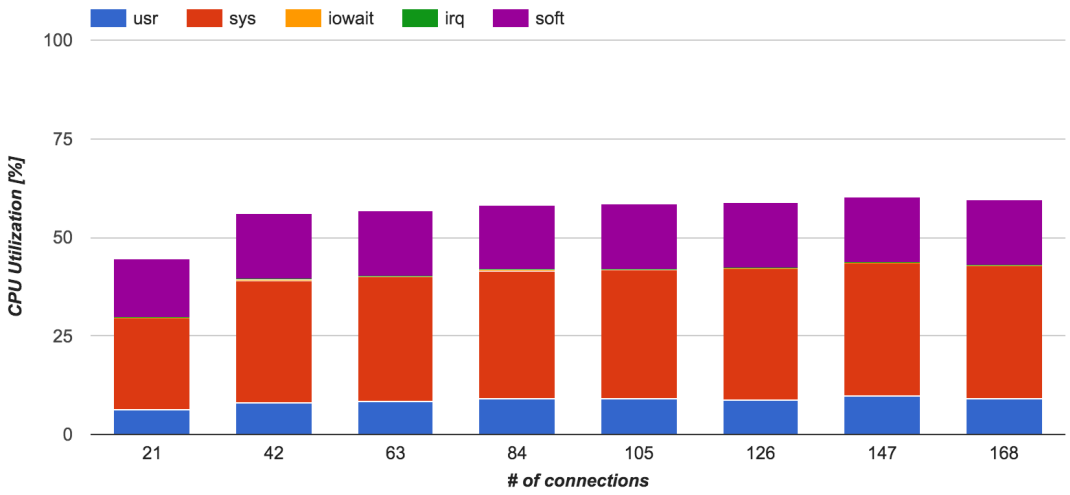

Figure 3.2: CPU Utilization for Out of the Box Configuration of Memcached

Figure 3.2 outlines the CPU Utilization broken down into mpstat categories. Note that unallocated percentage constitutes the *idle* percentage of the CPUs.

Memcached utilization (*usr*) increases slightly as the number of connections increases, however, overall remains stable and accounts for at most 9% of the total utilization. The kernel (*sys*) accounts for 33%, a significant portion of the CPU utilization relative

to other categories. The CPU utilization dedicated to processing software interrupts (*soft*) accounts for 16% of the total CPU utilization. This makes the second most significant category in the benchmark. Disk IO accounts for 0% of total utilization as all data is stored in memory and we do not need to access the disk. Hardware interrupts (*irq*) account for 0.01% of total utilization.

Overall, CPU utilization starts at 48% and increases to 60% as the number of connections increases. We can observe that at this CPU utilization we have not been able to fully utilize the server resources. Additionally, the kernel and software interrupts account for majority of the total CPU utilization. This is indicative of a larger pattern of Memcached execution - Memcached performance is dominated by the kernel and network stack. This is consistent with findings in MICA [12].

3.1.3 Evaluation

Given the trends presented in Figures 3.1 and 3.2, we can conclude the "out of box" configuration of Memcached does not deliver optimal performance. The number of operations per second could be increased by increasing CPU utilization which in turn should result in improved tail latency. Additionally, we have been able to observe that the kernel and software interrupt performance dominates the CPU utilization. A simple approach to increasing CPU utilization is to increase the number of threads we provision for Memcached.

3.2 Thread Scalability

In this section, we focus on increasing CPU utilization through the use of multiple threads. We have shown that the default configuration results in underutilization of the server resources and argued that increased CPU utilization should result in improved performance. Memcached, as a high performance object cache, is designed to be executed on a multi-core architecture. Scalability is primarily implemented through the use of multi threading. It is worth noting that multi-threading also results in increased application complexity. Individual threads requiring access to shared data are required to obtain a lock before they can proceed with data manipulation. We design a benchmark which focuses on thread scalability by linearly increasing the number of threads Memcached is provisioned.

Empirically, we expect the best performance to be achieved when there are as many Memcached threads as there are CPU cores. The cache server is equipped with 6 CPU cores and therefore we would expect 6 threads to maximize performance. This is also suggested by Leverich and Kozyrakis [11]. Utilizing less threads should result in underutilization of the CPU leading to sub-optimal throughput. More than 6 threads should conversely result in increased context switching overhead and therefore should lead to increased latency.

Utilizing results about the number of connections from previous section, we configure a constant level of workload with 3 Memtier threads and 7 connections per each thread. We maintain the same key-object configuration as in previous benchmark. The configuration details of the Memtier benchmark are outlined in Table 3.3.

Configuration Option	Explanation	Value
-s	Server	nsl200 (server hostname)
-p	Port number	11120
-c	Number of Connections	7
-t	Number of Threads	3
–key-minimum	Smallest key	1
–key-maximum	Largest key	100 000 000
–random-data	Generate Random Data	true
–data-size	The size of data in bytes	64

Table 3.3: Memtier Configuration Options

Memcached, on the other hand, is configured to increase the number of threads in each consecutive benchmark. Table 3.5 outlines the configuration used.

Configuration Option	Explanation	Value
-d	Run in Daemon Mode	true
-p	Port number	11120
-t	Number of Threads	[1..10]
-m	Memory Allocated	6144 (6GB)

Table 3.4: Memcached Threads Configuration Options

3.2.1 Throughput & Latency

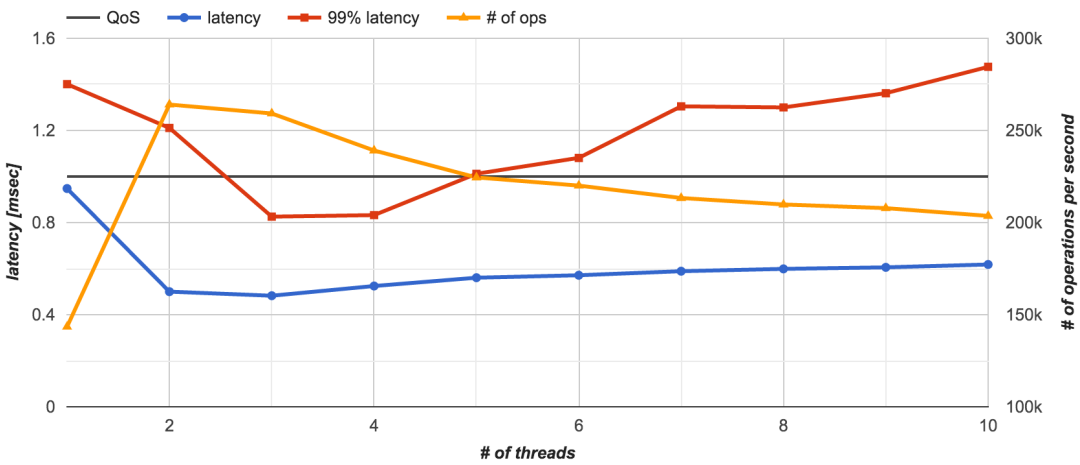

Figure 3.3: Memcached Threads: Latency & Throughput vs Number of Threads

Figure 3.3 plots the relationship between the number of threads used by Memcached on the horizontal axis, latency on the left vertical and the number of operations on the right vertical.

Mean latency, experiences a sharp decrease between 1 and 2 threads and increases steadily as the number of threads increases beyond 2 threads. The 99th percentile latency fails to meet the QoS constraints initially between 1 and 2 threads, dropping below the QoS constraint with 3, 4 and 5 threads. However, it climbs beyond the QoS requirements with 6 threads and continues to increase as the number of threads increases. The number of operations increases sharply between 1 and 2 threads, reaching a maximum of 270k requests per second while decreasing as the number of threads increases.

The performance obtained with increased number of threads does not match our expectations of best performance at 6 threads. However, we can still observe the expected downward shaped parabolic curve the 99th percentile latency plots. Let us examine the CPU utilization to gain a better insight into the problem.

3.2.2 CPU Utilization

Figure 3.4 provides the *mpstat* category breakdown of the CPU utilization of Memcached during the benchmark. Note that unattributed utilization accounts for idle time.

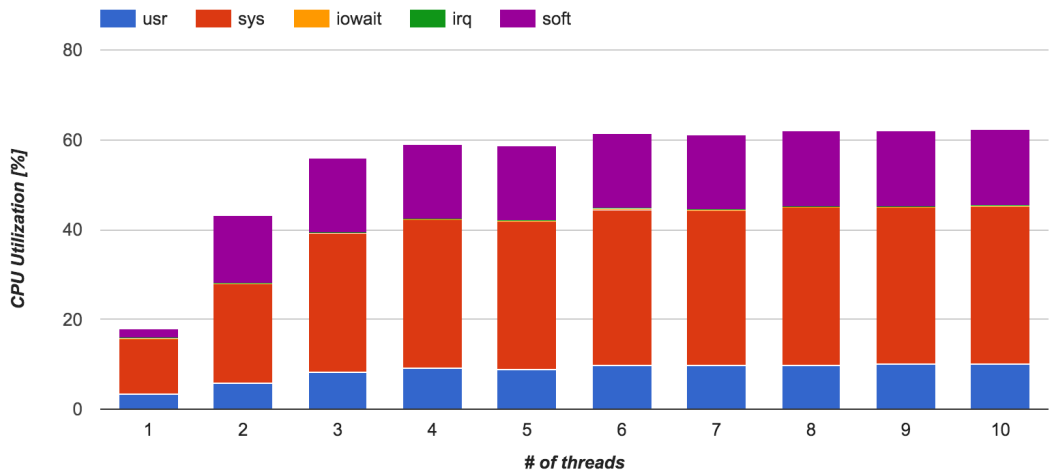

Figure 3.4: Memcached CPU Utilization with Multiple Threads

The CPU utilization of Memcached accounts for a small portion of the total utilization. Initially, with 1 thread Memcached only accounts for 3% of total utilization. As the number of threads increases, Memcached CPU usage increases too, however, only up to 4 threads at which point it remains constant. The kernel usage increases with the number of threads and stabilizes at 4 threads with 34%. An increase in the number of threads does not result in increase in the kernel usage. Software interrupt CPU usage increases from 2% to 15% between 1 and 2 threads and remains stable as the number of

threads increases. Similarly to previous benchmarks, disk IO and hardware interrupts take up insignificant portions of the CPU utilization.

Overall, we can observe that the total CPU utilization reaches at most 65%. The CPU usage overall does not reflect our expectation of increased CPU usage with more threads. In fact, there is no significant increased utilization with more than 4 threads.

Interestingly, the software interrupt usage only increases between 1 and 2 threads which is contrary to the expectation of a linear increase in interrupt processing with multiple threads. Let us investigate CPU utilization further by focusing on the breakdown of individual CPU usage.

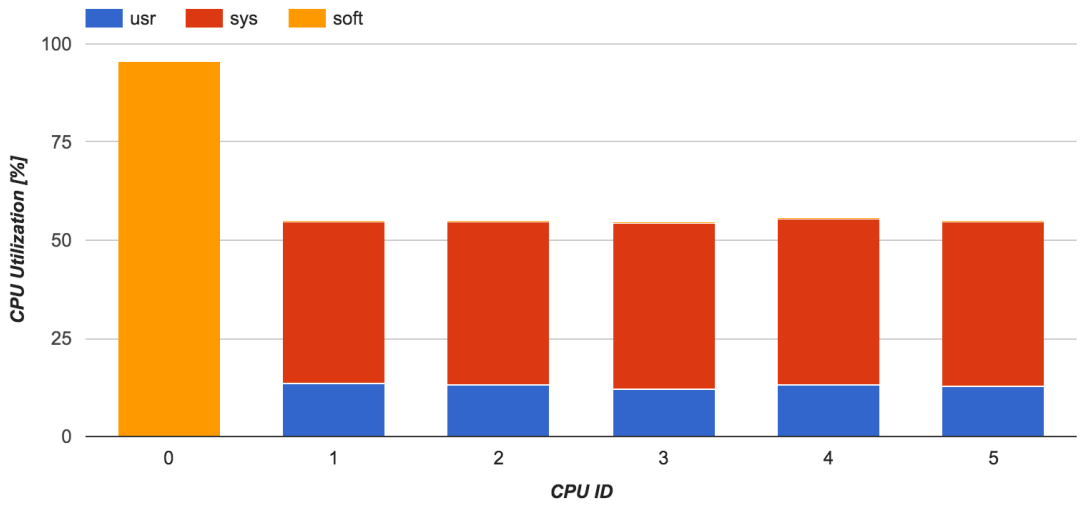

Figure 3.5: Memcached CPU Utilization with 6 threads by Individual CPU

Figure 3.5 shows the category breakdown of CPU utilization by each CPU for Memcached with 6 threads. We can observe that CPU 0 is processing all of the software interrupts while the remaining CPUs handle Memcached and the kernel. It is apparent that processing all of the software interrupts on a single CPU cores is a bottleneck in this case as the remaining CPUs are heavily underutilized. This phenomenon can be described as “load imbalance” [11]. The kernel is responsible for scheduling software interrupt processing and depending on the configuration it may choose to process all interrupts on a single CPU.

3.2.3 Threads Conclusion

In this section, we have explored the impact of threading on Memcached performance. We have reasoned that the best performance will be delivered when running with as many threads as there are CPU cores, however, our benchmarks have showed that this is in fact not the case. Closer inspection of the behavior has shown that the culprit is software interrupt processing on only one CPU core rendering it a bottleneck for the rest of the system. In order to solve this problem, we will examine explicit assignment of IRQ affinity to CPU cores in the next section.

3.3 IRQ Affinity

We have shown that increasing the number of threads itself does not result in improved CPU utilization and better Memcached performance. In this section, we focus resolving the problem of software interrupt processing on a single core. In order to resolve the load imbalance, it is important to understand how the kernel determines which CPU is responsible for processing a given interrupt.

Firstly, a request arrives at the network interface controller (NIC). The packet is processed by the NIC and pushed onto a receive queue. Secondly, an interrupt is raised to notify the kernel that an action is required. Thirdly, the kernel receives an interrupt, determines the application level destination of the request and inserts the request into a queue. Memcached utilizes *libevent* API to receive and send requests between the kernel and the application and is bound to the *epoll* socket [31]. When a request is inserted into the epoll socket, a level triggered software interrupt is fired [3]. Each Memcached thread is awaiting the software interrupt with *epoll wait()* [27] and the request is processed. In the case where a software interrupt is being processed on a core different from the recipient application, a context switch is required which contributes to high kernel space CPU utilization observed in previous benchmarks.

Therefore, we can reason distributing software interrupt processing across many CPU cores should lead to improved CPU utilization and improved overall performance in terms of throughput and latency as the overhead of context switching and the single CPU bottleneck are mitigated.

The kernel chooses which CPU core is responsible for processing a request is determined through IRQ Affinity. IRQ Affinity is an assignment of a given queue to a given CPU and the overall process is called IRQ Affinity pinning.

Firstly, let us inspect which IRQ queues are being used by the network interface. We can list the queues our *eth0* interfaces uses with the following command:

```
$ cat /proc/interrupts | grep eth0 | awk '{ print $1 " " $9 }'

81: eth0
82: eth0-TxRx-0
83: eth0-TxRx-1
84: eth0-TxRx-2
85: eth0-TxRx-3
86: eth0-TxRx-4
87: eth0-TxRx-5
```

We obtain a mapping of queues to individual Transmit and Receive (TxRx) queues. For each queue id, we can list their respective CPUs responsible for processing the queue. The following script provides us with this information:

```
$ for i in $(seq 81 1 87); do
    echo $i $(cat $/proc/irq/$i/smp_affinity_list);
  done;
```

```
81 0-5
82 0-5
83 0-5
84 0-5
85 0-5
86 0-5
87 0-5
```

We can observe that all of the queues are bound to all available CPUs (zero indexed). In order to assign a particular core to a given queue, we simply write the index of the CPU to the corresponding queue file. We can assign each queue with a unique core with the following script. Note that we leave *eth0* (queue 81) assigned to all CPUs as we do not want to bind processing of network requests to a specific core, we only want the transmit and receive queues to be bound.

```
$ for i in $(seq 82 1 87); do
    echo $(($i % 6)) > /proc/irq/$i/smp_affinity_list;
  done
```

With the transmit and receive queues assigned, we are now in a position to determine the effect of IRQ affinity pinning on Memcached performance. We will be using the same configuration as in Section 3.2 as well as increasing the number of threads linearly.

3.3.1 Threads, Latency & Throughput with IRQ pinned

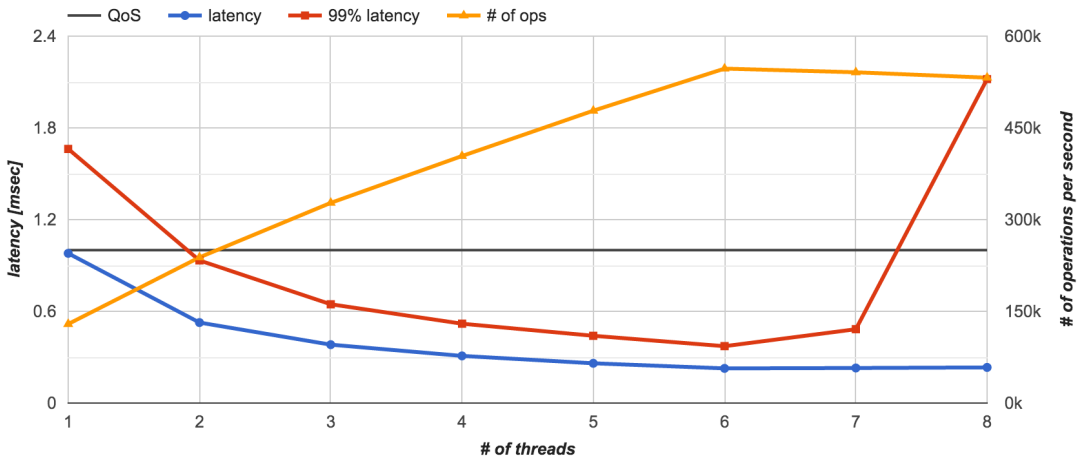

Figure 3.6: Memcached Latency & Throughput vs Threads with IRQ Affinity pinned to distinct cores

Figure 3.6 plots the relationship between latency on the left vertical axis, number of operations per second on the right vertical axis and the number of threads on the horizontal axis.

The mean latency decreases as we increase the number of threads and flattens out at 0.23ms at 6 threads or more. Tail latency, the 99th, decreases as we increase the number of threads. It reaches a minimum of 0.37ms, well within our QoS, at 6 threads and begins to increase as threads increase further. There is a sharp increase between 7 and 8 threads bringing it outside of our QoS. The number of operations per second remains constant between 1 and 2 threads at 240k requests per second. As the number of threads increases, so does throughput. In fact, throughput increases linearly between 2 and 6 threads and peaks at 546k requests per second. Throughput decreases slowly as the number of threads increases beyond 6.

Firstly, throughput improved drastically. In the peaks, we are able to achieve 546k requests per second with IRQ pinned compared to 264k requests previously. According to expectation, throughput is maximized with as many threads as CPU cores. Additionally, we can see throughput decline with more threads than CPU cores as context switching overhead increases.

Secondly, the 99th percentile latency has decreased significantly. With six or less threads, we are now able to achieve the required QoS. Additionally, the 99th percentile latency reaches a minimum with 6 threads providing the best combination of low latency and highest throughput. This is according to our expectation.

3.3.2 CPU Utilization with IRQ pinned

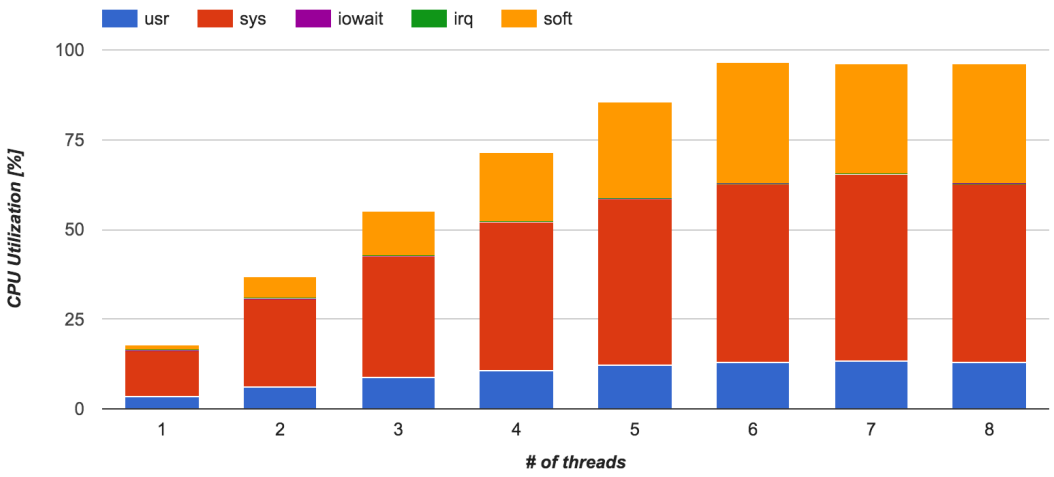

Figure 3.7: Memcached CPU Utilization with IRQ Affinity pinned

Figure 3.7 outlines the CPU utilization with IRQ Affinity pinned. The overall ratio of *usr*, *sys* and *soft* remains the same, however, we can observe that as we increase the number of threads the total utilization increases. We achieve near 100% utilization across all CPUs. Furthermore, CPU utilization breakdown does not change with more than 6 threads. This is reasonable as we are constrained by the number of cycles available to us, more Memcached threads are constrained to the same number of cycles while also incurring a context switching overhead. Therefore, more CPU threads than CPU cores do not result in better performance.

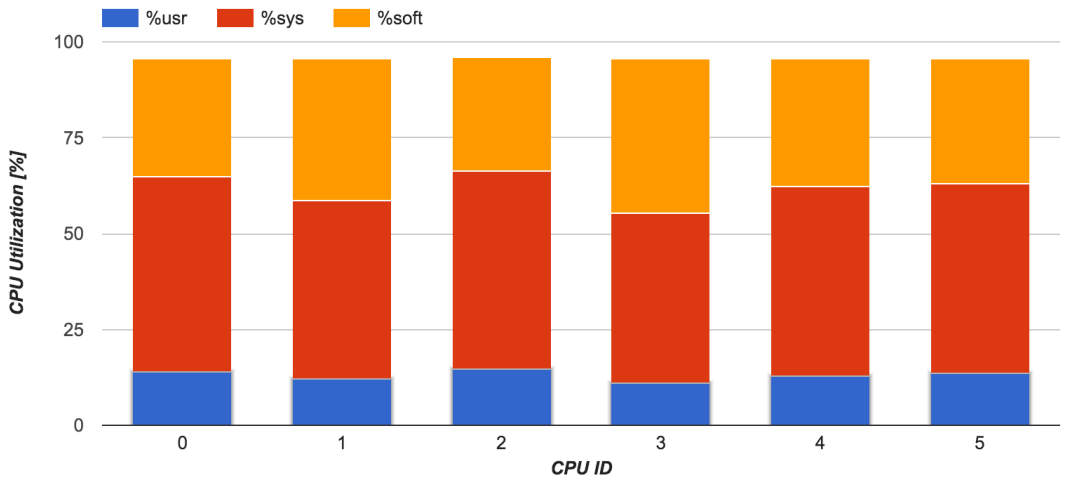

Figure 3.8: Memcached with 6 threads - Individual CPU utilization

Let us consider individual CPU utilization breakdown with 6 threads. Figure 3.8 outlines the utilization categories. We can observe that all cores are nearly 100% utilized as well as all cores participate in software interrupt processing. Utilization of each core is not the same, however. This can be due to a range of factors including given load on each Memcached thread as well as interference from kernel's memory management. Memcached also spawns an additional thread for load balancing purposes [6] which can cause interference in the individual load. This topic will be further examined in the following section.

3.3.3 IRQ Conclusion

We have shown that setting IRQ affinity to individual cores drastically improves performance of Memcached. Throughput has increased more than two fold while the 99th percentile latency decreased. Distribution of work across the CPU cores has improved and we have been able to fully utilize all of the cores available to us. For the rest of this chapter, all benchmarks executed will be considered with IRQ affinity set distinct cores.

3.4 Thread pinning

We now turn our attention to individual threads of Memcached. Thread pinning is the process of assigning a *set_irq_affinity* to each individual thread. As suggested by Leverich and Kozyrakis, "pinning memcached threads to distinct cores greatly improves load balance, consequently improving tail latency." [11] Furthermore, "While it is not strictly necessary, it may be additionally beneficial to pin every Memcache worker thread onto a single CPU to further improve performance by minimising cache pollution." [6] Therefore, we will investigate thread pinning and their impact on performance.

Firstly, it is important to understand how the kernel determines which core a process/thread will run on. It is the responsibility of the *scheduler* to maintain CPU cores busy and schedule work [25]. As such, the scheduler decided which core an application should be scheduled and executed on. As such, a scheduler aims to optimize for a given metric, such as utilization, priority or aims to avoid starvation. By default, when a new process is started it can be scheduled on any CPU core. As a result, the scheduler may choose to schedule multiple threads on the same CPU core. In order to explicitly prevent such behavior, we can set a distinct core for each thread using the *taskset* utility. We can discover the CPU affinity of a given process through the following command where *pid* is the process identifier.

```
taskset -p <pid>
```

”A Memcache instance started with n threads will spawn n + 1 threads of which the first n are worker threads and the last is a maintenance thread used for hash table expansion under high load factor.” [6]. We can discover memcached threads used for request processing using the following command where *tid* is the thread id discovered previously [6].

```
ps -p <memcache-pid> -o tid= -L | sort -n | tail -n +2 | head -n -1
```

For this benchmark, we use the same configuration as in previous sections. Tables 3.3 and 3.5 provides the complete configuration.

3.4.1 Latency & Throughput vs Threads

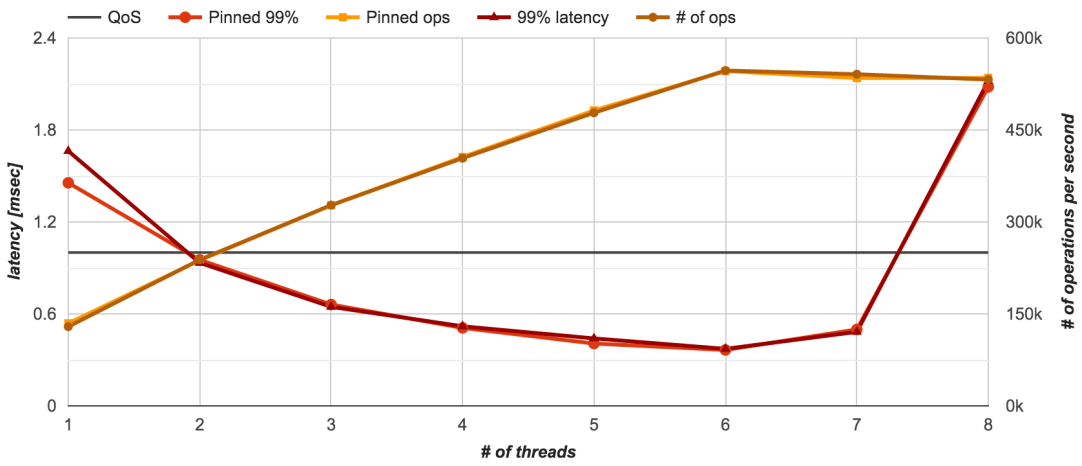

Figure 3.9: Memcached Latency & Throughput vs Threads: Comparison of pinned threads (labeled: *Pinned*) vs unpinned threads

Figure 3.9 presents a comparison of Memcached with pinned threads against unpinned threads. Overall, thread pinning appears to be no significant change in performance with threads pinned. In the case of Memcached with 1 thread, there appears to be an improvement in the tail latency, however, the QoS constraint is still violated.

We expected to see an improvement in tail latency or throughput. However, as other research suggests the improvement may only be observable in environments with a higher number of cores and/or in environments with shared workloads and only a fixed number of cores dedicated to Memcached [6].

3.4.2 CPU Utilization

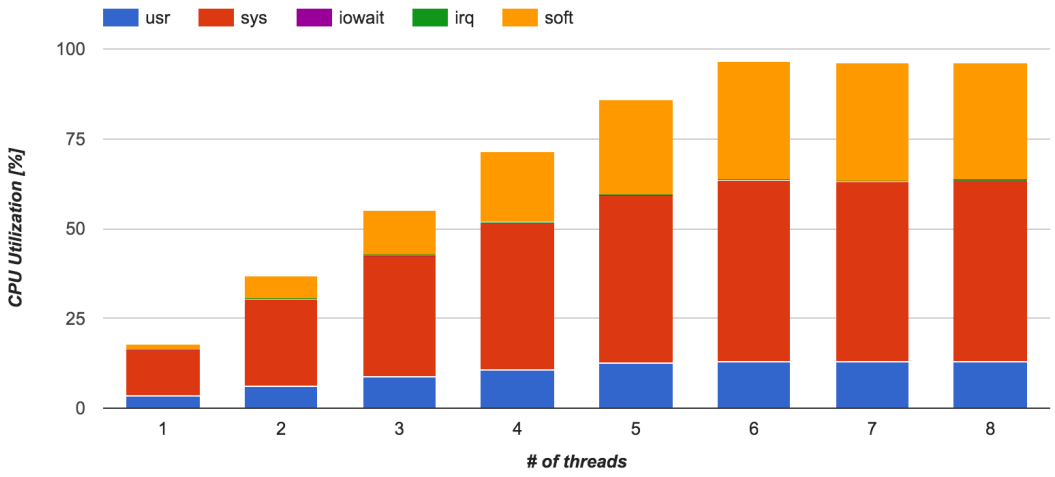

Figure 3.10: Pinned Memcached CPU Utilization

Figure 3.10 presents the overall CPU Usage with Memcached threads pinned. The CPU Utilization of pinned threads is nearly identical to unpinned threads presented in Figure 3.4.

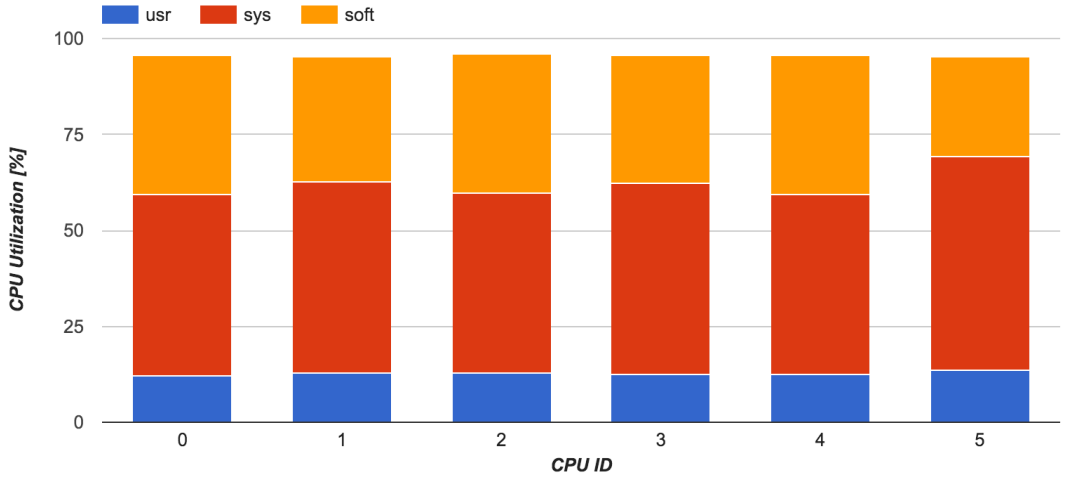

Figure 3.11: Pinned Memcached Individual CPU Utilization with 6 threads

Figure 3.11 plots the individual CPU usage breakdown. With thread pinning, we can observe a more evenly balanced distribution of CPU utilization as opposed to Figure 3.8.

3.4.3 Thread Pinning Conclusion

We have shown that thread pinning, in our benchmark, does not impact Memcached performance negative nor does it have a positive impact. CPU Utilization remains the same with thread pinning, however, the distribution of user, kernel and software interrupt processing is more balanced. Our findings are not consistent with findings reported by Leverich & Kozyrakis [11]. This may be due to scheduling policy inconsistency of the systems. However, it has been suggested that thread pinning reduces interference and therefore in subsequent benchmarks we will pin Memcached threads unless otherwise stated.

3.5 Group Size

In this section, we focus on the effect of connections on the performance of the cache. Memcached provides a configuration option *-R* to set the group size in the connection management policy of Memcached. The group size defines the "maximum number of requests per event, limits the number of requests processed for a given connection to prevent starvation (default: 20)" [7] and has a maximum value of 320 enforced in the implementation of Memcached [4]. This in effect means the number of requests processed from a single connection before memcached switches to a different connection to enforce a fairness policy.

In this benchmark, we focus on the effect of the group size on a Memcached deployment with 6 threads - the best configuration found so far. The benchmark increases the group size linearly in increments of 20 starting at 20 up to the maximum. The table below summarizes the Memcached and Memtier configurations used in this benchmark.

Memcached			Memtier		
Flag	Explanation	Value	Flag	Explanation	Value
-d	Daemon Mode	true	-s	Server	nsl200
-p	Port number	11120	-p	Port number	11120
-t	Thread count	6	-c	Conn. Count	7
-m	Memory	6144 (6GB)	-t	Thread Count	3
-R	Group Size	[20..320]	–key-minimum	Min Key	1
			–key-maximum	Max Key	100m
			–random-data	Gen random	true
			–data-size	Data Size	64

Table 3.5: Group Size benchmark configuration

3.5.1 Latency & Throughput

Figure 3.12 plots the relationship between group size, latency and throughput. Overall, there is very little difference in the performance at various group sizes. The throughput

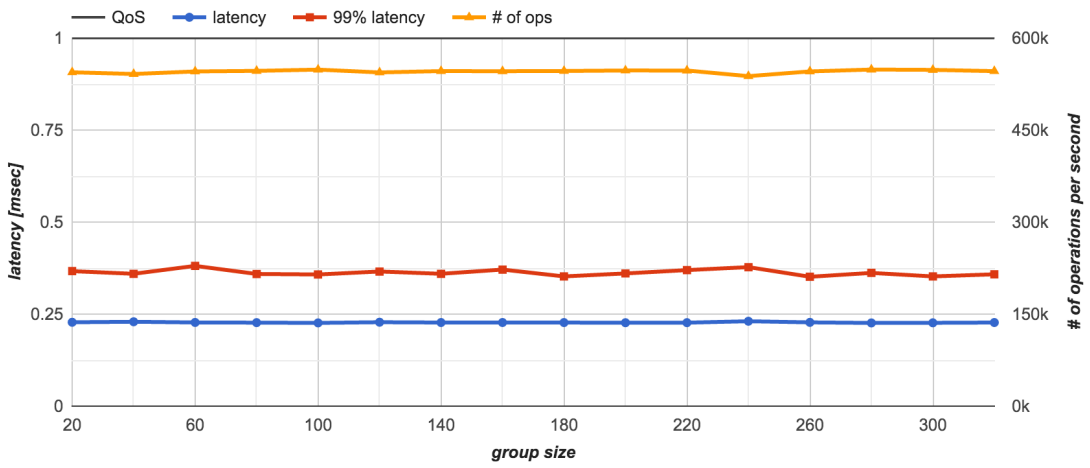

Figure 3.12: Latency & Throughput vs Memcached Group Size

remains constant at 545k requests per second with tail latency at 0.35ms. The mean latency remains constant too at 0.22ms.

3.5.2 CPU Utilization

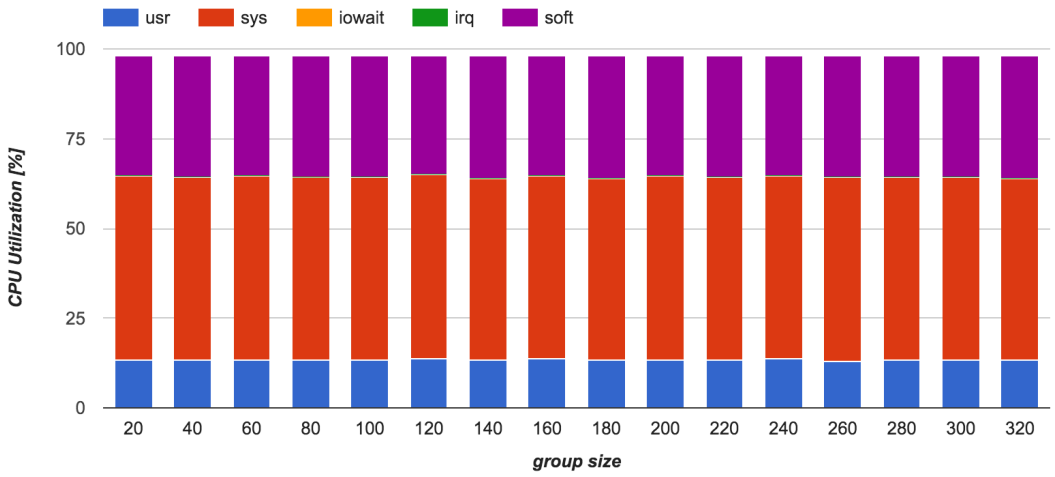

Figure 3.13: CPU Utilization vs Memcached Group Size

Figure 3.13 plots the CPU utilization reported by *mpstat*. We can observe that group size does not have any impact on the distribution of CPU utilization, nor does it impact the total utilization of the CPU.

3.5.3 Group Size Conclusion

In our benchmarks, we have observed that group size does not have any impact on the overall performance of Memcached. We have used 147 simultaneous connections

in our benchmark, however, a smaller number of connections with higher throughput may be required to effectively exploit the group size. The benchmark setup does not allow us to generate a high enough load from a singular connection and therefore we are unable to replicate the results obtained by Blake and Saidi [4].

3.6 Multiple Instances

In this section, we focus on a scenario with multiple separate Memcached instances (processes). In a production deployment, it may be desirable to logically split a Memcached deployment into multiple instances. This decision can be driven by application isolation, ability to migrate deployments independently or simply running multiple applications on the same server.

In the context of this paper, we consider an "instance" to be an application with an isolated view of the rest of the system. Therefore, when referring to an instance, we make the assumption that multiple instances cannot communicate with each other and more generally the instance makes no assumptions about the rest of the system or other applications on the same system.

We have previously concluded that the best performance is achieved with as many threads/processes as there are CPU cores. In this section, we build on top of this observation and design the benchmark with at most as many threads/processes as there are CPU cores.

Given the hardware configuration of this paper with 6 CPU cores, we can construct multiple scenarios of instances on the server. Table 3.6 outlines the configuration scenarios with respect to the number of instances, threads and the total number of threads/processes. Note that the scenario with only 1 instance corresponds the previously explored scenario and serves as a baseline for comparison.

Number of Instances	Threads per Instance	Total threads/processes
1	6	6
2	3	6
3	2	6
6	1	6

Table 3.6: Configuration Scenarios with at most 6 threads/processes on the server

For the purposes of this section, we will use a modified configuration for the workload generating clients. We have previously observed that Memtier with 3 threads and 7 connections provides a sufficient workload to saturate the object cache, however, achieves a tail latency close to 0.4ms. We increase the number of connections in this benchmark to 30 in order to gain flexibility for workload partitioning. This is necessary as 21 connections cannot be partitioned evenly across 6 instances due to indivisibility. Instead, we choose 30 connections per each workload generating host. The direct consequence of increasing the number of connections is expected to be an increase in the tail latency. Throughout the benchmark, we continue to target the required QoS.

Number of Instances	Threads per Instance	Memory per Instance
1	6	6GB
2	3	3GB
3	2	2GB
6	1	1GB

Table 3.7: Memcached configuration for multiple instances

Table 3.7 outlines the Memcached configuration at individual instance configurations. Note that the total amount of memory allocated remains constant. Additionally, when there are multiple threads per instance, each thread is pinned to an individual core.

Number of Instances	Threads	Connections	Key Maximum	Total Dataset
1	6	5	10 million	6.4 GB
2	3	5	5 million	6.4 GB
3	2	5	3.3 million	6.4 GB
6	1	5	1 million	6.4 GB

Table 3.8: Memtier configuration for multiple instances

Table 3.7 outlines the Memtier configuration at individual instance configurations. The total size of the dataset remains constant, however, the maximum key is lowered to allow for the increased number of instances. Furthermore, each workload generating host will execute as many instances of Memtier as there are corresponding instances of Memcached effectively spreading the load evenly across each individual instance.

3.6.1 Latency & Throughput vs Instances

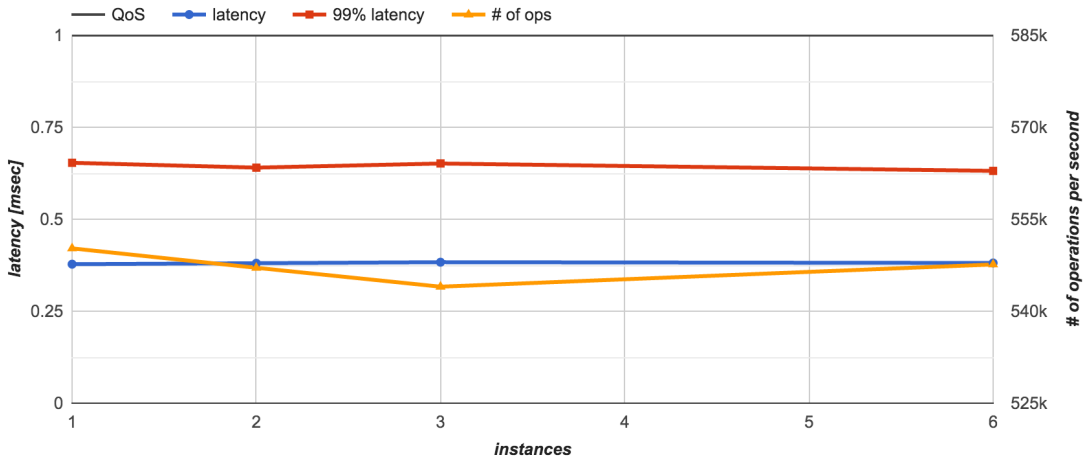

Figure 3.14: Multiple Instances: Latency & Throughput vs Number of instances. Note the scale of the right vertical axis starts at 525k.

Figure 3.14 plots mean latency, 99th percentile latency and throughput as the number of instances is increased. Mean latency and 99th percentile latency remain nearly constant as the number of instances increases. There is a very slight increase in latency at 3 instances, however, it is rather insignificant with respect to the scale. Throughput is maximized with only 1 instance and reaches a minimum at 3 instances, however, the difference is rather insignificant relative to the scale.

3.6.2 CPU Utilization vs Instances

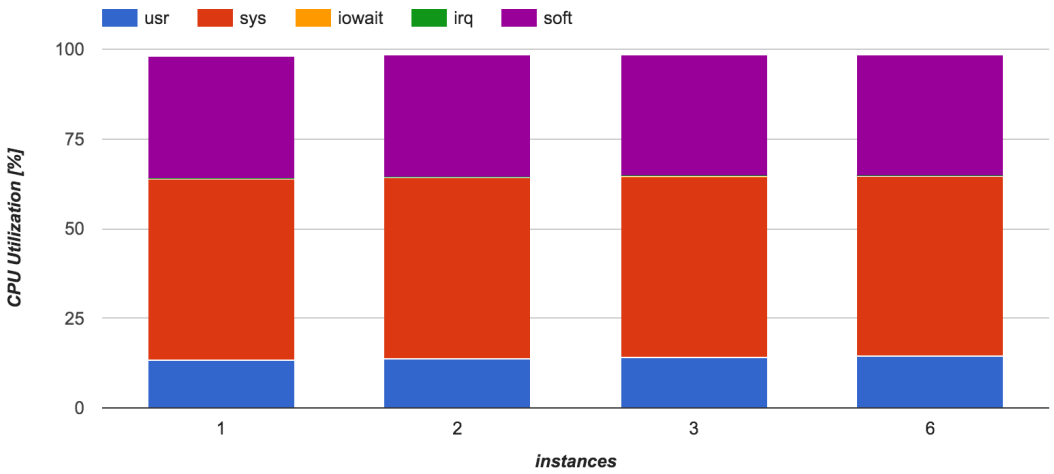

Figure 3.15: CPU Utilization with Multiple Instances

Figure 3.15 plots the CPU utilization with multiple instances. We can observe that the number of instances does not impact the total utilization nor does it impact the breakdown of individual category utilization.

3.6.3 Multiple Instances Conclusion

From the benchmark, we can conclude that multiple instances with proportionate allocation of resources do not incur any significant penalties in terms of performance. We have observed a slight decrease in throughput and increase in 99th percentile latency with a shared workload of 3 instances at 2 threads each. However, the relative scale of the change as well as the limited number of shared workload scenarios do not provide sufficient evidence to reach a conclusion.

3.7 Object Size

In this section, we turn our focus away from optimizing the performance of the cache itself onto the the effect of the data stored in the cache. We focus on the object size, the

data stored in the cache. In particular, we focus on the scalability of Memcached with larger objects. It has been reported that "object size distribution has a large impact on system behavior" [13].

In this benchmark, we utilize the same configuration as in Section 3.6, specifically we consider single instance Memcached with 6 threads since we found multi-instance setup of Memcached does not improve performance. On the client workload generation side, we reuse our configuration with 30 connections per each client host. We maintain the client side dataset constant at 6.4 GB, however, we adjust the maximum key based on the data size in order to maintain the same key range to data size ratio. Table 3.9 outlines the configuration details used for this benchmark.

Memcached			Memtier		
Flag	Explanation	Value	Flag	Explanation	Value
-d	Daemon Mode	true	-s	Server	nsl200
-p	Port number	11120	-p	Port number	11120
-t	Thread count	6	-c	Conn. Count	5
-m	Memory	6GB	-t	Thread Count	6
			–key-minimum	Min Key	1
			–key-maximum	Max Key	6.4GB / data_size
			–random-data	Gen random	true
			–data-size	Data Size	[64B..512KB]

Table 3.9: Memcached & Memtier configuration for Object Size benchmark

3.7.1 Latency & Throughput vs Object Size

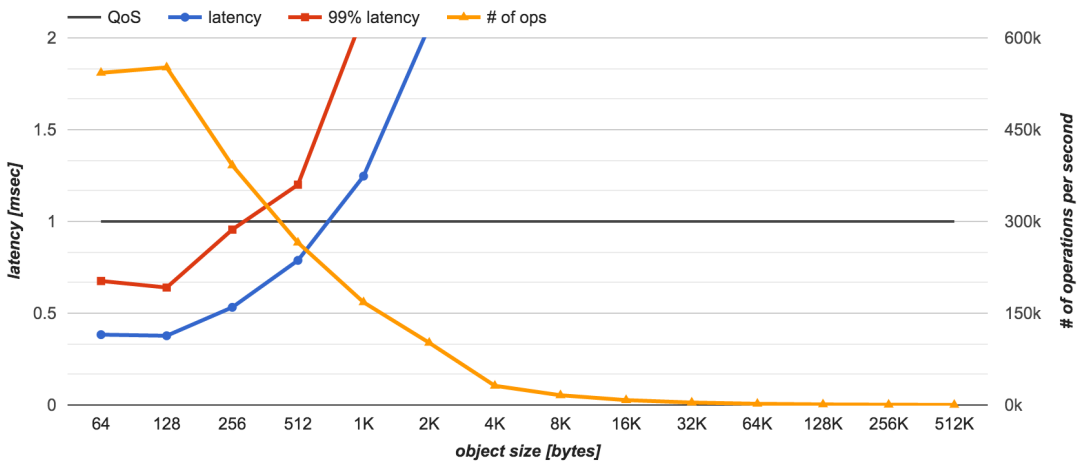

Figure 3.16: CPU Utilization with Multiple Instances

Figure 3.16 plots the relationship between object size, latency and throughput. As object size increases the number of operations decreases while the 99th percentile latency increases. This is an expected result as larger object sizes will incur increased latency.

Similar results are reported by [13]. Interestingly, with object sizes of 128 bytes we achieve the highest throughput with the smallest tail latency, however, this result is overshadowed by the overall trend of object size.

The QoS requirements are satisfied with object sizes between 64 and 256 bytes. With objects larger, we are no longer able to provide the required QoS.

3.7.2 CPU vs Object Size

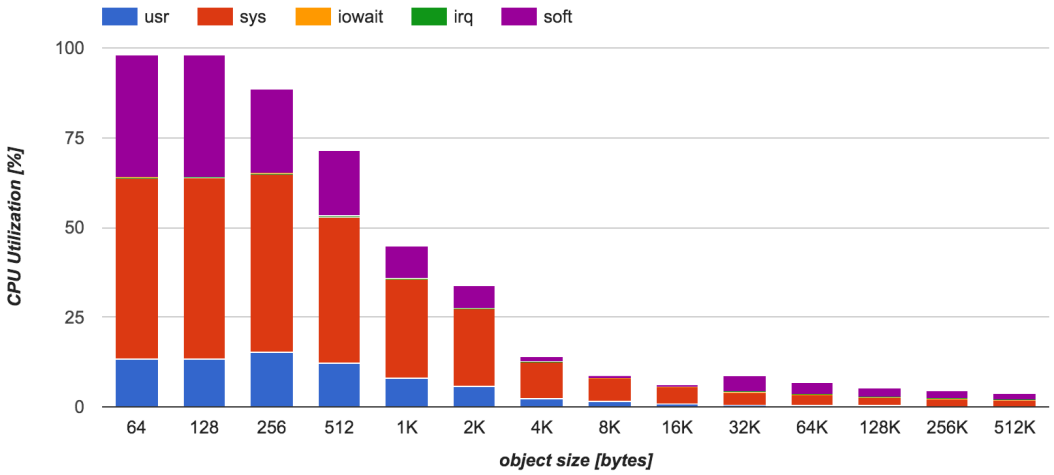

Figure 3.17: CPU Utilization with Multiple Instances

Figure 3.17 plots the CPU utilization against each object size. Overall, we achieve high CPU utilization with only objects of size 64 and 128 bytes. As the object size decreases, the total CPU utilization decreases too. The individual category breakdown remains the same with majority of CPU used by the kernel followed by software interrupts.

As object size increases, the network performance begins to dominate over the performance of the cache and/or the host CPU. Smaller objects, on the other hand, are processing constrained [13].

3.7.3 Object Size Conclusion

In this section, we have explored Memcached scalability with respect to object size. Our benchmarks show that Memcached scales well up to objects of size 256 bytes. Larger objects result in significant penalties in terms of throughput and tail latency. Reducing the load on our cache server, we would be able to obtain better scalability in terms of object size at the expense of throughput. Additionally, we have argued that large objects are predominantly network dependent. A faster NIC with corresponding switch interconnect would likely provide better scalability. In an analysis of Facebook Memcached workloads, a dominant portion of all object sizes fall below 270 bytes [2].

It is further argued that "small values dominate all workloads, not just in count, but especially in overall weight." [2] We can conclude that Memcached does not scale well for objects larger than 512 bytes. In practice, application design and architecture decisions can be made to either optimize Memcached for large object size performance and/or use many smaller objects with client side re-assembly.

3.8 Key Distribution

In this section, we investigate the effect of a skewed key distribution on the cache performance. In practice, "most web objects follows a zipf-like distribution, although the exact slope may vary" [13].

We consider a Zipf distribution defined by the following frequency function

$$f(k;s,N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)}$$

where k represents a given key, s represents an exponent describing the distribution (also called the zipf factor) and N is the total number of keys. Figure 3.18 plots the cumulative density function produced for a 10 million key range with varying values of s.

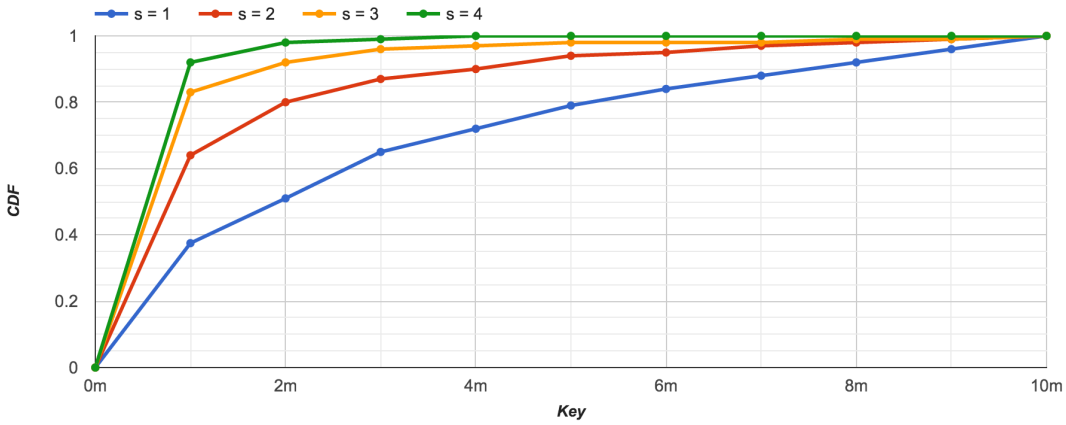

Figure 3.18: Zipf Cumulative Density Function with different values of s

In order to generate a zipf-like distribution with Memtier, a fork of memtier [16] with modified implementation is used as the official implementation does not support the zipf distribution.

3.8.1 Latency & Throughput

Figure 3.19 plots the relationship between latency, operations per second and the zipf factor. The higher the zipf factor, the more heavily skewed the distribution is. Overall, mean and 99th percentile latency remain nearly constant. Operations per second drop

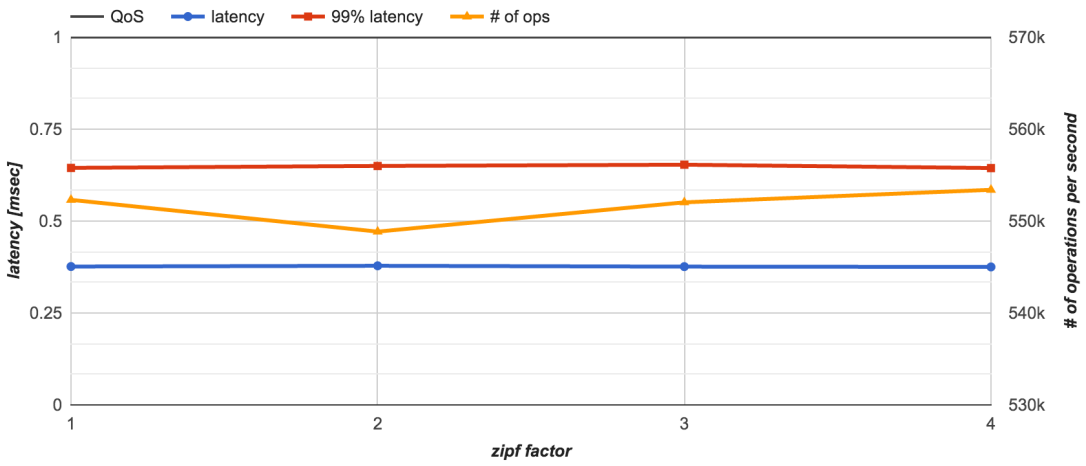

Figure 3.19: Memcached with Zipf-like key distribution

slightly at zipf factor of 2, however, on the relative scale of the operations per second it is rather insignificant.

The CPU utilization with a Zipf distribution remains the same as in Figure 3.7.

We find that there is no significant difference in overall performance of Memcached nor in CPU utilization with a zipf-like distribution. However, we also find that a Zipf-like key distribution increases throughput as opposed to a uniformly random distribution. We obtain an increase of 10k requests per second as opposed to results obtained after IRQ pinning.

Chapter 4

Redis

In this chapter, we will focus our attention on Redis. Redis is officially described as "in-memory data structure store, used as database, cache and message broker." [23] Like Memcached, Redis provides a simple text based API with *get*, *set*, and many more advanced features, for interaction over the network. Unlike Memcached, Redis is single threaded by design which introduces interesting scalability challenges.

We initially focus on the default performance of Redis when it is first deployed. Subsequently, we will turn our attention to Redis scalability as well as performance under various scenarios. We will consider the effect of multiple Redis instances, process pinning and IRQ Affinity. Furthermore we will explore Redis performance with different object sizes and key distributions.

Throughout this chapter, we focus on key performance metrics - throughput, mean and tail (99th) percentile latency as well as the target Quality of Service (QoS) of 99th percentile latency within 1 millisecond. Unless otherwise stated, all benchmarks are performed to target the QoS.

4.1 Out of the Box Performance

Firstly, we focus on the default Redis performance. Redis comes with a configuration file `redis.conf`[20] with sensible defaults. Redis supports data persistence management and by default keys are not evicted when the cache fills up. For the purposes of this paper, we are interested an eviction based cache and as such will configure Redis to use the Least Recently Used policy for key replacement. Configuration options can be supplied on the command line as well and override any configuration options specified in `redis.conf`. Table 4.1 summarizes our default Redis configuration.

With the configuration defined, we can deploy the Redis application with the following command.

```
redis redis.conf --port 11120 --maxmemory=6GB
    --maxmemory-policy=allkeys-lru
```

Configuration Option	Explanation	Value
–port	Port number	11120
–maxmemory	Maximum used memory	6GB
–maxmemory-policy	Policy for key evictions	allkeys-lru

Table 4.1: Redis Configuration

In order to get an initial feel for Redis performance, we setup the benchmark to increase load on the cache server linearly. Table 4.2 outlines the configuration used for the Memtier benchmark.

Configuration Option	Explanation	Value
-s	Server	nsl200 (server hostname)
-p	Port number	11120
-c	Number of Connections	[1..10]
-t	Number of Threads	2
–key-minimum	Smallest key	1
–key-maximum	Largest key	100 000 000
–random-data	Generate Random Data	true
–data-size	The size of data in bytes	64

Table 4.2: Memtier Configuration Options

Memtier can be started with the following command:

```
memtier -s <server> -p 11120 -c <connections> -t 2
  --random-data
  --key-minimum=1
  --key-maximum=100000000
  --random-data
  --data-size=64
```

We are showing the Redis deployment command as well as the Memtier command here for clarity, however, the commands will be omitted in subsequent sections.

4.1.1 Latency & Throughput vs Connections

Figure 4.1 plots the relationship between mean latency, 99th percentile latency and the number of operations per second against the number of connections. The mean latency increases with the number of connections linearly. Similarly, the 99th percentile latency increases linearly as the number of connections increases, however, it grows faster than the man latency. At 56 connections, we obtain a 99th percentile latency of 0.88 ms. A further increase in the number of connections leads to QoS violation. The number of operations per second remains constant with the number of connections and latency within QoS. The cache executes 93k requests per second, however, at this load we are unable to increase throughput while satisfying our QoS.

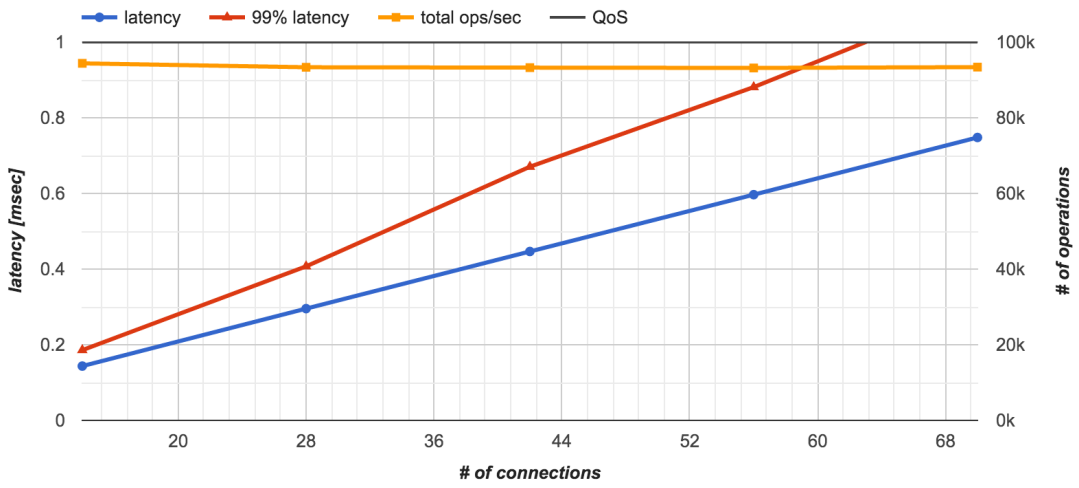

Figure 4.1: Latency & Throughput vs Number of Connections

4.1.2 CPU Utilization

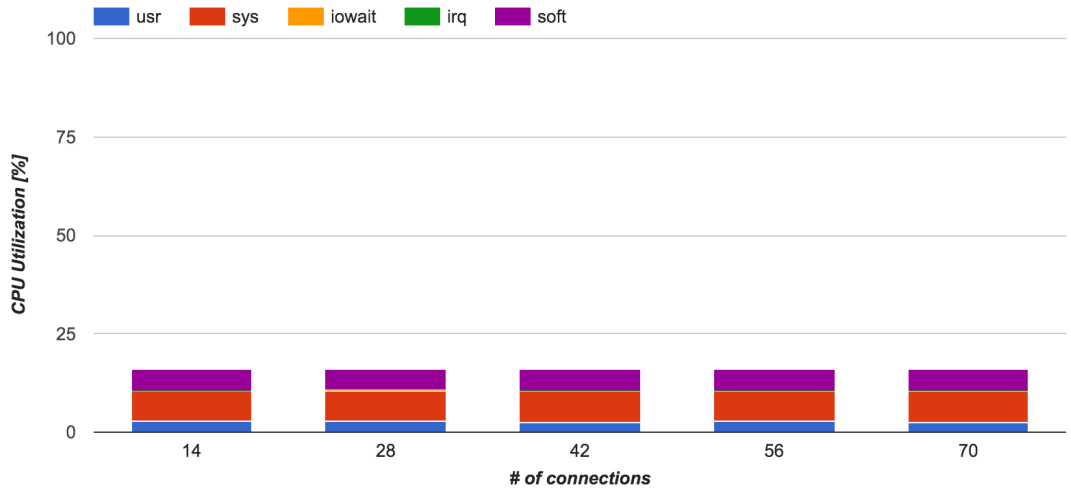

Figure 4.2: Default Redis CPU Utilization vs Number of Connections

Figure 4.2 plots the relationship between CPU utilization as reported by *mpstat* and the number of connections. Firstly, we can observe that the total utilization remains stable at 17%. We can clearly see that overall the server is heavily underutilized. However, the aggregate breakdown across all CPUs is misleading in this case. Figure 4.3 shows the individual CPU utilization for 56 client connections. We can observe that we achieve nearly 100% utilization on core 0 while the remaining cores are idle. Furthermore, we can observe that Redis accounts for only 14% utilization while the rest is used for kernel processing (46%) and software interrupt processing (34%). From the breakdown, we can conclude Redis utilization is kernel and software interrupt dominated rather than application processing dominated.

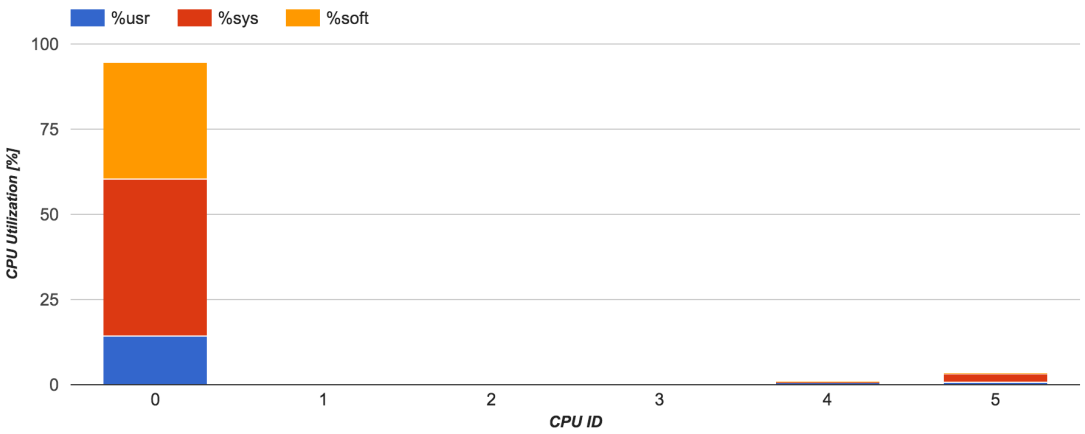

Figure 4.3: Default Redis Individual CPU Utilization at 56 connections

4.1.3 Conclusion

We have shown that Redis is capable of delivering 93k requests per second in its default configuration without any optimizations. Additionally, we have observed the impact of single threaded Redis on the overall host CPU utilization as well as the breakdown per individual core. We have also shown that the kernel and software interrupt processing play a significant role in overall performance.

4.2 Multiple Redis Instances

As seen in previous section, the overall cache server utilization suffers from single threaded nature of Redis. An immediate solution to this problem is to deploy multiple instances simultaneously. In this section, we consider a multi-instance Redis setup. It is important to keep in mind that each individual instance is completely isolated from the rest. As a result, the amount of memory space available to each instance decreases and therefore the key range we can store on each instance decreases. An application requiring access to a large number of keys will therefore be forced to partition the key space. Client side consistent hashing has the ability to alleviate this problem. Alternatively, a proxy application such as Twemproxy [29] can be used to spread the load and create a single point of access to the instances.

As the number of instances increases on the server, we expect the overall throughput to increase. As such, we initially utilize a larger number of connections than in the previous section, namely we use 210 connections (30 connections per each benchmarking host). We have determined empirically that 30 connections allow us to demonstrate Redis multi-instance scalability effectively. Additionally, 30 provides a great deal of flexibility when dealing with load partitioning across instances as we have 6 CPU cores available to us on the server. We consider 5 distinct multi-instance scenarios, they breakdown is outlined in Table 4.4.

Consequently, we can define the following Redis configuration for each scenario. We

Instance Count	Connections	Threads	Total
1	5	6	30
2	5	3	30
3	5	2	30
6	5	1	30
10	3	1	30

Table 4.3: Multi Instance Scenarios with Memtier connections and thread counts

maintain the same amount total memory while partitioning it evenly across instances. We of course also use the LRU Redis policy.

Instances	Memory	Total Memory
1	6GB	6GB
2	3GB	6GB
3	2GB	6GB
6	1GB	6GB
10	0.6GB	6GB

Table 4.4: Redis Maximum Memory config per number of instances

4.2.1 Latency & Throughput vs Instances

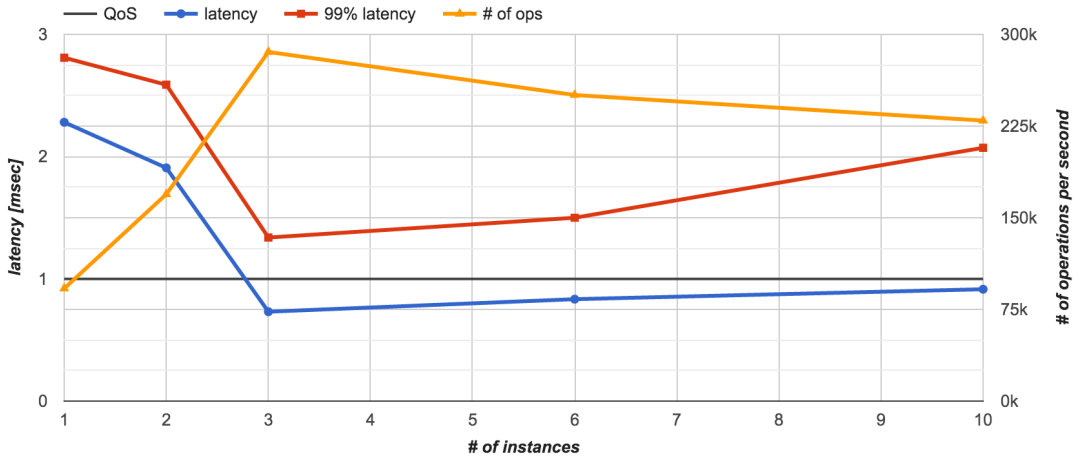

Figure 4.4: Redis Instances: Latency & Throughput

Figure 4.4 plots the relationship between latency, throughput and the number of instances.

Firstly, as the number of instances increases, both 99th and mean latency decrease and a reach a minimum at 3 instances. At 3 instances, we obtain a 99th percentile latency of 1.33 milliseconds, above the requires QoS. As the number of instances increases further, both mean and tail latency increase too.

Secondly, as the number of instances increases, throughput does too up to 3 instances where it reaches a maximum of 285k requests per second. A further increase in the number of instances leads to a decrease in throughput.

We can observe that multiple Redis instances do not scale as expected. We would expect the maximum throughput with the minimum tail latency to be achieved at 6 instances as we have 6 cpu cores. However, we maximize throughput while minimizing tail latency at 3 instances instead. Let us investigate the CPU utilization to get a better insight into the problem.

4.2.2 CPU vs Instances

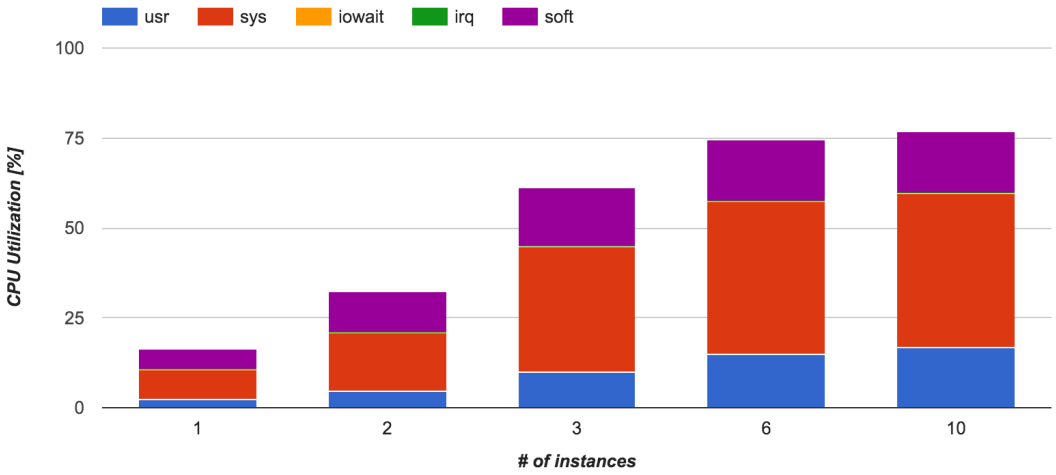

Figure 4.5: Redis Instances CPU

Figure 4.5 plots the CPU utilization against the number of instances with the respective category breakdown as reported by *mpstat*. We can observe that as we increase the number of instances, total CPU utilization increases. However, at no point do we reach a 100% utilization on the server. The utilization remains capped at 75% for configurations with 6 and 10 instances. Let us investigate the distribution of work on individual cores to get a better insight into the problem.

Figure 4.6 outlines the CPU utilization per individual CPU core with 6 instances. We can observe that all software interrupts are processed on core 0 with 100% utilization while the remaining cores remain underutilized. This renders core 0 a bottleneck for the remaining CPU cores and results in underutilization of resources.

4.2.3 Multiple Instances Conclusion

We have shown that multiple Redis instances do not scale linearly with the number of instances. In our benchmark, we have observed that the software interrupt processing is the bottleneck of multiple instances. We will address the load imbalance problem in the following chapter through IRQ Affinity pinning.

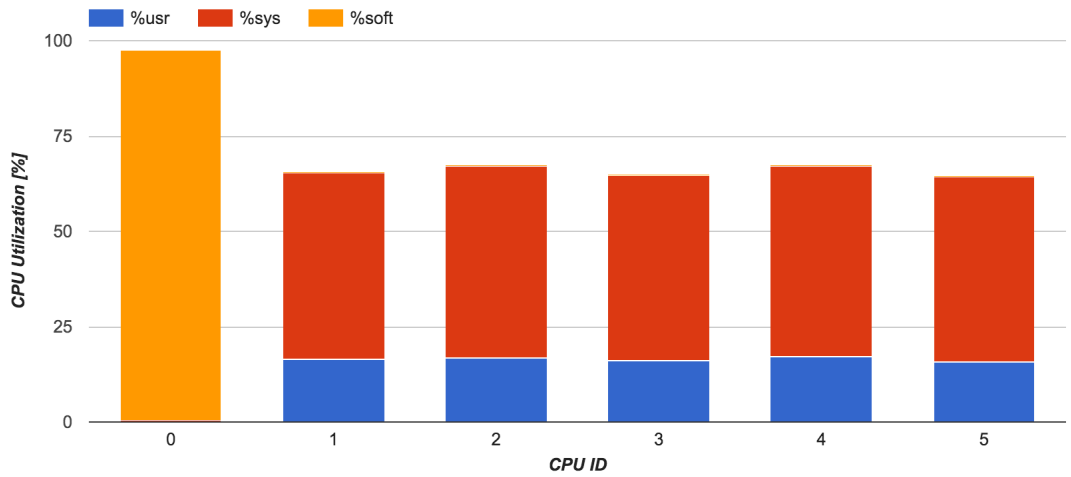

Figure 4.6: Redis Instances: Individual CPU Utilization at 6 instances

4.3 IRQ Affinity

In this chapter we turn our attention to addressing the load imbalance problem observed in when scaling Redis across multiple instances. In order to spread the software interrupt processing work across to multiple cores, we will assign each individual core a unique CPU core affinity. This process is outlined in detail in Section 3.3.

With IRQ Affinity assigned, we would expect the software interrupt processing to be spread evenly across all CPU cores and therefore removing the single CPU core bottleneck observed in the previous chapter.

4.3.1 Latency & Throughput with IRQ Affinity

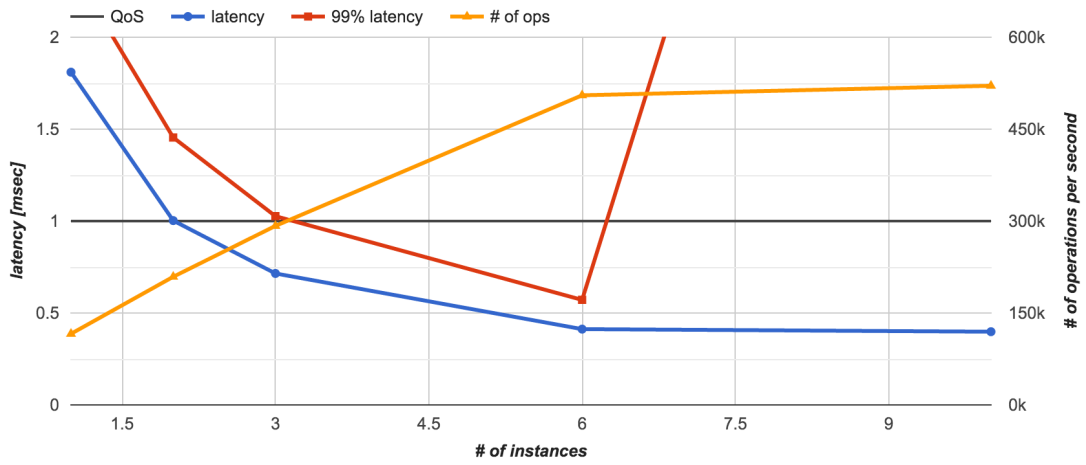

Figure 4.7: Redis Instances with IRQ Pinned: Latency & Throughput

Figure 4.7 plots the relationship between latency, throughput and number of instances after IRQ pinning. Firstly, mean latency decreases as the number of instances increases

up until it reaches a minimum of 0.4 ms at 6 instances. A further increase in the number of instances does not affect the mean latency. Secondly, the 99th percentile latency decreases sharply between 1 and 6 instances. It reaches a minimum of 0.57 ms at 6 instances. Additional instances lead to a sharp increase in tail latency. Furthermore, the number of operations increases steadily between 1 and 6 instances. There is a linear relationship between the number of instances and throughput with up to 6 instances. At 6 instances, we obtain 502k requests per second. A further increase in the number of instances leads to a slight increase in the number of operations, however, it is outside of the QoS constraints for tail latency.

4.3.2 CPU Utilization with IRQ Affinity

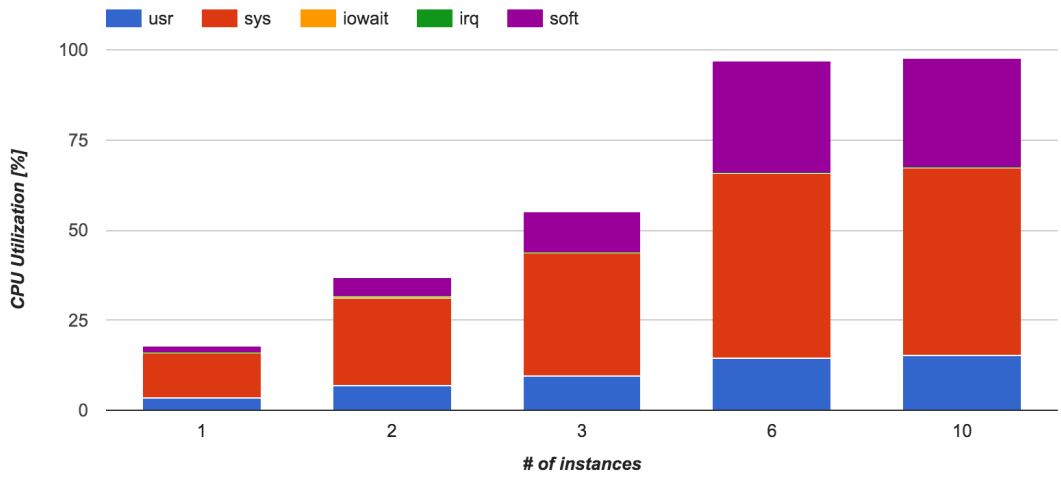

Figure 4.8: Redis Instances with IRQ Pinned: CPU Utilization

Figure 4.8 plots the category breakdown of CPU utilization against the number of instances. We can observe that the total utilization is nearly 100%. Additionally, we can observe that the proportion of kernel, user and software interrupt utilization increases linearly with the number of instances. This is indicative of good multi-instance scalability. At 10 instances, we obtain the same utilization as well as category utilization as with 6 instances. This is due to hard constraint on the amount of resources available on the server.

Inspecting individual CPU core utilization in Figure 4.9 at 6 instances, we can observe that software interrupt processing is spread across all CPU cores and allows all cores to be fully utilized.

4.3.3 IRQ Affinity Conclusion

In this section, we have shown that software interrupt processing is essential to good Redis performance. By spreading the software interrupt handling onto multiple CPU cores, we have been able to achieve linear scalability with the number of instances up to

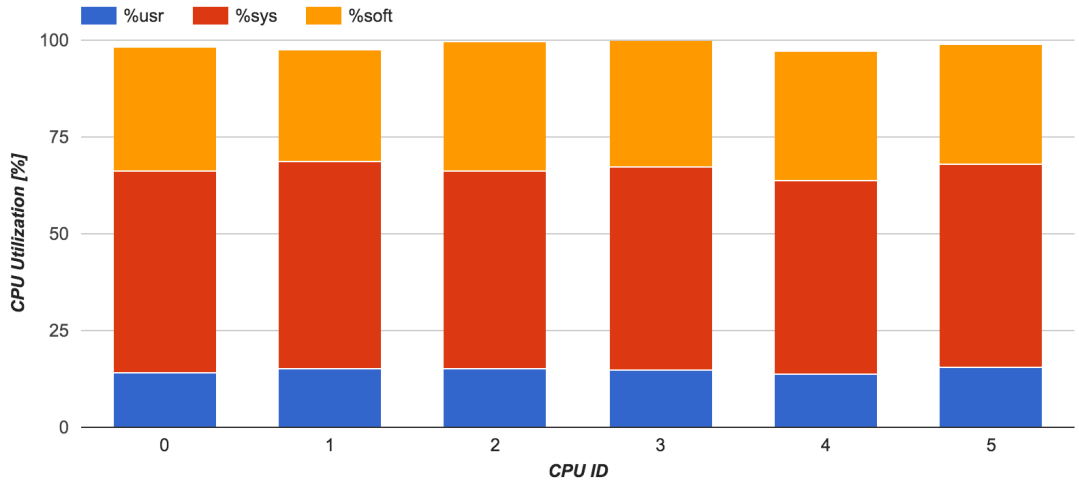

Figure 4.9: Redis Instances with IRQ Pinned: Individual CPU utilization at 6 instances

the hard limit of the number of CPU cores. This is significantly more than without IRQ pinning. Additionally, we have shown that Redis is capable of delivering 500k requests per second with 6 instances while satisfying QoS constraints. It is worth noting that the tail latency of Redis exhibits relatively small deviation from the mean latency. This suggests that there are only a small number of requests taking above average time to process and further indicates good scalability in terms of multiple instances.

4.4 Pinned Redis Instances

In this section, we take a brief look at process pinning. Process pinning is an assignment of CPU affinity to a particular process. Similarly to thread pinning, we can pin processes to individual cores. In this benchmark, we consider the best found configuration so far, that is 30 connections with IRQ affinity set.

Firstly, after starting Redis, we can discover the required process IDs through the use of the *ps* utility. Assigning process affinity can then be done through the *taskset* utility. The following bash command sets the CPU affinity of process *pid* to core *5*.

```
taskset -pc 5 pid
```

It has been suggested that process and thread pinning can help reduce “load imbalance” [11] by reducing interference caused by scheduling multiple simultaneous workloads. However, in our benchmarks we have not been able to obtain any significant speed up by pinning Redis processes to individual cores. We have obtained nearly identical results as with IRQ pinning and for clarity purposes will not be including a figure.

4.5 Object Size

In this section, we shift our focus from improving the performance of the cache itself to understanding how it performs under various scenarios, namely how does the size of each object stored impact overall performance within the QoS.

We consider the case of increasing the object size by a factor of 2 at each step. We utilize the same configuration for Memtier as in Section 3.7. The generated dataset remains the same in size while the range of possible keys shifts to account for the increased individual object size.

4.5.1 Latency & Throughput vs Object Size

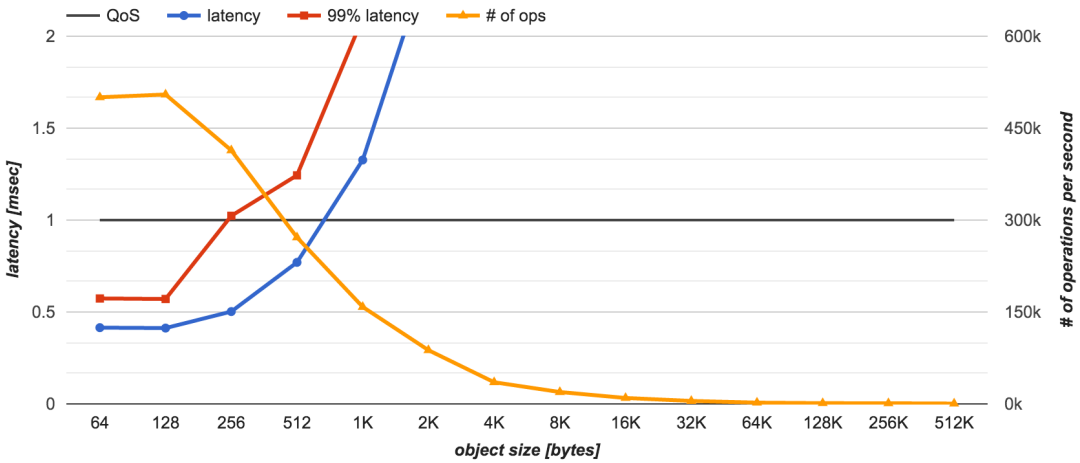

Figure 4.10: Redis & Object size: Latency and Throughput

Figure 4.10 plots the relationship between object size, latency and throughput. Please note that the horizontal axis increases in multiples of 2.

Initially, we start with object size of 64 bytes, the object size used all previous benchmarks. This serves as a baseline. As object size increases to 128, we observe an slight increase in the number of operations. Mean and tail latency remain the same.

Subsequently, as object size grows, we observe a decrease in the total number of operations and an increase in both mean and tail latency. In fact, QoS constraints are already violated with object sizes 256 bytes large. As object size increases even more, throughput falls drastically while mean and tail latency sky rockets.

As object size increases, the pattern of execution becomes network dominated rather than computation dominated. Larger objects take longer to transmit resulting in increased tail latency.

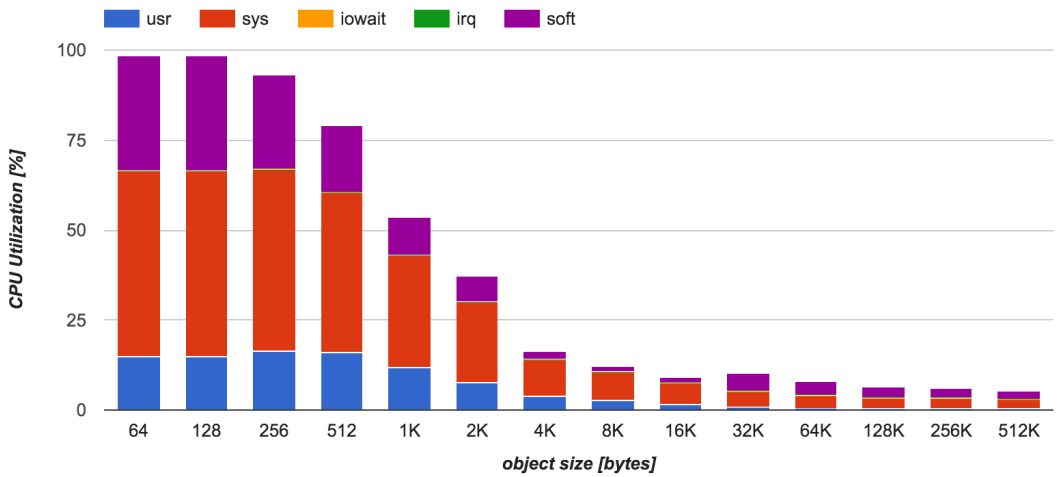

Figure 4.11: Redis & Object size: CPU Utilization

4.5.2 Latency & Throughput vs Object Size

Figure 4.11 considers the effect of object size on the CPU utilization of the server. Initially, we obtain a 100% utilization on the server side with object sizes 64 and 128 bytes large. As object size increases, and the tail latency increases, the inter-arrival time of each packet also increases. This results in reduced utilization. Effectively, the server is idle awaiting work. This pattern dominates as object size increases further.

4.5.3 Object Size Conclusion

We find that Redis does not scale well with object size under the required QoS requirements. This is the result of larger objects being network dominant rather than processing dominant. Relaxation of the QoS requirements for larger objects would allow us to scale Redis to larger object sizes better. With the current QoS requirements, a system could split an object into smaller values and store them independently. This would require disassembly of the object as well as re-assembly upon reception, however, individual requests would be able to meet the QoS.

4.6 Key Distributions

In this section, we consider the effect of a Zipf-like distribution on the performance of Redis. We use the same benchmark setup as presented in Section 3.8.

We consider a Zipf-like distribution with multiple zipf factors. Note that the higher the zipf factor, the heavier the skew towards a portion of the keys.

4.6.1 Latency & Throughput

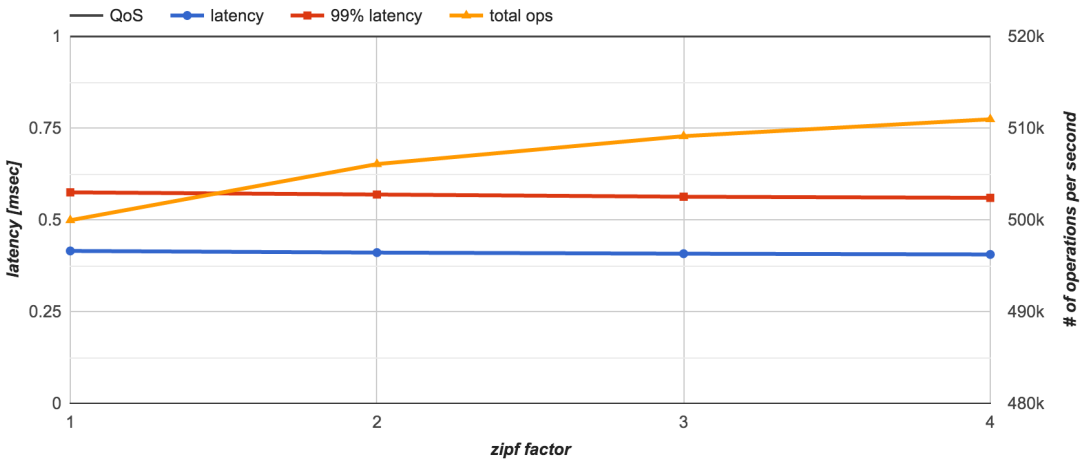

Figure 4.12: Redis with Zipf-like key distribution

Figure 4.12 plots the relationship between latency, operations per second and the zipf factor. As the zipf factor (skew) increases, the 99th percentile latency decreases slightly from 0.57 ms to 0.55 ms. The number of operations per second increases from 499k requests per second to 510k requests per second as the skew of the distribution increases. The decrease in the number of operations and increase in 99th percentile latency is likely due to spacial locality of the requests resulting in higher hit rate of the CPU cache.

The CPU Utilization of the server remains the same in the IRQ benchmark, Figure 4.8 with a total utilization at near 100%.

We find that Redis performance improves on a Zipf-like key distribution. Throughput increases by about 2% while the 99th percentile decreases very slightly.

Chapter 5

Redis & Memcached: Head to Tail

Having explored both Memcached and Redis, we now turn our attention to a 'head to tail' comparison of the object caches. Initially, we focus on a comparison of the default performance. Subsequently, we expand the scope to multiple threads as well as multiple processes.

5.1 Out of the Box

Firstly, let us evaluate the default performance of Memcached and Redis (M&R). The default performance is an important benchmark as well as a baseline. It is likely that M&R users who simply require a cache which is 'good' enough - performs to expectations, however, scalability is not a concern - will spend less time optimizing the cache performance and tuning the stack.

At this point, it is important to note that Memcached is a multi-threaded application with 4 threads by default. Figure 5.1 plots M&R performance with default configuration.

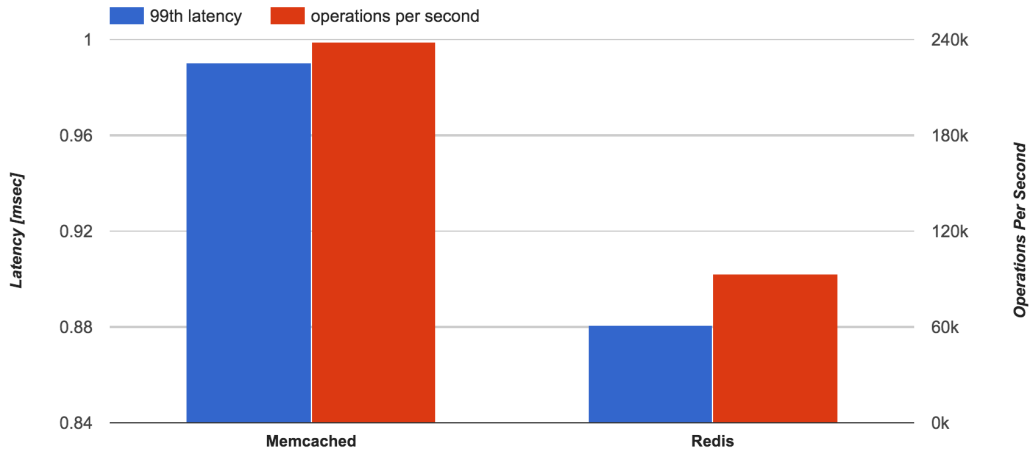

Figure 5.1: Out of the Box config: 99th percentile latency & Operations per second

Memcached performs better in its default configuration. This is not an unsurprising result provided Memcached runs with 4 threads while Redis is single threaded. Interestingly, we would expect the performance of Memcached to be on the order of 4 times as much as Redis, however, Memcached only achieves close to 240k requests per second while Redis achieves 93k requests per second. This is about 2.5 times less than Memcached.

With default configuration, Memcached outperforms Redis. This is simply the result of utilizing 4 threads with Memcached. However, in the default configuration Memcached suffers from scalability inefficiencies. A comparison of a 4 threaded application vs a single threaded application may seem unfair, however, it is important to understand the baseline performance. In subsequent sections, we will focus on comparisons on a more even ground.

5.2 Scaling Up

Let us now consider Memcached with multiple threads in comparison to multiple Redis instances. Both M&R use the same benchmark configuration.

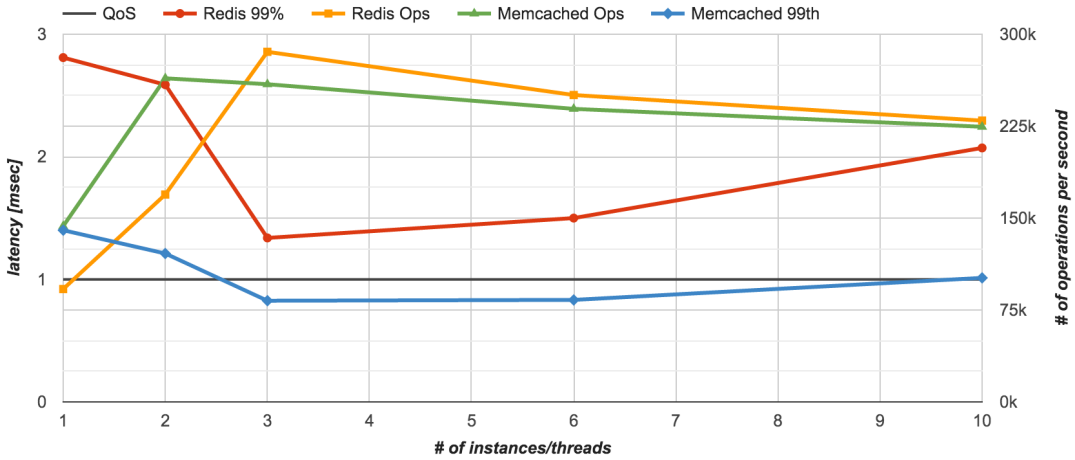

Figure 5.2: Memcached Threads vs Redis Instances: 99th percentile latency & Operations per second

Figure 5.2 plots 99th percentile latency and number of operations per second for both M&R. Memcached clearly performs better, achieving below QoS 99th percentile latency with a minimum of 0.82 milliseconds at 3 threads. Redis, on the other hand achieves a minimum of 1.33 milliseconds at 3 threads and does not satisfy the QoS. Interestingly, both M&R reach their respective minimums at 3 threads or instances respectively.

Memcached throughput is maximized at 2 threads with 264k requests per second. Redis throughput, on the other hand, is maximized at 3 threads with 285k requests per second.

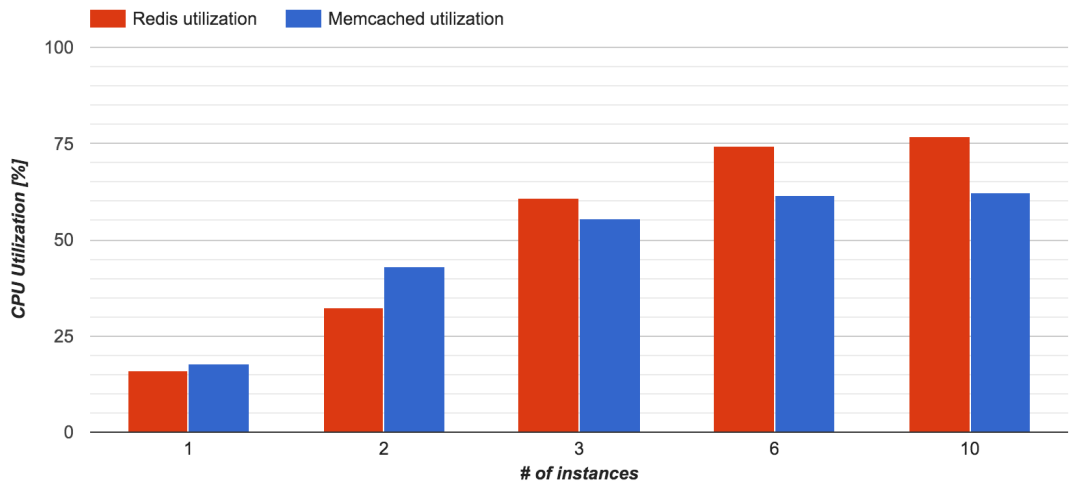

Figure 5.3: Memcached Threads vs Redis Instances: CPU Utilization

Figure 5.3 shows CPU utilization against instances. We can observe that at no point do we reach 100% utilization. This is indicative of the load balance problem explored in previous sections. Additionally, with 1 or 2 threads/instances, Memcached achieves higher CPU utilization than Redis, however, with 3 or more instances, Redis achieves better CPU utilization.

In this direct comparison, Memcached performs better as it actually achieves the desired QoS. However, both caches suffer from load imbalance - software interrupt processing only occurs on a single CPU core which makes it a bottleneck.

5.3 Scaling Up with IRQ Pinning

We have seen in previous chapters that software interrupt processing on a single CPU core greatly decreases overall performance of the cache. In this section, we compare M&R performance with IRQ pinned.

Figure 5.4 plots 99th percentile latency and operations per second for both M&R. Firstly, 99th percentile latency is minimized for both M&R at 6 threads/instances. Redis reaches a minimum of 0.57 ms while Memcached 99th percentile latency is 0.44 ms. Memcached clearly performs better in terms of 99th percentile at minimum. Additionally, Memcached achieves the desired QoS with 2 or more instances while Redis only achieves the QoS with 6 threads.

Secondly, the number of operations per second within QoS peaks at 6 threads for both M&R. Memcached achieves 546k operations per second while Redis executes 505k requests per second. Both caches experience a linear growth in throughput up to 6 threads at which point the number of operations flattens out.

Figure 5.5 plots the CPU utilization against the number of instances/threads. We can observe that CPU utilization increases linearly with the number of instances for M&R.

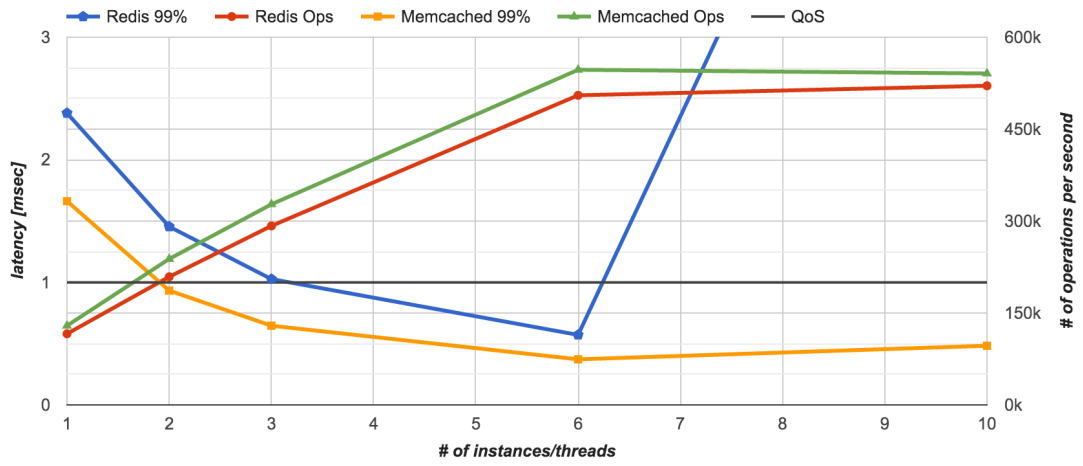

Figure 5.4: Memcached Threads vs Redis Instances with IRQ pinned: Latency % Throughput

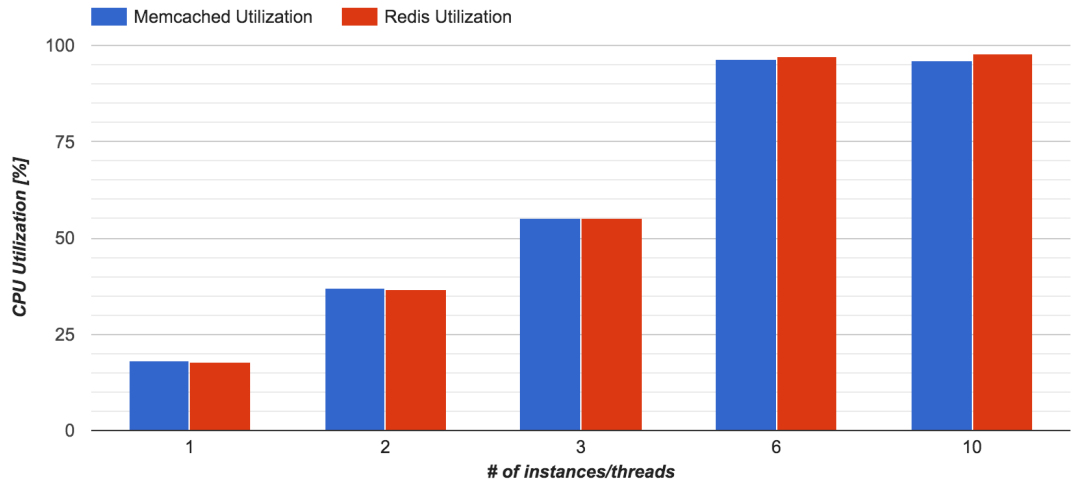

Figure 5.5: Memcached Threads vs Redis Instances with IRQ pinned: CPU Utilization

Additionally, M&R exhibit the same CPU utilization with a given number of instances. This is unsurprising given similar levels of throughput and latency as well as majority of execution time spent in the kernel and processing software interrupts, we would not expect to see significant differences between the two caches.

In a previous chapter on Memcached, we have also investigated the performance of Memcached when deployed in a multi-instance setup with 1 thread - in effectively single threaded mode. We have found that there is very little difference in performance (**??**). We concluded that a multi-instance Memcached setup with 1 thread performs as good as a single instance multi-threaded setup.

In previous chapters, we have considered additional optimizations such as thread/process pinning or group size in the case Memcached, however, we found that those optimizations did not increase performance further. Therefore,

Overall, we find that Memcached outperforms Redis. Memcached achieves lower 99th

percentile latency with higher throughput.

5.4 Object Size

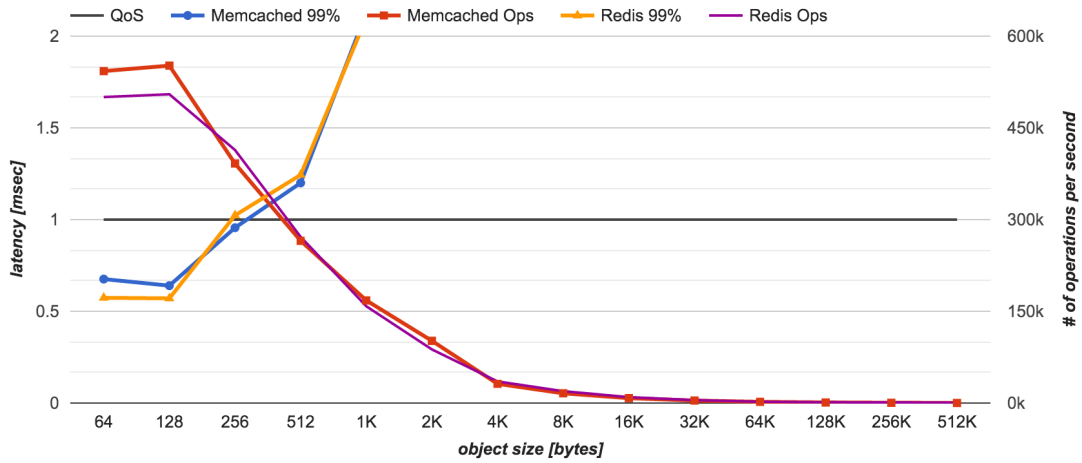

Figure 5.6: Memcached vs Redis: Latency & Throughput vs Object Size

In this comparison, we consider M&R with respect to object size. Figure 5.6 plots the 99th percentile latency and number of operations per second against object size. We find that both caches scale well up to 128 bytes, however, their performance begins to degrade with object sizes 256 bytes and greater. Memcached satisfies the QoS object size of 256 bytes while Redis exceeds the QoS mark. A further increase in object size leads to decreased number of operations per second and increased 99th percentile latency. The network performance starts to dominate the execution time and the overall performance degrades. This is further apparent from Figure 5.7.

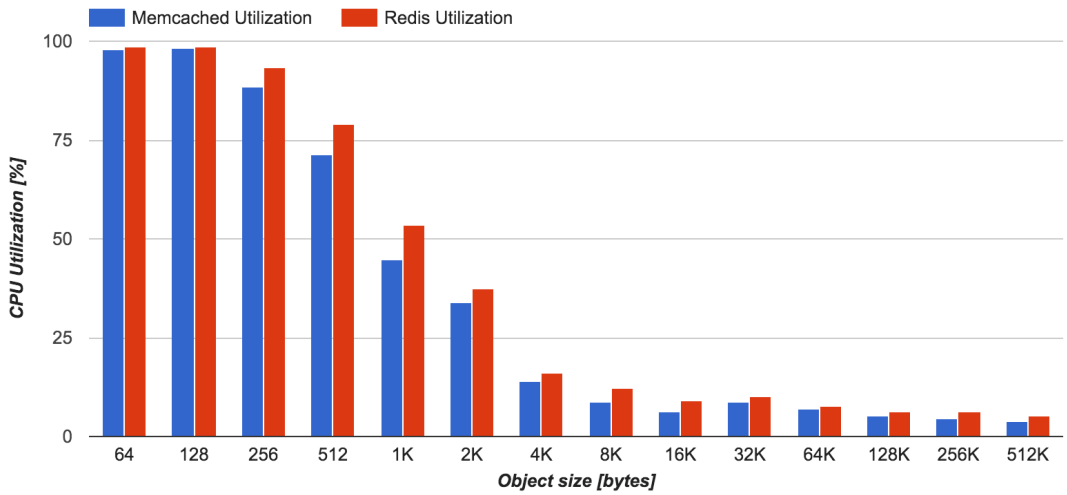

Figure 5.7: Memcached vs Redis: Latency & Throughput vs Object Size

As object size increases, we achieve lower server CPU utilization. Furthermore, we can observe that CPU utilization remains similar, with Redis slightly higher than Memcached, between M&R at various object sizes.

5.5 Key Distribution

In this comparison, we focus on the effect of a keys which follows a Zipf-like distribution.

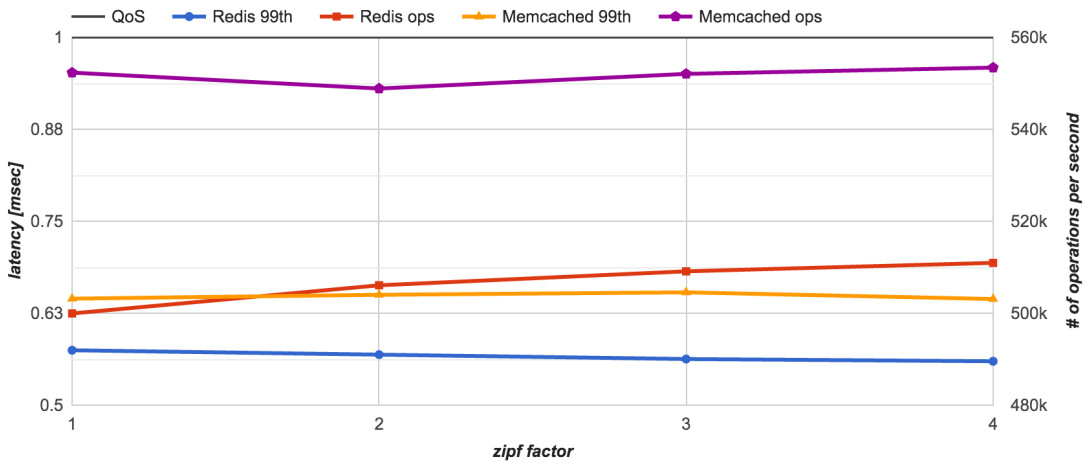

Figure 5.8: Zipf-like distribution of keys: Latency & Throughput vs Zipf factor

Figure 5.8 plots the comparison of M&R in terms of 99th percentile latency, operations per second and the zipf factor. We can observe that for both M&R, 99th percentile latency remains unaffected as the skew increases. In terms of operations per second, Memcached remains relatively stable with slight fluctuation, however, it does not exhibit a strong correlation with the skew. Redis, on the other, increases the number of operations as the skew increases. Interestingly, Memcached does not experience the same increase in throughput with increased skew, however, Memcached does experience an increase in the total number of operations, in comparison to results obtained after IRQ pinning, by about 10k. We can conclude that the skewed distribution has an impact on the performance of both caches. As the pattern of gets and sets revolves around the same keys, the overall performance increases. A likely explanation for Memcached's stable throughput with respect to the skew level is lock contention. It would appear that the lock contention does not cause Memcached to reduce throughput, however, it does not allow it increase it either.

Chapter 6

Conclusion

Bibliography

[1] Amazon. Elasticache. https://aws.amazon.com/elasticache/.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[3] Nick Black and Richard Vuduc. libtorque: Portable multithreaded continuations for scalable event-driven programs.

[4] Geoffrey Blake and Ali G Saidi. Where does the time go? characterizing tail latency in memcached. *System*, 54:53.

[5] highscalability.com. The architecture twitter uses to deal with 150m active users, 300k qps, a 22 mb/s firehose, and send tweets in under 5 seconds. http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html.

[6] Solarflare Communications Inc. Filling the pipe: A guide to optimising memcache performance on solarflare hardware. 2013.

[7] Danga Interactive. Memcached. http://memcached.org.

[8] Abhi Khune. Building a follower model from scratch. https://engineering.pinterest.com/blog/building-follower-model-scratch.

[9] Raffi Krikorian. Real-time delivery architecture at twitter. http://www.infoq.com/presentations/Real-Time-Delivery-Twitter.

[10] Redis Labs. Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark.

[11] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, page 4. ACM, 2014.

[12] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[13] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. *ACM SIGARCH Computer Architecture News*, 41(3):36–47, 2013.

[14] linux.die.net. mpstat(1) - linux man page. http://linux.die.net/man/1/mpstat.

[15] Kevin Montrose. Does stack exchange use caching and if so, how? http://meta.stackexchange.com/questions/69164/does-stack-exchange-use-caching-and-if-so-how/69172#69172.

[16] Milan Pavlik. Memtier benchmark with zipf generator. https://github.com/easyCZ/memtier_benchmark.

[17] Tom Preston-Werner. How we made github fast. https://github.com/blog/530-how-we-made-github-fast.

[18] Niels Provos and Nick Mathewson. libevent an event notification library. http://libevent.org.

[19] Redis. redis.io. http://redis.io/.

[20] Redis. redis.conf. https://github.com/antirez/redis/blob/3.0/redis.conf, 2015.

[21] RedisLabs. memtier_benchmark: A high-throughput benchmarking tool for redis and memcached. https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached#.VunpLhKLTMU, 2013.

[22] Paul Saab. Scaling memcached at facebook. https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919/.

[23] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2009.

[24] Jeff Scudder. Effective memcache. https://cloud.google.com/appengine/articles/scaling/memcache.

[25] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Chip multi processing aware linux kernel scheduler. In *Linux Symposium*, page 193. Citeseer, 2005.

[26] Paul Smith. Redis: under the hood. https://pauladamsmith.com/articles/redis-under-the-hood.html.

[27] Supachai Thongprasit, Vasaka Visoottiviseth, and Ryousei Takano. Toward fast and scalable key-value stores based on user space tcp/ip stack. In *Proceedings of the Asian Internet Engineering Conference*, pages 40–47. ACM, 2015.

[28] Twitter. Twemcache: Twitter memcached. https://github.com/twitter/twemcache.

[29] Twitter. Twemproxy. https://github.com/twitter/twemproxy.

[30] Alex Wiggins and Jimmy Langston. Enhancing the scalability of memcached. *Intel document, unpublished*, 2012.

[31] Hao Zhang, Bogdan Marius Tudor, Gang Chen, and Beng Chin Ooi. Efficient in-memory data management: An analysis. *Proceedings of the VLDB Endowment*, 7(10):833–836, 2014.