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

Abstract Goes here

Acknowledgements

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

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

Table of Contents

0.1 Abstract 3

1 Introduction **7**
- 1.1 Motivation 7
- 1.2 Memory Object Caches 7
  - 1.2.1 Desired qualities 7
  - 1.2.2 Design and Implementations 8
  - 1.2.3 Performance metrics 8
  - 1.2.4 Memcached 9
- 1.3 Methodology 10
  - 1.3.1 Quality of Service 11
  - 1.3.2 Hardware 11
  - 1.3.3 Workload generation 11
  - 1.3.4 Benchmark 12

2 Memcached **15**
- 2.1 Out of the Box Performance 15
  - 2.1.1 Latency, Throughput and Number of Connections 16
  - 2.1.2 CPU Utilization 17
- 2.2 Memcached Thread Scalability 18
  - 2.2.1 Throughput & Latency 19
  - 2.2.2 CPU Utilization 20
  - 2.2.3 Thread Evaluation 21
- 2.3 Thread pinning 22
  - 2.3.1 Latency & Throughput vs Threads 22
  - 2.3.2 CPU Utilization 23
  - 2.3.3 Thread Pinning Conclusion 23
- 2.4 Group Size 23
  - 2.4.1 Latency & Throughput 24
  - 2.4.2 CPU Utilization 24
  - 2.4.3 Group Size Conclusion 25
- 2.5 Receive & Transmit Queues fixing 25
- 2.6 Multiple Memcached Processes 25

3 Redis **27**
- 3.1 Out of the Box Performance 27
  - 3.1.1 Latency vs Throughput 28

3.1.2 CPU Utilization 29
3.2 Multiple Redis Instances 29
3.2.1 Latency and Throughput 31
3.2.2 CPU Utilization 32
3.3 Pinned Redis Instances 33
3.3.1 Pinned Latency and Throughput 33
3.3.2 Redis Persistence 34
3.4 Object Size 34
3.4.1 Latency and Throughput 34
3.4.2 CPU Utilization 35
3.5 Key Distributions 37
3.5.1 Gaussian distribution 37
3.5.2 Zipf distribution 37

4 Redis & Memcached: Head to Tail 39

5 Conclusion 41

Bibliography 43

Chapter 1

Introduction

1.1 Motivation

TODO

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 hardware in order to not incur hard drive retrieval delay. As a result, memory caches are often explicitly configured with the maximum amount of memory available.

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.

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, due to memory restrictions, the cache is responsible for enforcing an eviction policy. Most state of the art caches utilize least recently used (LRU) cache eviction policy, however, other policies such as first-in-first-out can also be used.

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.

1.2.4 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.” [5] Despite the official description aimed at dynamic web applications, memcached is also used as a generic key value store to locate servers and services [1].

1.2.4.1 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.

1.2.4.2 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.” [4]

Secondly, in order to provide performance as well as portability, memcached is implemented on top of *libevent* [10]. “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.” [10]

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 [13]

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.

1.2.4.3 Configuration

TODO

1.2.4.4 Production deployments

TODO: Discuss Facebook, Amazon, Twitter, ... deployments of memcached

1.3 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.

1.3.1 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 [7].

1.3.2 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.

1.3.3 Workload generation

Workload for the cache server is generated using Memtier Benchmark developed by Redis Labs [6]. 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.

1.3.4 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.

1.3.4.1 Memtier

Memtier benchmark is “a command line utility developed by Redis Labs for load generation and benchmarking NoSQL key-value databases” [6]. 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` [10], 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 1.1 outlines the relevant configuration options. The complete set of configuration options is available on RedisLabs [12].

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 1.1: Memtier Configuration Options

1.3.4.2 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.

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 2

Memcached

The purpose of this chapter is to benchmark and evaluate memcached performance. Firstly, we will examine performance under default configuration of both the server and the client. Secondly, threading will be explored in relation to latency and throughput. Thirdly, the effect of memcached's `group size` will be explored in relation to performance. Additionally, configuration of receive and transmit queues will be explored and finally, an execution model of multiple processes will be visited in order to establish a comparison baseline. Throughout the benchmarks, we will be focusing cache performance which meets desired the QoS.

2.1 Out of the Box Performance

Firstly, we consider Memcached in it's default configuration. A list of the configuration parameters as well as the default values is presented in Section 1.2.4.3. The purpose of benchmarking Memcached in the default configuration is to obtain a baseline performance. This will in turn allow us to consider potential optimizations with respect to the baseline.

The Memcached server is started with the following command.

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

We run Memcached in daemon mode (*-d*), set the maximum amount of memory Memcached can use to 6GB (*-m*) and finally we bind Memcached to port 11120 (textit-p). Note that by default, Memcached runs with 4 threads.

In order to generate a workload with increasing intensity, the number of simultaneous connections is increased linearly. As the number of connections grows, so does the number of requests per second issued to Memcached. We configure the Memtier benchmark as follows:

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

```
--key-maximum=10000000
--data-size=32
```

Memtier is run simultaneously on 7 hosts (distinct from the server). The Memcached server and port number are provided (*-s*, *-p*). Each host executing Memtier runs 3 distinct threads with *c* connections in each thread. The number of connections increases linearly from 1 connection to 17 connections. As a result, the number of connections in consecutive benchmarks increases by 21. This is the result of running Memtier on 7 hosts with 3 threads per each host. Additionally, the *key-minimum*, *key-maximum* defines the key range utilized in this benchmark. and *data-size* configuration parameters are provided for clarity and are set to the default values configured by Memtier. The result of the above configuration is to generate a linearly increasing load on the Memcached server.

2.1.1 Latency, Throughput and Number of Connections

Firstly, we are interested in the relationship between throughput and latency shown in Figure 2.1. 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.

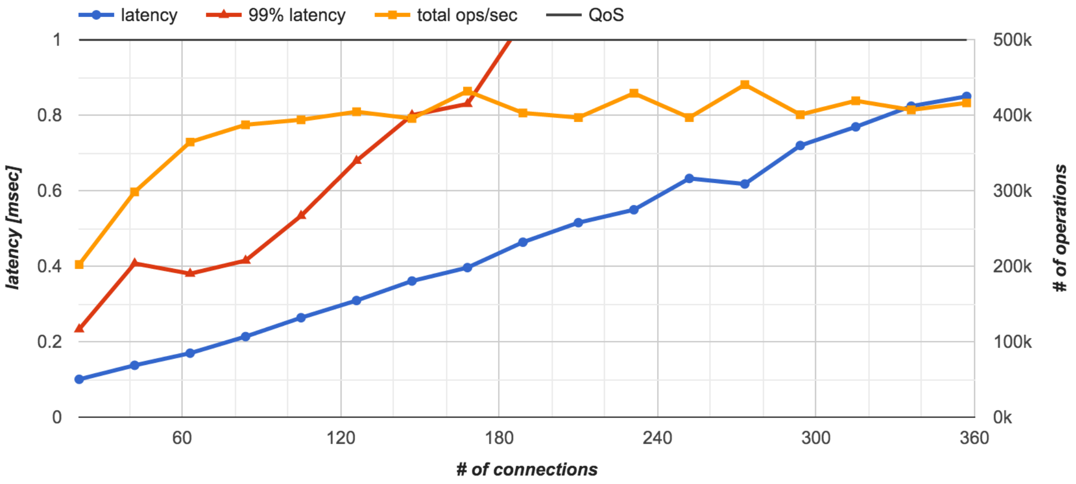

Figure 2.1: Latency & Throughput vs Number of Connections

As the number of connections increases, mean latency increases too. There is a linear trend between the number of connections and the mean latency. The 99th percentile latency also increases with the number of connections, however, it grows faster than than the number of connections does and exceeds the required QoS constraint at 189 simultaneous connections. The total number of operations, increases quickly as the number of connections increases and begins to flatten at 100 connections or 400k requests per second. As the number of connections increases beyond 100, the throughput remains relatively stable at around 420k requests per second.

Examining the relationship between latency and throughput, we can observe that initially we are able to increase throughput to 415k requests per second, however, a further increase in throughput comes at a disproportionately larger cost in terms of 99th percentile latency. This is reasonable as there is a limit to the number of requests we can process per second, a larger number of requests will incur queuing delay which translates to increased latency.

2.1.2 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* [9] utility which reports the percentage of CPU utilization broken down into multiple categories. The following table [9] 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 2.2 outlines the CPU Utilization broken down into mpstat categories. Note that *idle* percentage is represented in the chart as the remaining transparent area of each column.

Firstly, as the number of connections increases, the *usr* utilization remains nearly constant at 9%.

Secondly, *sys* utilization increases as the number of connection increases between 21 and 120 connections and remains relatively stable as the number connections increases further. This behavior corresponds to the saturation point also observed in the relationship of connections and throughput.

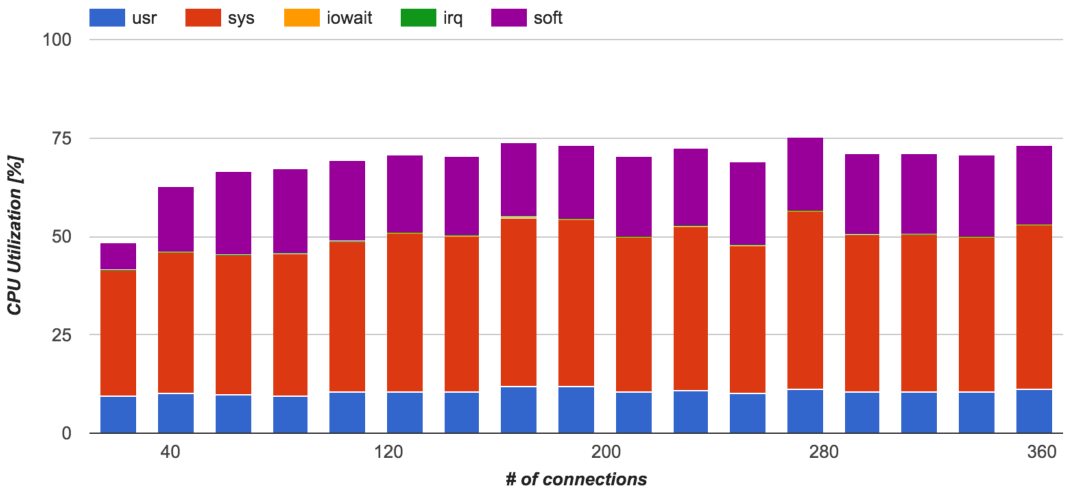

Figure 2.2: CPU Utilization for Out of the Box Configuration of Memcached

Thirdly, *soft*, increases as the number of connections increase due to the increased number of requests per second which are required to be serviced.

Furthermore, *idle*, represented as the remaining CPU utilization in each column, decreases as the number of connection increases. This is primarily due to an increase in *sys* and *soft*. The peak total CPU utilization is 75%, this corresponds to 4 Memcached threads on distinct cores being utilized at 100% and additional low utilization on the remaining 2 cores.

Finally, *iowait* and *irq* account for less than 0.01% of the total CPU utilization. This is a reasonable result as Memcached is an in-memory object cache and therefore we do not expect to see any disk I/O.

As a result of the breakdown, we can conclude that Memcached itself is not CPU intensive. Rather, the bulk of the CPU utilization is taken up by the kernel as well as processing software interrupts from high number of requests per second. The overall Memcached performance appears to be tightly linked to the performance of the network stack as well as the underlying kernel. This observation is consistent with findings in MICA [8].

Memcached performs reasonably well ‘out of the box’, delivering 420k requests per second with 99th percentile latency meeting the QoS. The CPU utilization is dominated by the kernel and the network stack. However, Memcached has a lot to offer in terms of configuration and performance improvements which we will explore in subsequent sections.

2.2 Memcached Thread Scalability

Memcached, as a high performance object cache, is designed to be executed on a multi-core architecture. It implements scalability through the use multiple threads allowing memcached to utilize many core architectures. Therefore, the next step in scaling a memcached deployment is to examine the impact threads have on Memcached's performance.

Memcached execution model is capable of processing incoming and outgoing requests in parallel, however, operations executed require a global application lock to be acquired. Therefore, the expected number of threads which maximizes throughput and minimizes latency can be expected to be achieved when memcached is provisioned with the same number of threads as hardware CPU cores which is also suggested by Leverich and Kozyrakis [7].

Utilizing results from the previous section, we configure the workload with 168 connections. At this workload, we have been able to achieve a throughput of 430k requests per second with latency meeting the QoS. Therefore, the configuration for the clients is as follows:

Utilizing findings from the previous section, a configuration with 84 connections can be used to generate a consistent load while the number of threads provisioned for memcached can be varied. Therefore, we can set up each benchmark client as follows:

```
memtier -s <server> -p 11120 -c 8 -t 3
  --random-data
  --key-minimum=1
  --key-maximum=10000000
  --data-size=32
```

The Memcached server configuration is setup with an increasing number of threads in each iteration of the benchmark. The used configuration is as follows:

```
memcached -d -p 11120 -m 6144 -t <thread_count>
```

2.2.1 Throughput & Latency

Figure 2.3 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.

Firstly, mean latency (*blue*) decreases as the number of threads increases from 1 to 6 where it reaches a minimum of 0.301ms. As the numbr of threads grows beyond 6, the mean latency increases at a slow pace.

Secondly, 99th percentile latency (*red*) starts at 2.05ms, above the required QoS. As the number of threads increases, the 99th percentile latency drops sharply. With 4 to 6 threads, the 99th percentile latency satisfies the QoS with a minimum of 0.84ms reached at 5 threads. Beyond 6 threads, the 99th percentile latency rises sharply beyond the required QoS.

Thirdly, the number of operations per second (*yellow*), increases linearly with the number of threads up until 6 threads where it reaches a maximum of 558k requests per

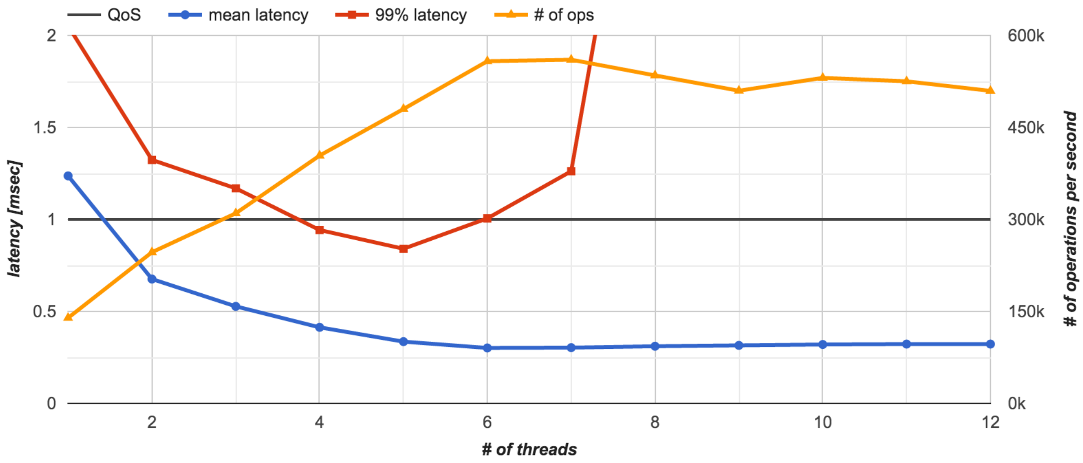

Figure 2.3: Memcached Threads: Latency & Throughput vs Number of Threads

second. A further increase in the number of threads results in a decrease of the number of operations per second.

Overall, QoS constraints are only achieved with 4 to 6 threads. We can attribute this effect to the multi threaded design of Memcached. With 3 threads or less, the load exerted by the clients cannot be processed quickly resulting in requests being queued up. This leads to an increased 99th percentile latency. With a sufficient number of threads to process the load effectively, a CPU core is able to service the requests sooner and therefore reduce the 99th percentile latency while also increasing the total number of operations.

With more than 6 threads, the kernel is forced to schedule multiple threads on the same core. This effect is described as ‘load imbalance’ by Leverich & Kozyrakis. With more than 1 Memcached thread executing on a single core, it is reasonable to expect the throughput to decrease, as opposed to only 1 Memcached thread per each core, due to context switching.

Overall, the best performance in terms of maximizing throughput and minimizing latency is provided by as many threads as CPU cores, in our case 6 threads. However, this is not in fact the minimum 99th percentile latency achieved in the benchmark - at 5 threads we obtain 99th percentile latency of 0.84ms with 480k requests per second.

2.2.2 CPU Utilization

Figure 2.4 provides the *mpstat* category breakdown of the CPU utilization of Memcached during the benchmark. Note that unattributed utilization accounts for idle time.

Firstly, Memcached CPU usage (*usr*) increases linearly with the number of instances up to 6 threads and accounts for 12.7%. With more than 6 threads, the Memcached CPU usage remains constant. This is reasonable as Memcached itself is not CPU intensive, however, a larger number of threads will require more CPU to operate effectively. Consequently, with more than 6 threads, context switching occurs due to

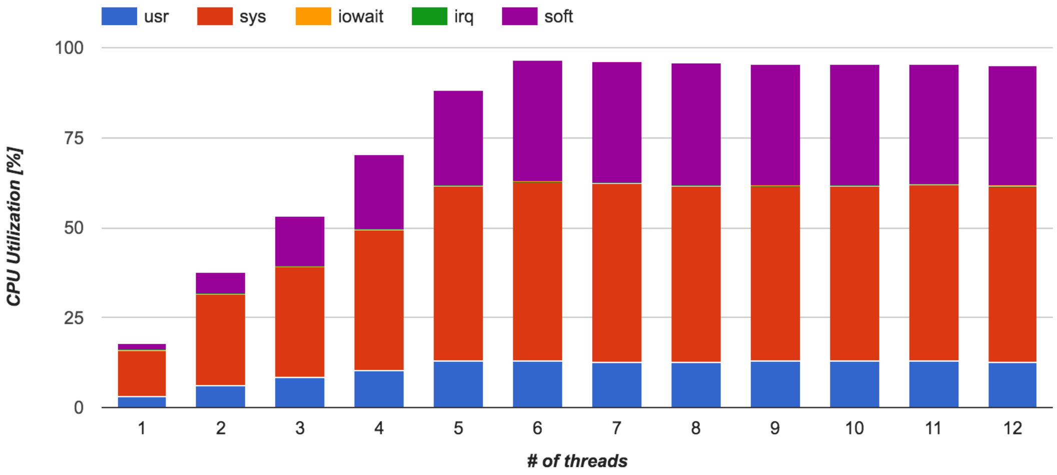

Figure 2.4: Memcached CPU Utilization with Multiple Threads

insufficient number of CPU cores to monopolize a given core and therefore the CPU usage remains constant.

Secondly, kernel utilization increases with the number of threads up to 6 threads at which point it stabilizes and accounts for 50%. A further increase in the number of threads does not result in increased utilization. It is reasonable to expect a large kernel CPU utilization as it is responsible for receiving and sending of network requests and memcached is a network bound application [2].

Thirdly, similarly to kernel CPU usage, CPU utilization for software interrupts (*soft*) increases with the number of threads. At 6 threads, it accounts for 33.7%. The increase in the time spent processing software interrupts is due to overall increase in throughput as requests are processed faster. An increase in the number of requests per second directly results in an increase in the number of software interrupts received and issued by *libevent*.

Finally, we observe no disk I/O wait as Memcached is an in memory object cache and therefore does not access the hard drive unless the memory is exhausted and swapping to disk is triggered. Additionally, hardware interrupt servicing *irq* only accounts for 0.01% as batching in the NIC is enabled.

2.2.3 Thread Evaluation

Increasing the number of Memcached threads results in increased throughput and decreased latency. The best performing configuration observed is with as many threads as there are CPU cores. The direct cost of increased number of threads is CPU utilization.

2.3 Thread pinning

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.” [7] and therefore the reasonable next step in optimizing memcached performance is to attempt thread pinning and analyse the results obtained.

By default, when a new process is started, its affinity is set to all available CPUs. We can discover the 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.” [4]. We can discover memcached threads used for request processing using the following command where *tid* is the thread id discovered previously [4].

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

For this benchmark, the previous configuration with 6 threads will be used.

2.3.1 Latency & Throughput vs Threads

Figure 2.5 presents a comparison of Memcached with pinned threads against unpinned threads.

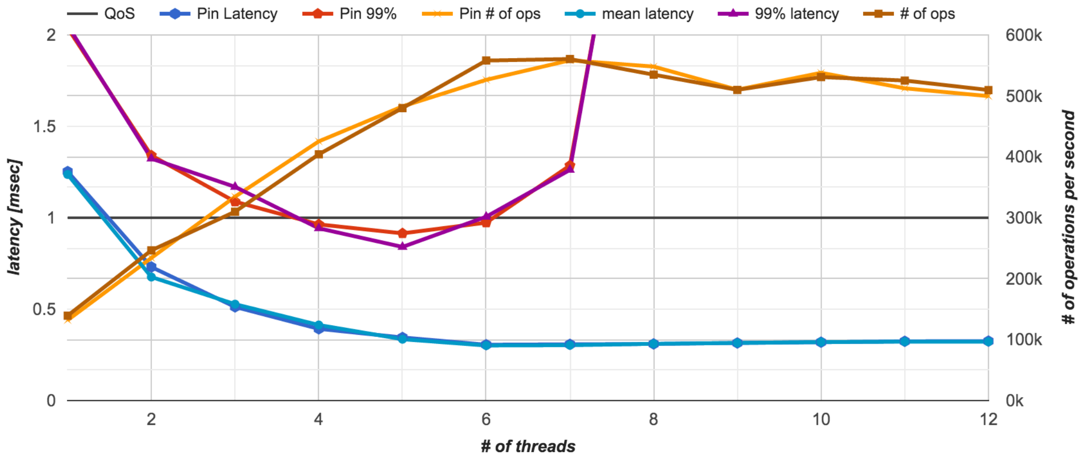

Figure 2.5: Memcached Latency & Throughput vs Threads: Comparison of pinned threads (labeled: *Pin*) vs unpinned threads

Overall, we can observe that there is very little change in all of the results observed. Mean latency remains the same, 99th percentile latency only differs slightly in it’s minimum value at 5 threads and the number of operations per second remains the same.

Overall, we find that in our benchmarks thread pinning does not affect performance significantly. This is contrary to results reported by Leverich and Kozyrakis [7]. According to their research, thread pinning results in improved load balance and a result decreases 99th percentile latency. This in turn results in increased throughput.

2.3.2 CPU Utilization

Figure 2.6 presents the CPU Usage with Memcached threads pinned.

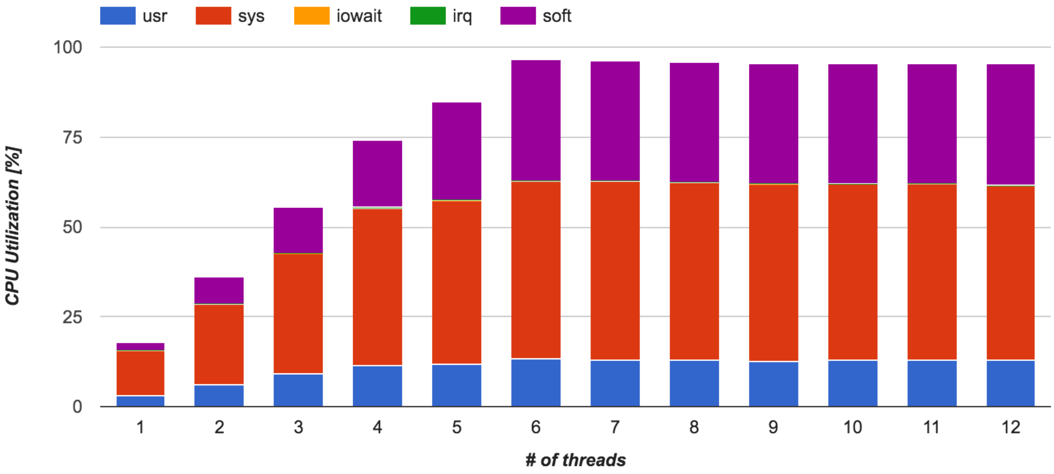

Figure 2.6: Pinned Memcached CPU Utilization

The CPU Utilization of pinned threads is nearly identical to unpinned threads presented in Figure 2.4.

2.3.3 Thread Pinning Conclusion

In our benchmarks, we have not been able to achieve the significant improvements in both latency and throughput suggested to be gained by thread pinning. This is likely due to differences in hardware as Leverich and Kozyrakis [7] perform benchmarks on a significantly more performant hardware as well as running on a slightly older version of memcached.

2.4 Group Size

Memcached provides a configuration option -*R* to set the group size used inside 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)” [5]. This in effect means the number of requests that will be processed from a single connection before memcached switches to a different connection to enforce a fairness policy.

In this benchmark, we consider the 6-threaded Memcached configuration without thread pinning with the addition of the *-R* configuration parameter to set the group size. Memcached implementation limits the minimum value of group size to be 20 while the maximum can be at most 320 (if set higher, memcached will override the setting) [3]. Therefore, we set up the benchmark to increase the group size by 20 in each consequtive iteration. The workload generated by the clients remains the same as in previous sections.

2.4.1 Latency & Throughput

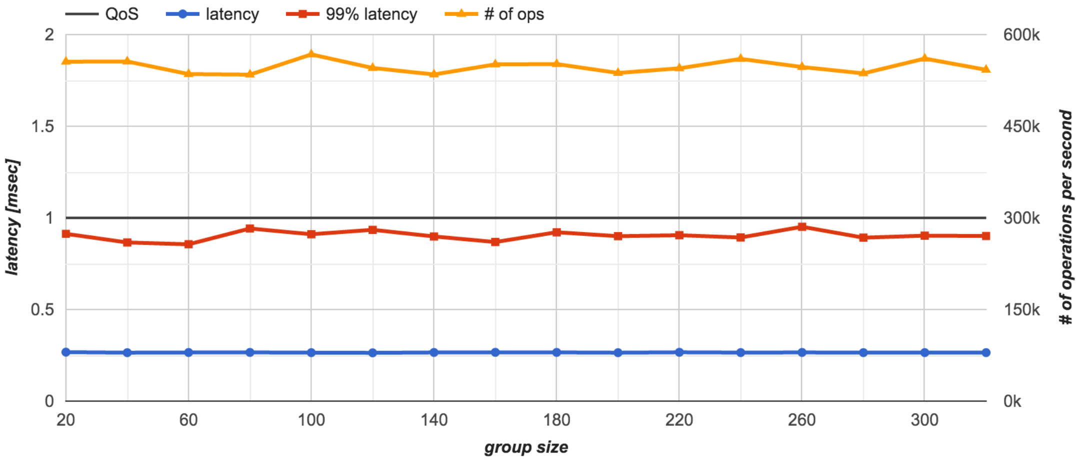

Figure 2.7: Latency & Throughput vs Memcached Group Size

Figure 2.7 plots the relationship between group size, latency and throughput.

Firstly, we can observe that mean latency remains unaffected as group size increases. Secondly, the total number of operations remains stable at an average of 550k requests per second. This corresponds to the same level of throughput as observed with the default group size of 20. Thirdly, the 99th percentile latency has decreased compared to the default at group size of 20. We have been able to reduce the 99th percentile latency to an average of 0.9ms by increasing the group size. However, there does not appear to be a strong direct correlation with a particular group size providing lower 99th percentile latency.

2.4.2 CPU Utilization

Figure 2.8 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.

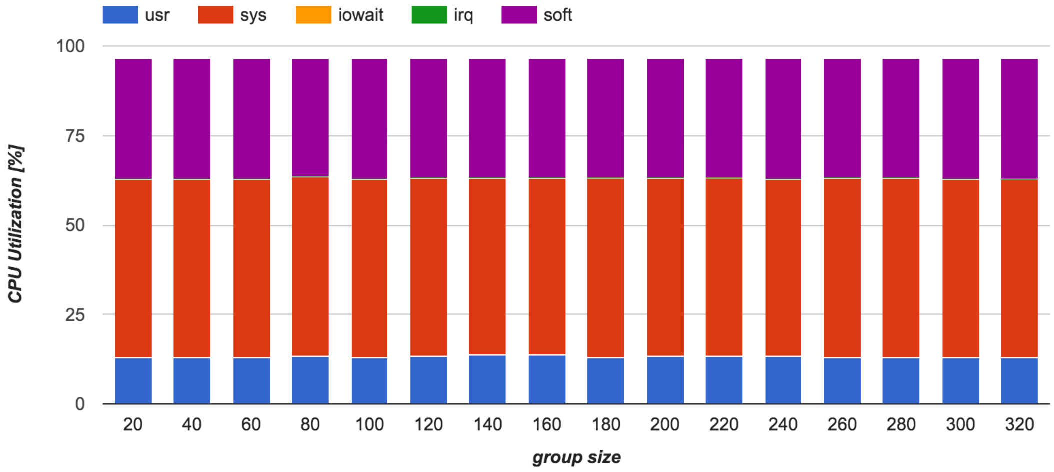

Figure 2.8: CPU Utilization vs Memcached Group Size

2.4.3 Group Size Conclusion

In our scenario, we have used 168 simultaneous connections from clients. In order to exploit the group size effectively, a small number of connections with very high number of requests per second may be required in order to effectively utilize the larger group size. With the hardware setup in this paper, we are unable to generate such a load and verify this claim. However, Blake and Saidi [3] have suggested that increasing the group size leads to increased throughput and decreased 99th percentile latency.

2.5 Receive & Transmit Queues fixing

TODO: Haven't been able to obtain any improvements in terms of performance (both throughput and latency), not sure if the topic should still be discussed in detail.

2.6 Multiple Memcached Processes

In this section, we will examine the impact multiple memcached processed have on the overall performance of the caches as a whole. In some applications, it is important to be able to partition the system in such a way systems interact with different instances of memcached. Additionally, this information will also serve as a useful benchmark comparison for Redis performance.

Chapter 3

Redis

In this chapter, Redis performance in terms of latency, throughput and system resource requirements is examined. Initially, we will focus on performance under the default configuration of Redis. Subsequently, the scalability characteristics of Redis are explored. Focus is given to the impact of multiple Redis instances running simultaneously on the same machine under various levels of workload.

Unless otherwise stated, all benchmarks are performed to target the required Quality of Service (QoS) of achieving 99th percentile latency under 1 millisecond.

3.1 Out of the Box Performance

Firstly, in order to understand the baseline performance of Redis we consider the default configuration of Redis. A Redis deployment can be started with the following command:

```
redis redis.conf --port 11120
```

By default, a Redis deployment comes with a default configuration file `redis.conf`[11]. Any options specified in the configuration file can be overridden from the command line by prefixing them with `--`, in our case we are overriding the port number and setting it to 11120. All other configuration options remain unmodified.

In order to understand the default Redis performance, we design the benchmark to exert an increasing level of load on the Redis server. Initially, we start with 2 threads and 1 connection per each thread on all workload generating clients and increase the number of connections per thread linearly. The workload generating clients are executed with the following command:

```
memtier -s nsl200 -p 11120
    -c <connection_count> -t 2
    -P redis
    --random-data
    --key-minimum=1 --key-maximum=10000000
```

```
--data-size=32
```

3.1.1 Latency vs Throughput

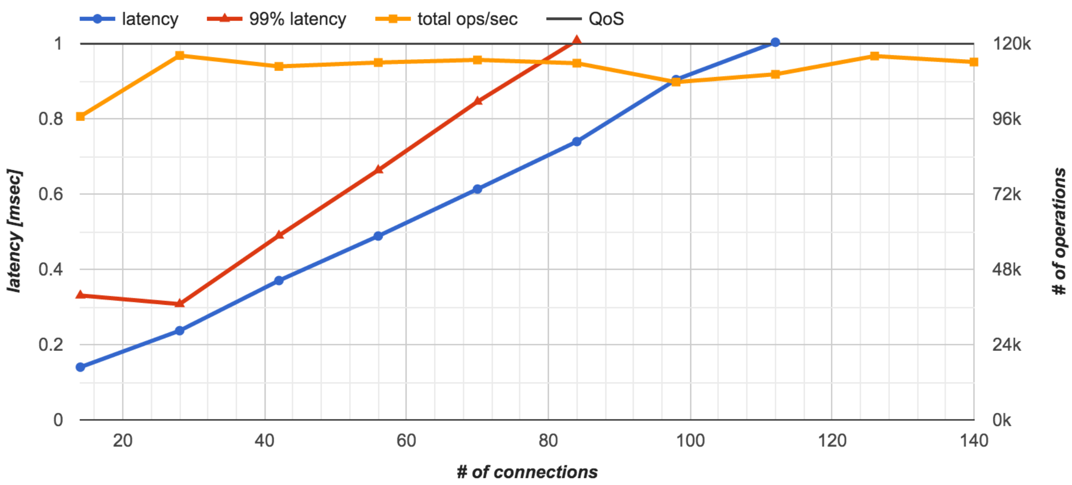

Figure 3.1: Redis: Latency & Throughput vs Number of Connections

Figure 3.1 plots the relationship between mean latency and the 99th percentile latency on the left vertical axis, the number of operations per second on the right vertical axis and the number of client connections on the horizontal axis. Each connection contributes to the overall load of the cache equally, therefore, a larger number of connections results in increased load exerted by the clients. The graph has been trimmed to show only data which satisfies the QoS requirements.

Firstly, mean latency (*blue*) increases linearly with the number of connections. This is a reasonable result as a linear increase in the number of requests sent by the clients should result in linear increase in mean latency.

Secondly, the 99th percentile latency (*red*) increases as the number of connections increases with the exception of 28 connections. The 99th percentile latency increases faster than the mean latency does, however, this is also reasonable as increased load will result in queuing delay incurred on the server before Redis is able to process the request, driving the 99th percentile latency up. The QoS requirements are only satisfied up to 84 simultaneous connections.

Thirdly, the number of operations (*yellow*) increases between 14 and 28 requests and reaches a maximum at 28 connections. Beyond 28 connections, the number of operations remains stable around 116k requests per second.

We can observe that the number of operations Redis processes increases steadily until it reaches 119k requests per second at which point a further increase in throughput comes at a disproportionately grater cost in both mean and 99th percentile latency. The peak throughput observed under the QoS requirements is around 123k requests per second.

The performance degrades significantly with an increased workload beyond the peak shown at 123k requests per second. The throughput remains the same while the mean and the 99th percentile latency spikes heavily.

3.1.2 CPU Utilization

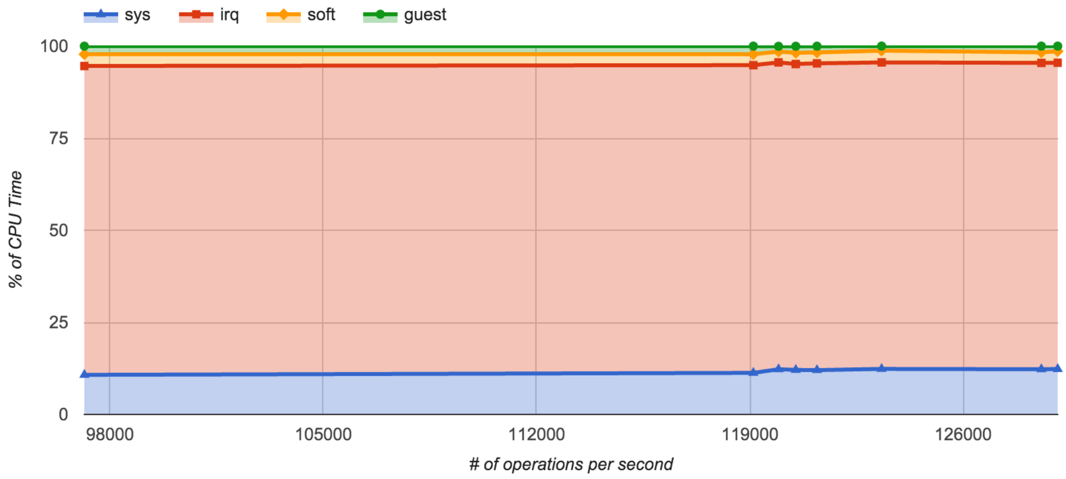

Figure 3.2: Redis: CPU Utilization

Figure 3.2 outlines the CPU utilization in terms of time spent processing system calls (sys), servicing hardware interrupts (irq), handling software interrupts (soft) and processing related to Redis (guest).

We can observe that servicing hardware interrupts consumes the majority of the processing time - 83 percent. This is the result of receiving and dispatching a large number of requests through the NIC. The amount of CPU time devoted to processing system calls is about 10 percent while processing software interrupts accounts for 3 percent. Redis itself requires only about 2 percent of the overall CPU time.

Redis appears to be network constrained rather than CPU constrained in the workload examined above. This is reasonable as Redis is executing within one main event loop therefore it does not require locking and does not generate heavy load on the CPU when looking values up in the cache.

3.2 Multiple Redis Instances

As seen in section 3.1, Redis cannot increase overall throughput of the system with only a single instance as only a single core is capable of processing requests and becomes the bottleneck. An immediate solution to this problem is to provision a larger number of instances on a multi-core system in order to better utilize the hardware resources.

In this benchmark, we examine the effects increased number of Redis instances with respect to latency, 99th percentile latency and overall throughput of the system. Additionally, we examine the effect of multiple instances on the CPU usage.

Firstly, multiple instances of Redis can be spawned easily on the server by binding them to distinct port numbers. Out of the box, Redis does not provide the capability to proxy multiple instances of Redis through a single port in order to load balance the instances. There is the option to configure a Redis cluster, however, the intended use case is primarily for resiliency and fail over. In this benchmark, we consider a simpler scenario where each instance is isolated from each other and acts as an independent cache. This is a simplification of a real world scenario, however, a large deployment of Redis could be designed to partition the key space and utilize multiple independent instances similarly. The Redis application can be spawned with the following script:

```
for i in [1..n]
    redis-server redis.conf --port (11120 + i) --maxmemory (6 / i)gb
```

Note that we are explicitly specifying the maximum amount of memory each instance will be allocated. In our case, we partition 6 GB of memory space evenly between the individual instances.

Secondly, in order to obtain comparable results, the load exerted on the Redis cache must remain constant. The load itself, however, needs to be partitioned across all of the instances of Redis evenly. In order to achieve this, each workload generating client spawns `i` instances of the benchmark and targets its respective Redis instance. We use the following script to start the workload generating clients:

```
for i in [1..n]
    memtier -s nsl200 -p <port>
        -c round(16 / i)
        -t 1
        -P redis
        --random-data --data-size=64
        --key-minimum=1 --key-maximum=round(100000000 / i)
        --test-time=400
```

Initially, we start with 16 connections and 1 thread. As we increase the number of instances, the number of connections goes down, however, a larger number of instances are deployed. Note that we are using a `round` function to ensure that the number of connections as well as the maximum key are integers. In order to smooth out load variance caused by integer divisibility, we consider two cases. One in which the `round` function is defined as the `ceiling` function and the other when it is defined as the `floor` function. The results of both types of the `round` function are then averaged. If a higher number of workload generating clients were available for the experimentation, the round approach would not be required.

Furthermore, the generated dataset is 6.4GB (100 million keys * 64 bytes of data). This is by design and leads to evictions in the cache as the size approaches the maximum.

3.2.1 Latency and Throughput

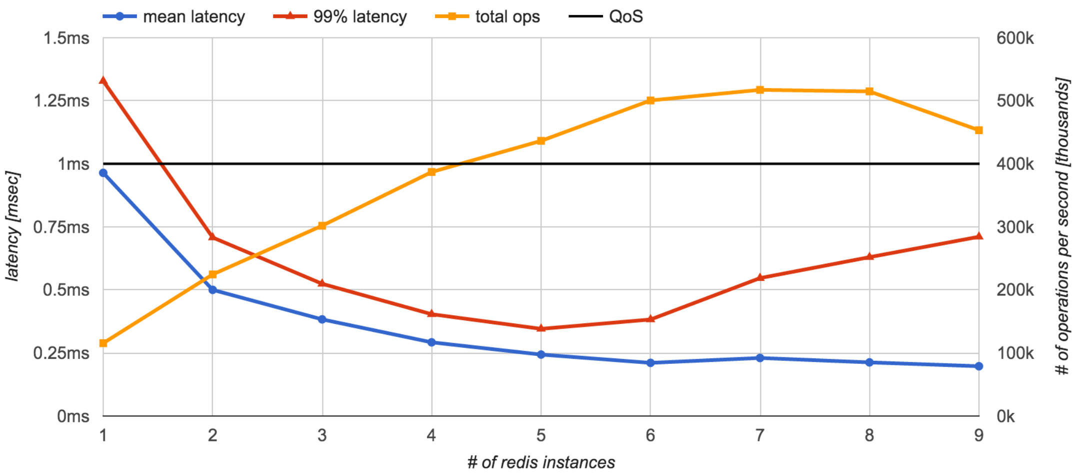

Figure 3.3: Redis Instances: Latency, Throughput vs Number of instances

Figure 3.3 plots the relationship between the number of Redis instances running simultaneously on the cache server against the mean and 99th percentile latency on the left vertical axis and the total number of operations per second on the right axis. The black line positioned at 1 ms outlines the QoS target of the benchmark.

The mean latency (blue) decreases steadily as the number of Redis instances increases. At five Redis instances, the mean latency flattens out and remains stable as the number of instances is increased further.

The 99th percentile latency (red), on the other hand, experiences a sharp decrease as the number of instances is increased from 1 to 2 which brings the 99th percentile latency within the desired QoS constraints. As the number of instances increases further, the 99th percentile latency continues to decrease steadily. The minimum is reached at 5 instances with latency of 0.346 ms. A further increase in the number of instances results in higher 99th percentile latency.

The total number of operations increases steadily as the number of instances increases, peaking at 517,000 requests per second with 7 and 8 instances. A further increase in the number of instances results in a decrease in the total number of operations.

The QoS target is achieved in all but the 1 instance scenario. This is due to significantly larger load exerted on a single instance which in the subsequent multi-instance benchmarks gets distributed across the larger number of instances and therefore satisfies the QoS constraints.

Additionally, Redis manages to scale quite well as the number of instances increases up to 6. At this point, there are as many instances as there are CPU cores on the Redis server. Consequently, at 6 instances we achieve near minimum 99th percentile latency while also achieving near maximum throughput.

Furthermore, at 6 instances the 99th percentile latency is at around one third of the accepted QoS constraint. In real world deployment, it should be possible to utilize this gap to increase the throughput further. In our benchmarks, we have not been able to exert a load which would both increase throughput as well as increase the 99th percentile underutilization. This is due to a limited number of client benchmarking hosts to effectively tune the load to the required level.

3.2.2 CPU Utilization

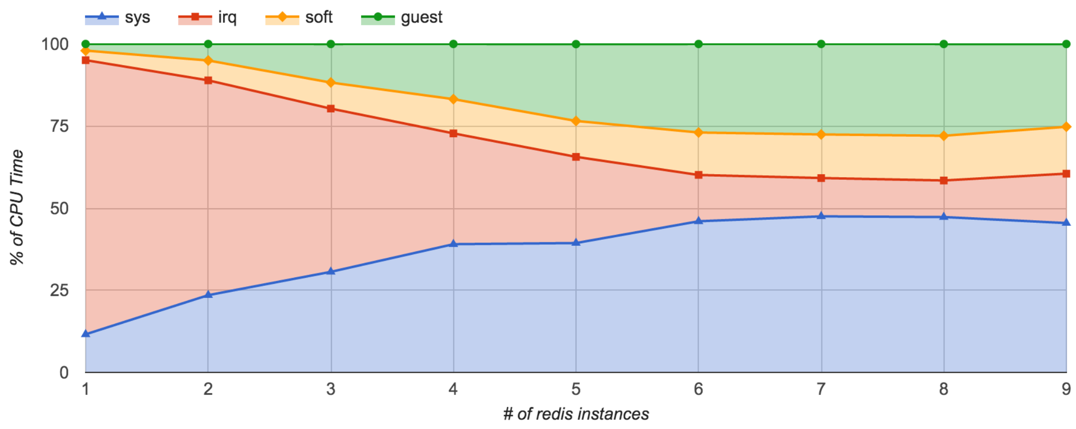

Figure 3.4: Redis Instances: CPU Utilization vs Number of Instances

Figure 3.4 presents the relationship between the number of instances and the CPU utilization.

Firstly, the CPU usage of Redis (green), *guest*, increases as the number of instances provisioned increases steadily. At 6 instances and more, the overall CPU usage remains fairly constant accounting for 27 percent of the total CPU utilization. This is due to the server having 6 CPU cores, requiring context switching between applications rather than parallel processing.

Secondly, the CPU dedicated to software interrupts (yellow), *soft*, follows a similar pattern to Redis. As the number of instances increases, so does the CPU usage dedicated to software interrupts. At 6 instances or more, the CPU utilization remains stable, accounting for 10 percent of the total usage. Software interrupts are inherently correlated to the software applications running on the server and therefore it is reasonable to observe a similar pattern between software interrupts and the application itself.

Thirdly, the CPU usage required by the operating system (blue), *sys*, grows steadily as the number of Redis instances increases. At 6 instances or more, the operating system CPU usage peaks and remains stable at 48 percent.

Finally, the portion of the CPU time dedicated to processing hardware interrupts (red), *irq*, decreases steadily as the number of Redis instances increases. This is due to batching of requests at the network interface card. At 6 or more instances, hardware interrupt servicing takes up about 11 percent of the total CPU usage.

Despite the increased resource requirements by Redis, it still only accounts for about 27 percent CPU usage at 6 processes with majority of the CPU time spent in the operating system and or networking. The observation from the single instance benchmark in the previous section therefore extends to a multi instance setup. Redis appears to be network intensive rather than CPU intensive.

By spawning multiple instances of Redis, we have been able to significantly increase the overall performance of the server cache. Running multiple simultaneous instances, however, requires the key space to be partitioned.

3.3 Pinned Redis Instances

In the previous section we have observed that the performance of a Redis server can be greatly improved by provisioning multiple Redis instances simultaneously. Pinning processes to distinct cores is suggested to improve tail latency [7]. In this section, we examine the effect process pinning has on the performance of Redis. We consider exactly the same workload as in the previous section 3.2 as well as exactly the same server setup with the exception of pinning the Redis processes. That is, the workload is kept constant while it is partitioned across multiple instances.

A Redis process can be pinned to a unique core through the use of the `taskset` utility as follows:

```
taskset -pc <redis_pid> <core_id>
```

The Redis processes identified as `redis pid` is pinned to the CPU core identified by `core id`. We can identify the process id of a Redis application through the `ps` command. When running more Redis applications than there are CPUs, we assign it to the *n*th index of the application modulo the total number of cores, which is 6.

3.3.1 Pinned Latency and Throughput

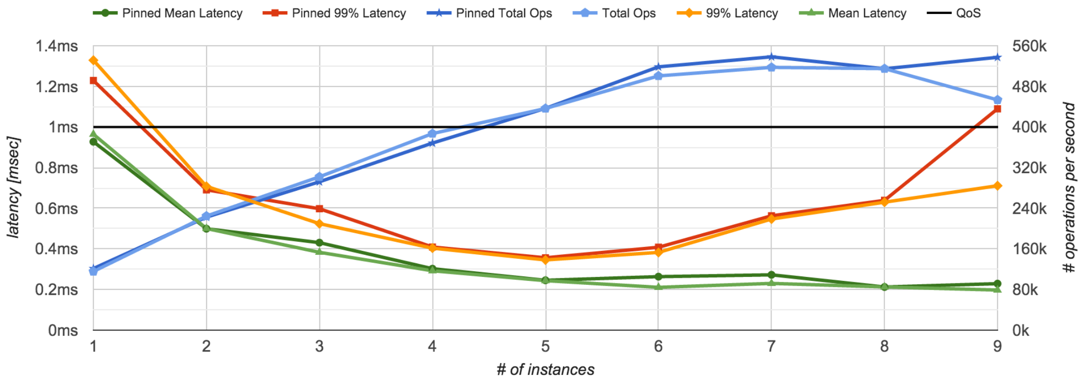

Figure 3.5: Redis Instance Pinning: Instances vs Latency and Throughput

Figure 3.5 plots the relationship between the number of instances on the horizontal axis, latency on the left vertical and throughput on the right vertical. Additionally, the performance obtained without pinning are plotted alongside the pinned results.

We can observe that pinning Redis processes results which strongly correlate to the results of the unpinned benchmark. Across mean latency, 99th percentile latency and throughput, there is very little variance in the performance observed.

3.3.2 Redis Persistence

TODO

3.4 Object Size

In this section, the impact of the size of the object stored in the cache is investigated. Redis imposes no restrictions on the size of objects stored in the cache.

In order to investigate the impact of object size on the cache, we consider a benchmark with an increasing object size. The object size is increased in powers of two starting at 2 bytes and ranging to 512 KB. This allows us to capture the majority of important sizes commonly used when designing applications.

The server configuration remains the same as the current best found configuration, the multi-instance configuration with 6 instances. The clients are configured as follows:

```
for i in [1..19]
    memtier -s nsl200 -p <port>
        -c 3
        -t 1
        -P redis
        --random-data --data-size=pow(2, i)
        --key-minimum=1 --key-maximum=(1066666666 / pow(2, i))
        --test-time=400
```

We run 19 iterations of the benchmark since 512 KB is equivalent to 2 to the power of 19 bytes. The `data-size` is configured to be increasing in powers of 2. The key range is defined as 6.4 GB split across 6 instances and further accounts for the increased size. Table 3.1 outlines the configuration options for each iteration.

3.4.1 Latency and Throughput

Figure 3.6 displays the relationship between object size on the horizontal logarithmic axis, latency on the left vertical axis and throughput on the right vertical axis.

Firstly, as object size increases up to 512 bytes, the mean latency remains stable at 0.55 ms. Beyond 512 bytes, the mean latency starts to increase and climbs beyond the

Iteration	Data Size (bytes)	Key Maximum	Total Size (GB)
1	2	53333333	6.4
2	4	26666667	6.4
3	8	13333334	6.4
4	16	6666667	6.4
5	32	3333334	6.4
6	64	1666667	6.4
7	128	833334	6.4
8	256	416667	6.4
9	512	208334	6.4
10	1024	104167	6.4
11	2048	52084	6.4
12	4096	26042	6.4
13	8192	13021	6.4
14	16384	6511	6.4
15	32768	3256	6.4
16	65536	1628	6.4
17	131072	814	6.4
18	262144	407	6.4
19	524288	204	6.4

Table 3.1: Redis Object Size - Data Size and Maximum Key for each iteration. Total size is calculated as the product of `Data Size`, `Key Maximum` and `6` instances.

desired QoS constraint at object size of 4 KB. A further increase in object size leads an disproportionately greater increase in mean latency.

Secondly, the 99th percentile follows the same pattern as the mean latency, however, it begins to climb over the desired QoS sooner, at object size of 1 KB.

Thirdly, the number of operations per second remains constant for object sizes under 256 bytes. An increase in object size decreases the number of operations per second. This is a reasonable result as an increase in the object size leads to higher bandwidth requirements and therefore leads to a lower number of operations per second.

Overall, Redis appears to be capable to scale well with object sizes up to 512 bytes. Larger object sizes put additional strain on the cache and require buffering which leads to increased latency of the average, and therefore 99th percentile, request. Primarily, Redis is not designed to store large (1KB+) values. It is, however, possible to partition large values into smaller ones and perform assembly/disassembly of the value on the client side.

3.4.2 CPU Utilization

Figure 3.7 displays the relationship between object size and CPU usage on the Redis server.

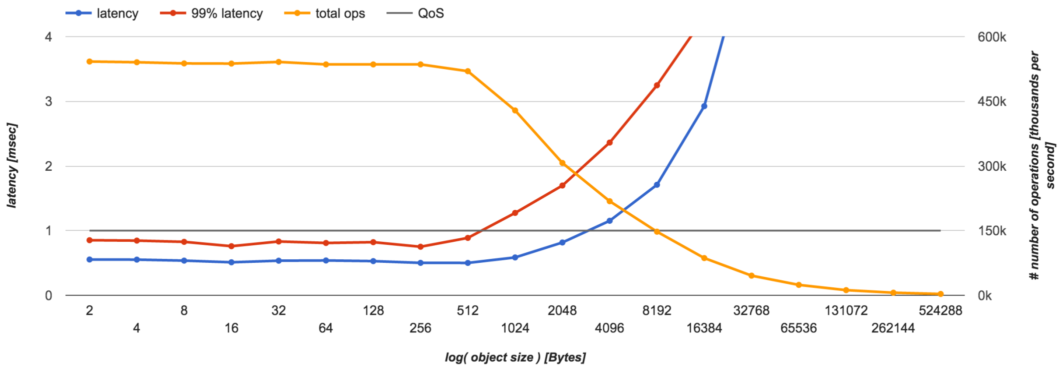

Figure 3.6: Redis Object Size: Latency and Throughput

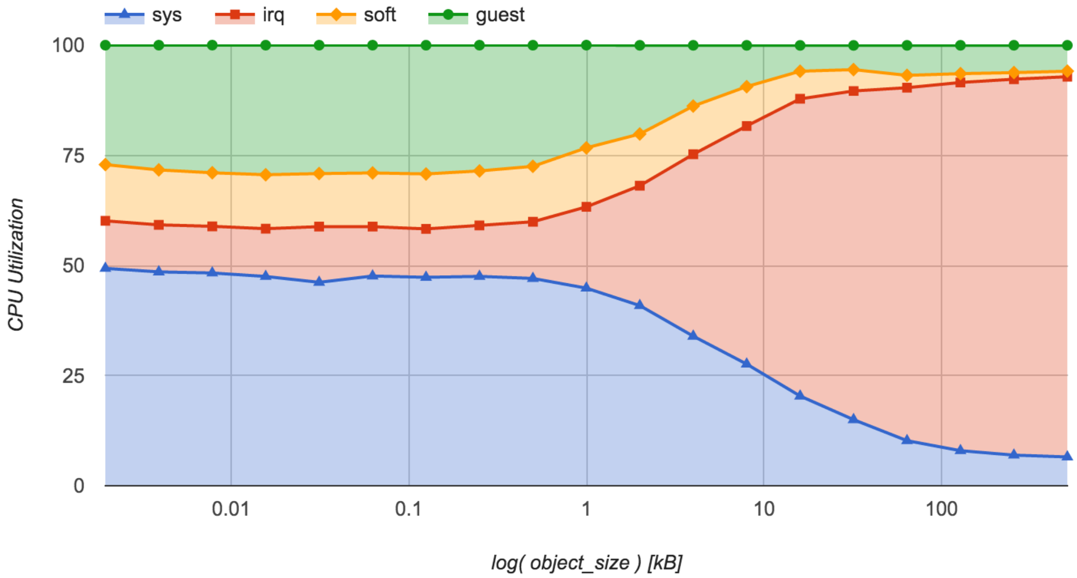

Figure 3.7: Redis Object Size: CPU Usage

Firstly, as object size increases up to 1 KB, the operating system (sys) requires about 47 percent of the total time to process the incoming requests and dispatch them to the relevant applications. As object size increases further, the time required decreases as the number of operations decreases.

Secondly, the Redis applications (guest) require 27 percent of the total time when processing requests under 1 KB, with requests larger the direct cost of running Redis decreases as there are less requests to process. A similar pattern holds for the software interrupts (soft), as there are less requests coming.

Finally, the time allocated to servicing hardware interrupts (irq) remains at 12 percent below 1KB, with object size increases beyond 1 KB, there is significant increase in the time required to service hardware interrupts. This is due to buffering of large objects and is effectively the cause of high mean and 99th percentile latency as well as low throughput.

Overall, Redis is designed to work well with objects sizes below 1 KB. As the object size increases, the cache experiences a degraded performance due to buffering of network input and output.

3.5 Key Distributions

TODO

3.5.1 Gaussian distribution

TODO

3.5.2 Zipf distribution

TODO

Chapter 4

Redis & Memcached: Head to Tail

Evaluation goes here

Chapter 5

Conclusion

Bibliography

[1] 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.

[2] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

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

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

[5] Danga Interactive. Memcached. `http://memcached.org`.

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

[7] 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.

[8] 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.

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

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

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

[12] 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.

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