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

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 [16]

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 [13].

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 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.

2.1 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 2.1 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 2.1, 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 2.1: 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 2.2 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 2.2: 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.

2.1.1 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 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.

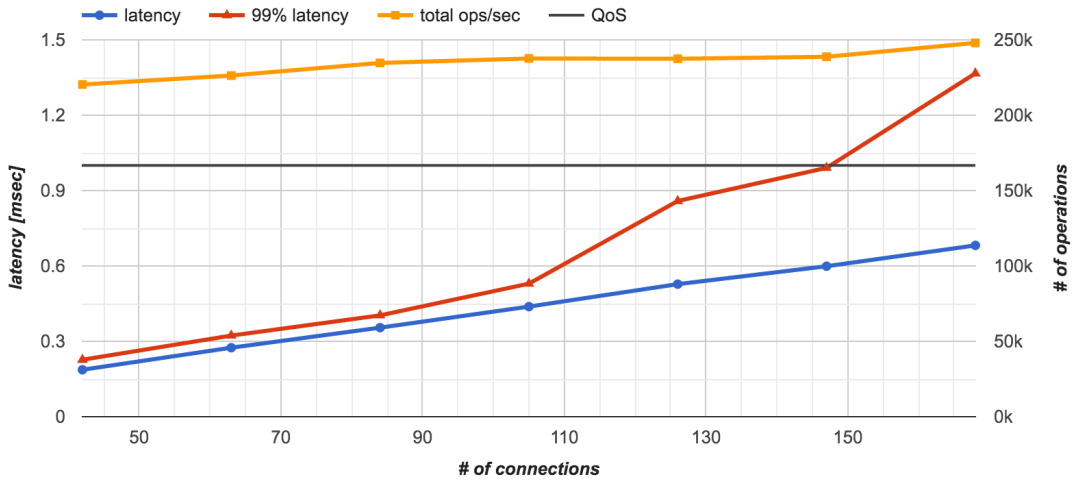

Figure 2.1: Latency & Throughput vs Number of Connections

As the number of 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.

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.

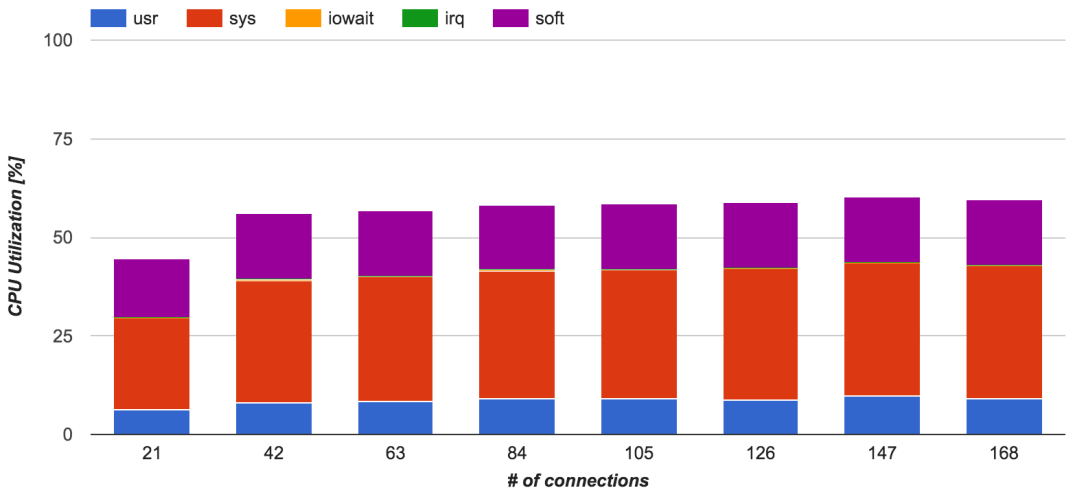

Figure 2.2: CPU Utilization for Out of the Box Configuration of Memcached

Figure 2.2 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 [8].

2.1.3 Evaluation

Given the trends presented in Figures 2.1 and 2.2, 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.

2.2 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 [7]. 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 2.3.

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 2.3: Memtier Configuration Options

Memcached, on the other hand, is configured to increase the number of threads in each consecutive benchmark. Table 2.4 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 2.4: Memcached Threads Configuration Options

2.2.1 Throughput & Latency

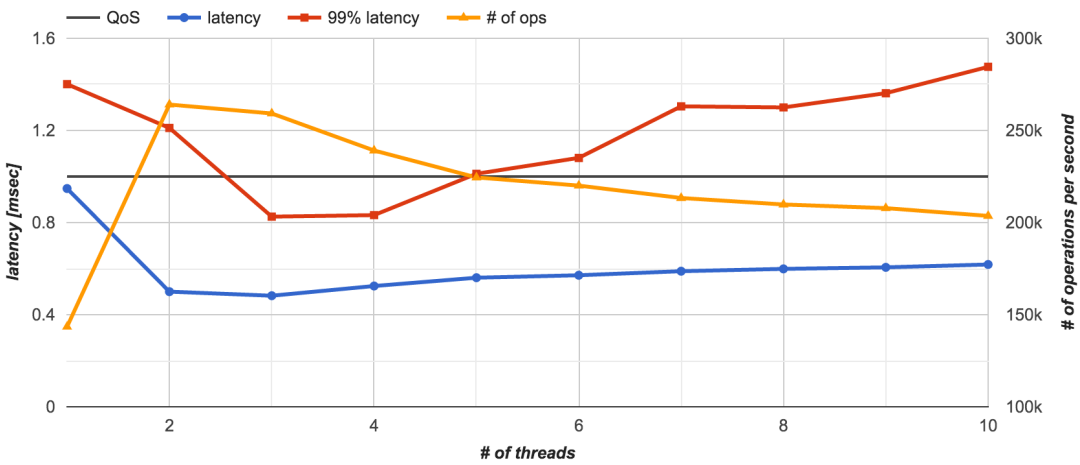

Figure 2.3: Memcached Threads: Latency & Throughput vs Number of Threads

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.

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.

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.

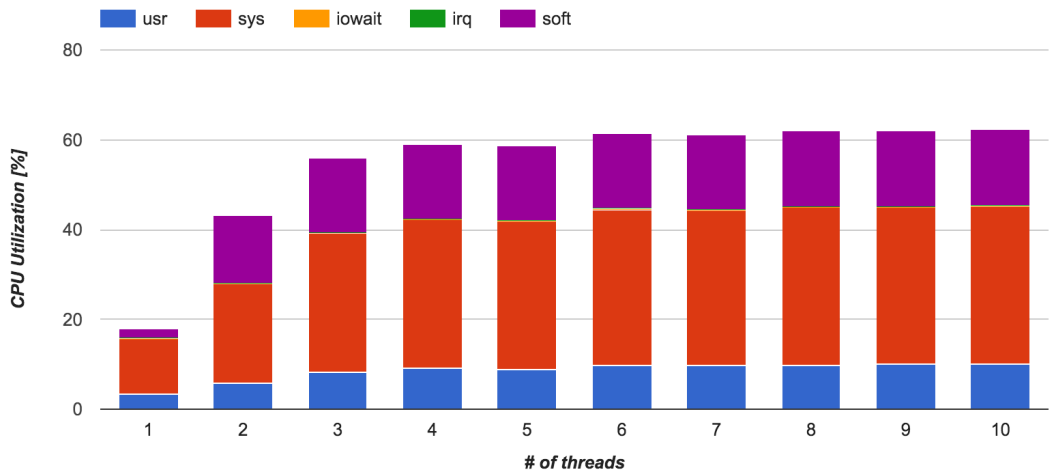

Figure 2.4: Memcached CPU Utilization with Multiple Threads

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.

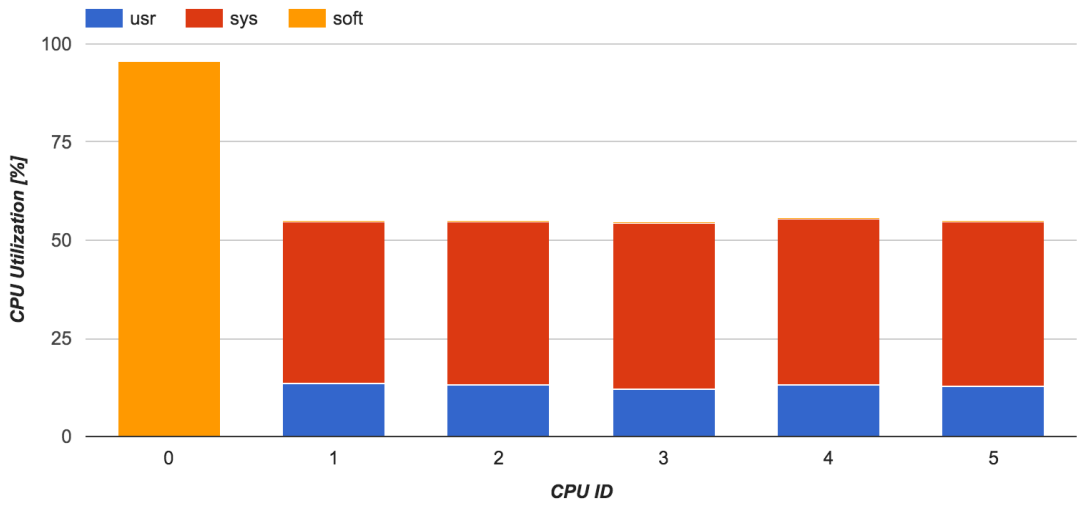

Figure 2.5: Memcached CPU Utilization with 6 threads by Individual CPU

Figure 2.5 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” [7]. 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.

2.2.3 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.

2.3 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 [17]. When a request is inserted into the epoll socket, a level triggered software interrupt is fired [2]. Each Memcached thread is awaiting the software interrupt with *epoll wait()* [15] 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 2.2 as well as increasing the number of threads linearly.

2.3.1 Threads, Latency & Throughput with IRQ pinned

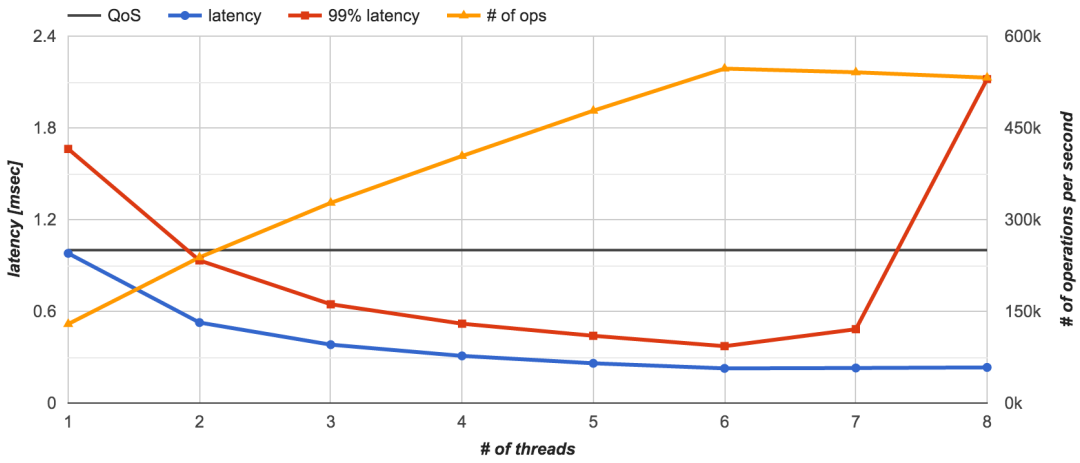

Figure 2.6: Memcached Latency & Throughput vs Threads with IRQ Affinity pinned to distinct cores

Figure 2.6 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.

2.3.2 CPU Utilization with IRQ pinned

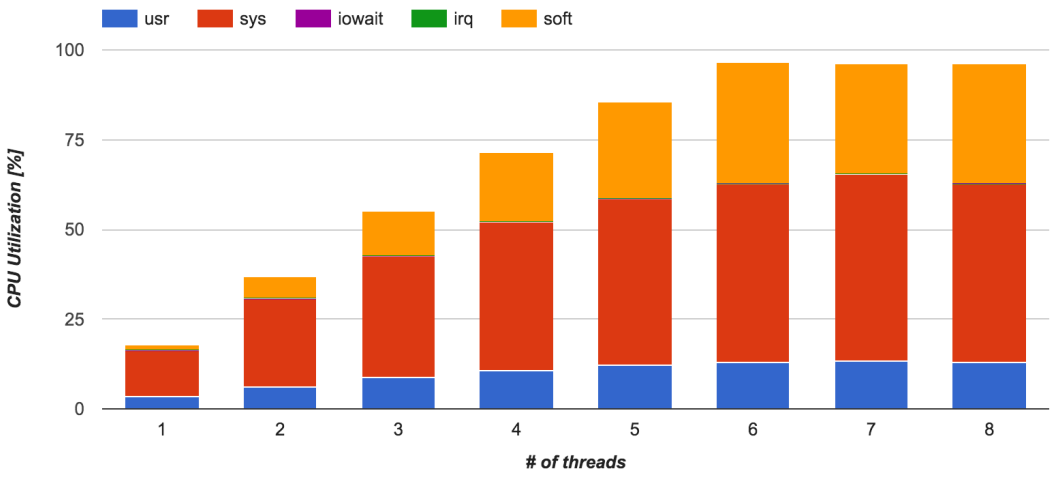

Figure 2.7: Memcached CPU Utilization with IRQ Affinity pinned

Figure 2.7 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.

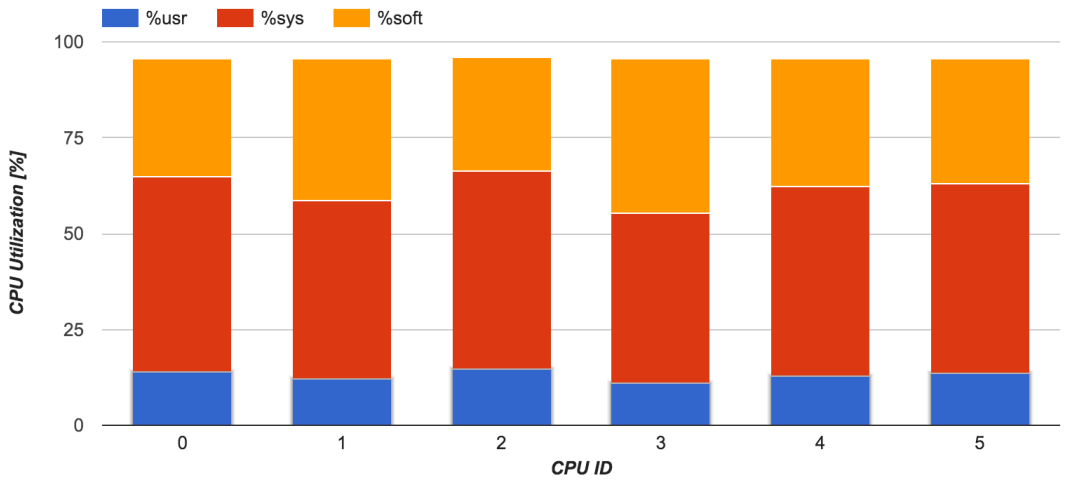

Figure 2.8: Memcached with 6 threads - Individual CPU utilization

Let us consider individual CPU utilization breakdown with 6 threads. Figure 2.8 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 [4] which can cause interference in the individual load. This topic will be further examined in the following section.

2.3.3 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.

2.4 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." [7] 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." [4] 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 [14]. 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.” [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, we use the same configuration as in previous sections. Tables 2.3 and 2.4 provides the complete configuration.

2.4.1 Latency & Throughput vs Threads

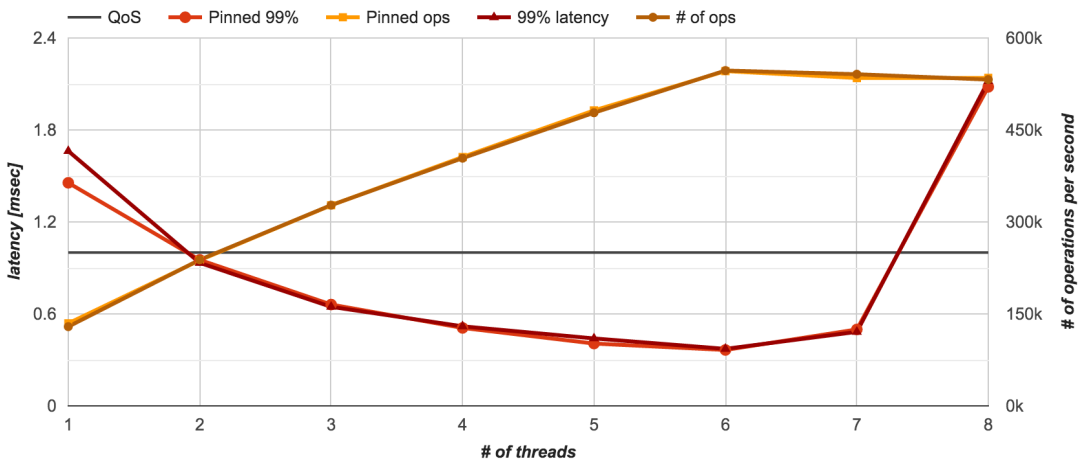

Figure 2.9: Memcached Latency & Throughput vs Threads: Comparison of pinned threads (labeled: *Pinned*) vs unpinned threads

Figure 2.9 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 [4].

2.4.2 CPU Utilization

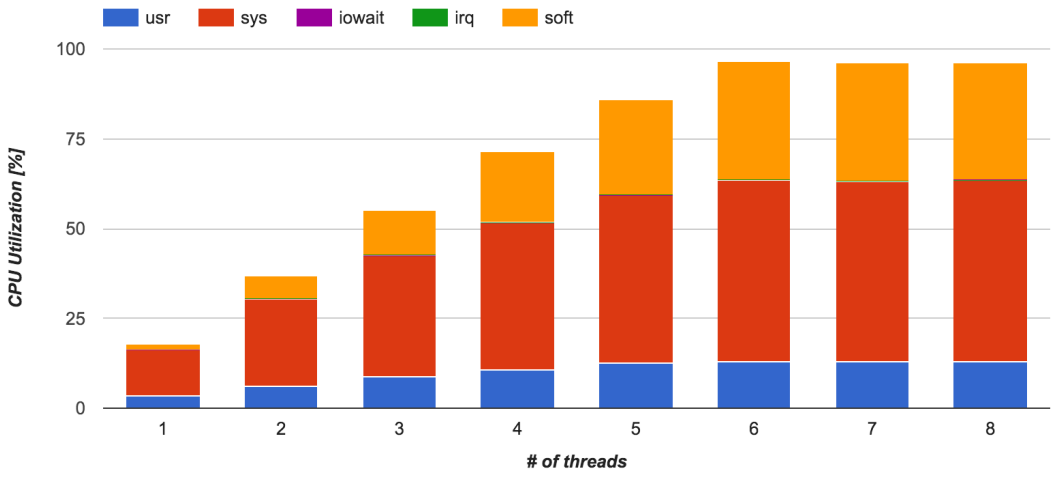

Figure 2.10: Pinned Memcached CPU Utilization

Figure 2.10 presents the overall CPU Usage with Memcached threads pinned. The CPU Utilization of pinned threads is nearly identical to unpinned threads presented in Figure 2.4.

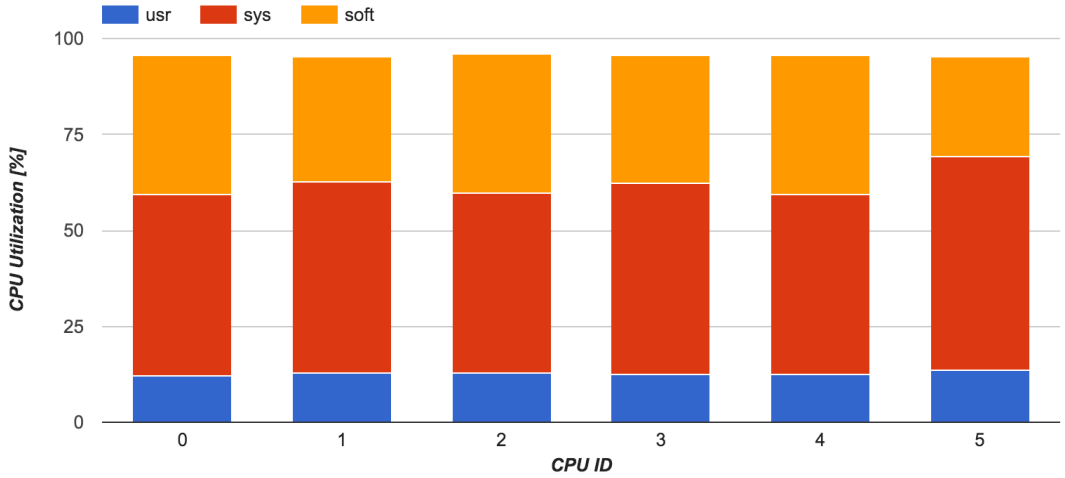

Figure 2.11: Pinned Memcached Individual CPU Utilization with 6 threads

Figure 2.11 plots the individual CPU usage breakdown. With thread pinning, we can observe a more evenly balanced distribution of CPU utilization as opposed to Figure 2.8.

2.4.3 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 [7]. 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.

2.5 Group Size

TODO 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.

2.5.1 Latency & Throughput

Figure 2.12 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.

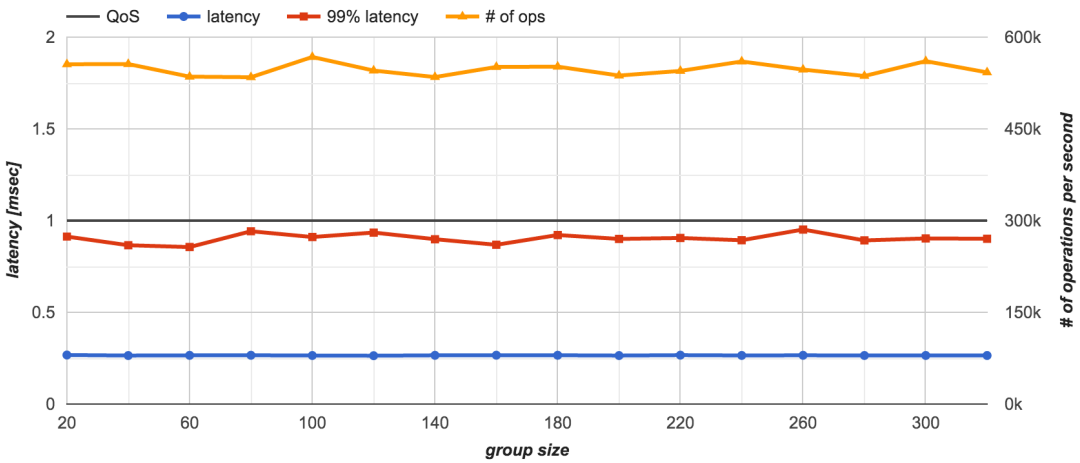

Figure 2.12: Latency & Throughput vs Memcached Group Size

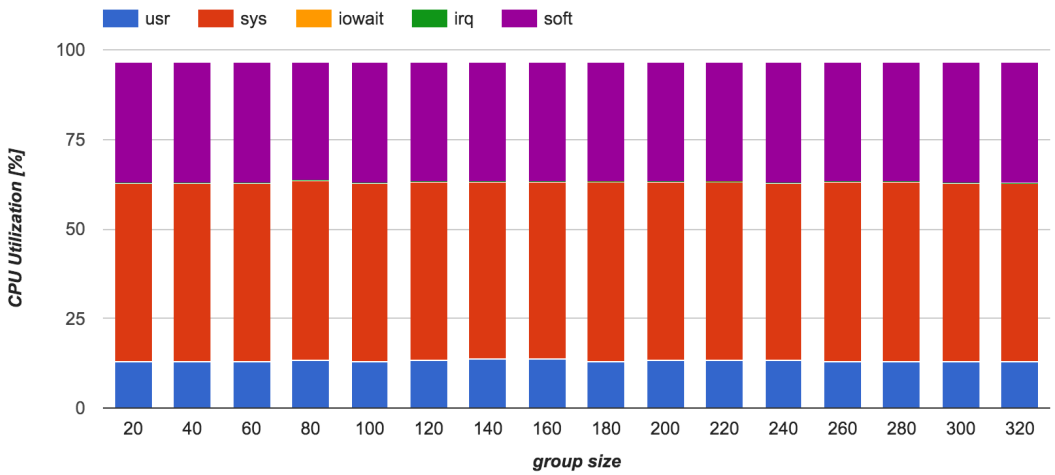

Figure 2.13: CPU Utilization vs Memcached Group Size

2.5.2 CPU Utilization

Figure 2.13 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.

2.5.3 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.

2.6 Multiple Memcached Processes

TODO 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

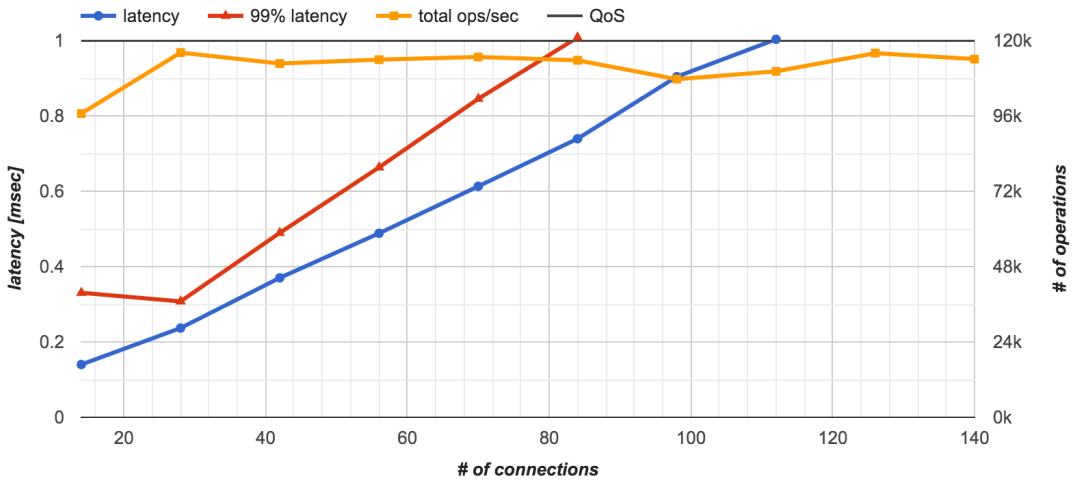

Figure 3.1: Redis: Latency & Throughput vs Number of Connections

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. At this point, we have reached the maximum a single Redis instance is able to process per second. These results are similar to results reported on Redis.io documentation [12] when scaled down to CPU speed equivalent our setup.

3.1.2 CPU Utilization

TODO

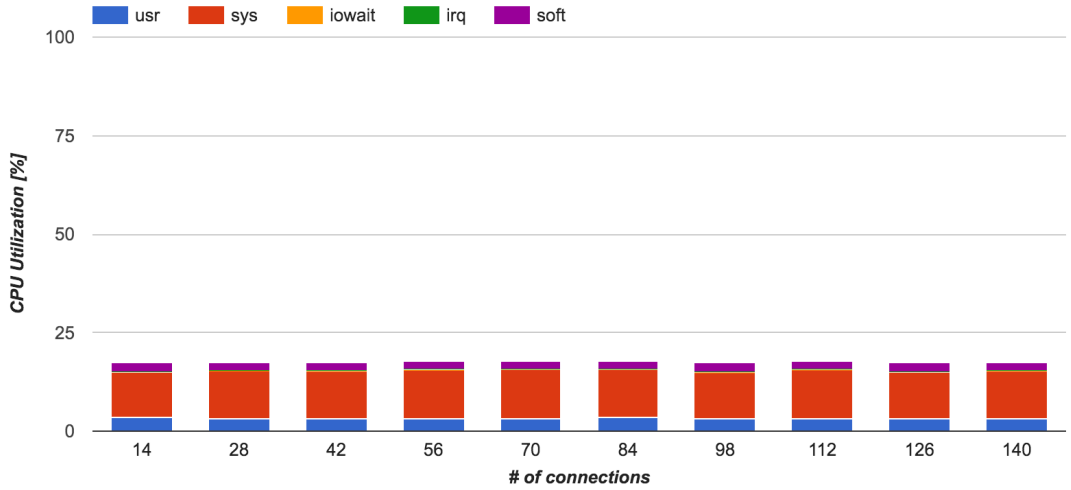

Figure 3.2: Redis: CPU Utilization

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. Table 3.1 outlines the configuration in terms of the number of connections. Overall, we aim to find a configuration such that we use 24 connections per each client host.

Instances	Threads	Avg # of Connections	Connections Total	Percentage
1	3	8	24	100%
2	3	4	24	100%
3	2	4	24	100%
4	2	3	24	100%
5	1	5	25	104.16%
6	1	4	24	100%
7	1	3.5	24.5	102.08%
8	1	3	24	100%

Table 3.1: Redis Multiple Instances - Number of Threads & Connections per each workload generating host. The percentage outlines how close the configuration for a given number of instances is to the target of 24 connections.

Having defined the configuration in Table 3.1, we use the following script to start the workload generating clients:

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

Initially, we start with 24 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, in the case when the number of connections is not an integer, 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 rounding 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. Note that the load exerted may initially exceed the QoS constraints, however, as the load gets partitioned across more instances, the QoS constraint will be satisfied.

3.2.1 Latency and Throughput

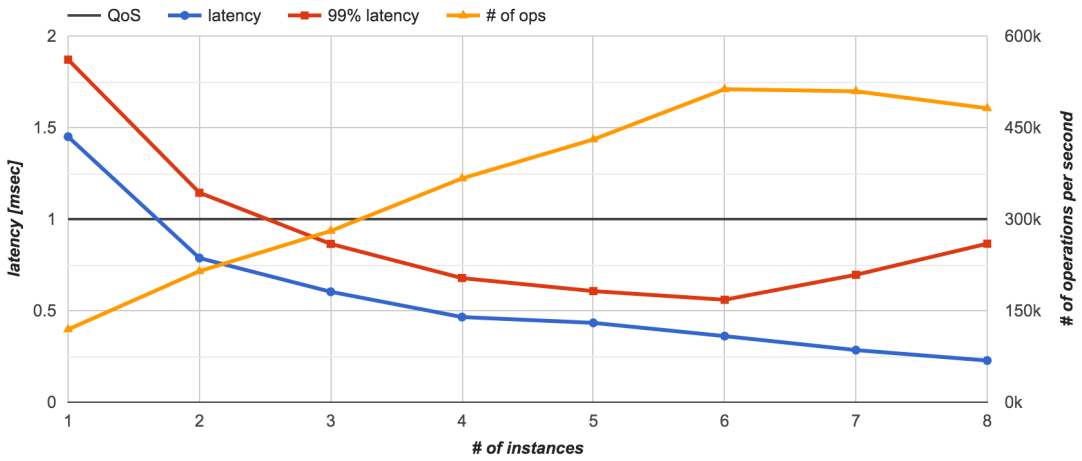

Figure 3.3: Redis Instances: Latency, Throughput vs Number of Instances

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.

Firstly, as the number of instances increases, mean latency decreases. This behavior is expected as requests are processed in parallel, their mean processing time should decrease. We can observe that the decrease in mean latency is linear in terms of the number of instances. Interestingly, as the number of instances grows beyond 6, the trend continues and we see a decrease in the mean latency.

Secondly, the 99th percentile latency decreases steadily as the number of instances increases up to 6. At 6 instances we reach a minimum of 0.56ms. A further increase

in the number of instances results in increase in the tail latency. This effect is due to insufficient parallel level resources (not enough cores) to support more than 6 instances leading to context switching. A request may not be able to be services immediately, as a context switch is required, and therefore remains enqueued. The time spent queuing is the source of the increased tail latency.

Thirdly, the number of operations per second increases linearly with each instance up to 6 instances. At 6 instances, we reach a maximum of 512k requests per second. Increasing the number of instances further only leads to decreased throughput.

The QoS constraint is satisfied with more than 3 instances of Redis. However, maximum throughput with minimum tail latency is only achieved with 6 threads, as many as there are CPU cores.

3.2.2 CPU Utilization

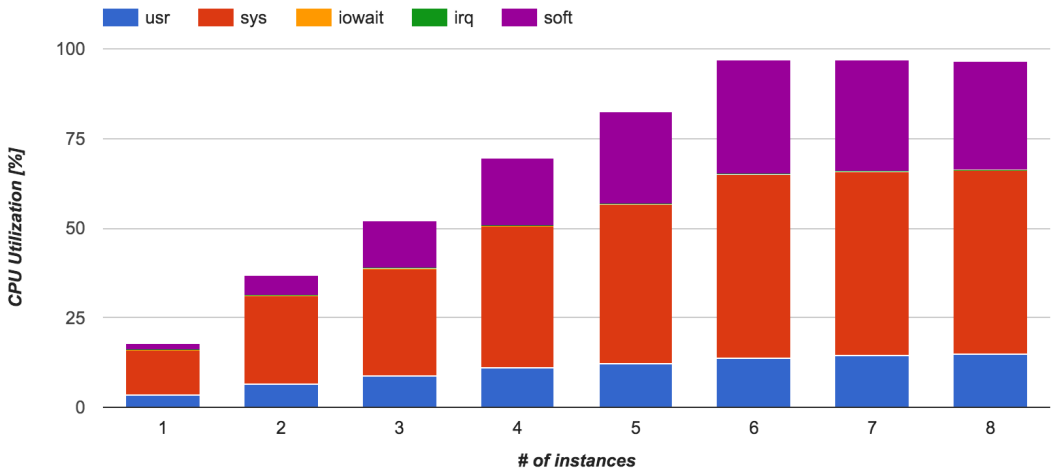

Figure 3.4: Redis Instances: CPU Utilization vs Number of Instances

Figure 3.4 presents a category breakdown of CPU utilization for given category as reported by *mpstat*.

Firstly, CPU utilization of Redis (*usr*) increases linearly with the number of instances up until we reach 6 instances. At this point, CPU utilization stabilizes at 14%. This is due to a larger number of instances requiring more CPU time as well as being scheduled in parallel on multiple cores. When all available CPU cores are exhausted (6 CPUs), multiple instances are scheduled on the same CPU core and therefore total utilization does not increase.

Secondly, the kernel (*sys*) CPU utilization increases with each additional instance and at it's peak of 6 instances accounts for 51% of total CPU utilization. The increase can be attributed to increased number of requests the system is required to handle when processing requests in parallel. With all CPU cores utilized, the system utilization cannot increase further due to hardware constraints.

Thirdly, the time dedicated to processing software interrupts increases linearly too. This is caused by a larger number of requests being processed in parallel which in turn requires an increased number of software interrupts to be triggered in order to process the incoming and outgoing requests.

Finally, *iowait* accounts for no CPU utilization as all operations are performed in memory only. Additionally, *irq* accounts for 0.01% total utilization.

3.2.3 Redis Instances Evaluation

Redis, like Memcached, appears to be network intensive rather than CPU intensive on its own. Multiple Redis instances allow the cache server to scale better and achieve 512k requests per second at 99th percentile latency of 0.56ms. This is close to a 5 fold increase over the single Redis instance benchmark. However, increasing the number of Redis instances also results in key space partitioning. We are no longer able to utilize the server as a singular cache with 6 GB of memory available, instead, we now have 6 individual instances with 1 GB of memory each. In a production environment, the client side would be required to implement consistent hashing in order to be able to utilize the cache entirely. Similarly, a load balancing proxy could be used to spread the load across the instances, however, this setup is outside of the scope of this paper.

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 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

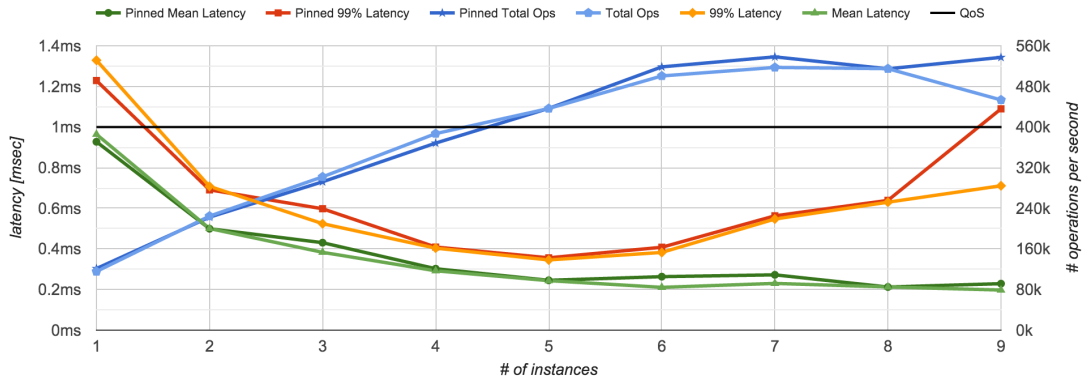

Figure 3.5: Redis Instance Pinning: Instances vs Latency and Throughput

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 3.2 outlines the configuration options for each iteration.

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 3.2: 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.

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 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.

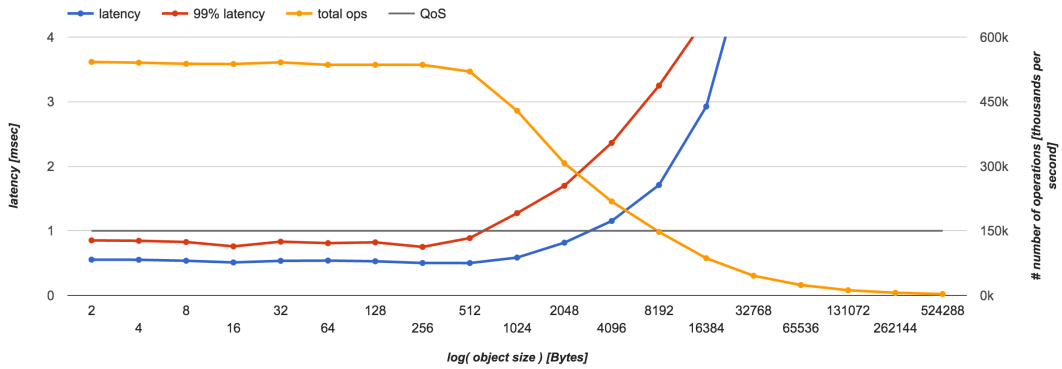

Figure 3.6: Redis Object Size: Latency and Throughput

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

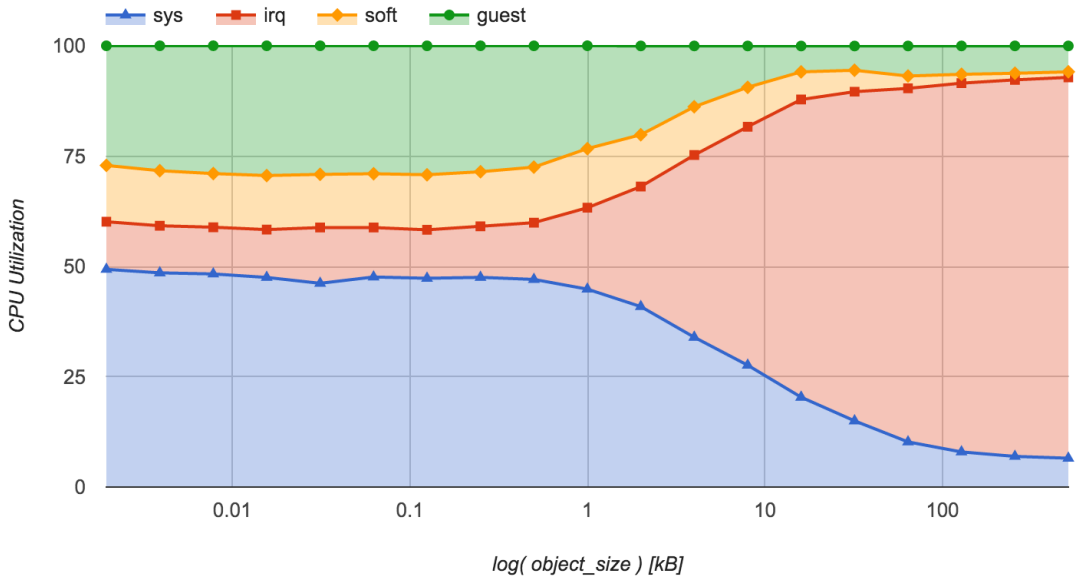

Figure 3.7: Redis Object Size: CPU Usage

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

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

[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] redis.io. Benchmark results on different virtualized and bare-metal servers. `http://redis.io/topics/benchmarks#benchmark-results-on-different-virtualized-and-bare-metal-servers`.

[13] 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.

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

[15] 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.

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

[17] 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.