# Memcached vs Redis: Benchmarking In-memory Object Caches

Milan Pavlik

2016-01-18

# 1 Abstract

Abstract Goes here

# Contents

## 2 Motivation

As the world's demand and reliance on the Internet and near instantaneous communication increases, so do the requirements of computer systems. Clock speed improvements in CPU architectures and shift to multiprocessing architectures are by themselves not sufficient to provide sufficient required computing power. With the improvement in commodity hardware and a shift to commodity computing, it has become increasingly important to design applications capable of utilizing both multiprocessing on a single machine as well as capable of exploiting distributed computing.

Parallelization of work has also introduced an increased complexity system architectures as well as application architectures. The increased complexity is derived primarily from the effort to better utilize multiprocessing. As a result, coherence, scalability and resiliency becomes of great concern to system architects.

The general approach to improving performance is to *"(a) Work harder, (b) Work smarter, and (c) Get help."* [8] Utilizing distributed computing aims to achieve *c)*. An approach to work harder can be utilize parallel architectures better. Additionally, a system cannot always be fully parallelized due to access to shared resources. Due to Amdahl's Law, such systems will not be able to fully utilize the potential speedup provided by advances in architectures alone. Conversely, thinking in the opposite direction in terms of *b)* we can ask if a simpler and less complex architecture can perform better? And if so, what are the aspects of it's design that do make it more performant?

As complexity increases, system architectures are dependent on the ability to perform effectively. Therefore, it is important to understand how a single server scalability is influenced by applications with single and multi threaded architectures.

The motivation behind this study is to understand and evaluate two state of the art object caches with varying architectural decisions in terms of performance scalability on a commodity server. Firstly, we focus is on a well studied object cache called *Memcached*, a high performance application designed with multi-threading as a core feature. Secondly, we focus on a younger cache - *Redis* - a single threaded high performance cache. Finally, having analyzed the performance of two architecturally different object caches, we can evaluate their performance and gain a better insight into the effectiveness of each of their respective design.

# 3 Memory Object Caches

Traditionally, a cache is a data structure in either hardware or software capable of storage and retrieval of data. Generally, a *value* of a computation is stored in the data structure with a given *key*. A cache is generally used to speed up data retrieval. Often, a pattern of execution is to first attempt to retrieve a *value* from the cache by it's *key*. If the *key* is present in the cache, *value* is returned which is called a *hit*. Otherwise, a failed retrieval is indicated and the attempt to access the cache is called a *miss*. If a miss occurs, data is frequently computed or retrieved elsewhere and stored in the cache to speed up the next execution cycle.

Caches are a heavily used across hardware and software systems. For example, the CPU uses multiple levels of caches in order to speed up memory access. Another example of a cache is in database servers to cache queries and reduce computation time. Efficient use of caching can drastically improve time required to retrieve data.

## 3.1 Purpose

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.

## 3.2 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.

## 3.3 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.

## 3.4 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.

## 3.5 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." [3] 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].

### 3.5.1 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.

### 3.5.2 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." [2]

Secondly, in order to provide performance as well as portability, memcached is implemented on top of *libevent* [9]. "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." [9]

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 [10]

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.

### 3.5.3 Production deployments

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

# 4 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.

## 4.1 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 [5].

## 4.2 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.

## 4.3 Workload generation

Workload for the cache server is generated using Memtier Benchmark developed by Redis Labs [4]. 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.

### 4.3.1 Memtier

Memtier benchmark is "a command line utility developed by Redis Labs for load generation and benchmarking NoSQL key-value databases" [4]. 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` [9], 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
3. Tear down `c` connections

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.

### 4.3.2 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.

# 5 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.

## 5.1 Default Performance

Firstly, it is essential to establish a performance baseline of *memcached* under high utilization. In order to establish the baseline, a default configuration of memcached will be used with the exception of the amount of memory allocated for exclusive use by the application. The *memcached* application will be started with

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

specifying the port and the amount of memory to be used by the application.

To find a saturation point, we can increase the number of connections linearly and analyze the results. To load test the cache, we execute the following command on all client servers simultaneously.

```
memtier -s nsl200 -p 11120 -c <connections> -t 2
  --random-data
  --key-minimum=100
  --key-maximum=10000
```

The effect will be to generate random data with keys between the specified ranges and send requests to the server in two simultaneous threads.

### 5.1.1 Default Throughput vs Latency

Firstly, we are interested in the relationship between throughput and latency shown in Figure 1. The mean latency and the 99th percentile latency are plotted against corresponding number of operations (throughput). We can observe that as the number of operations increases so does latency. Additionally, latency (both 99th percentile and mean) increase linearly until a saturation point is reached when a

further increase in throughput is met with an exponentially larger increase in latency. The highest throughput achieved under quality of service restriction of 99th percentile latency under 1 millisecond is 375,000 operations per second. The highest level of throughput corresponds to 84 simultaneous connections, or 12 connections per each client which is similar to benchmarks used in the literature [7].

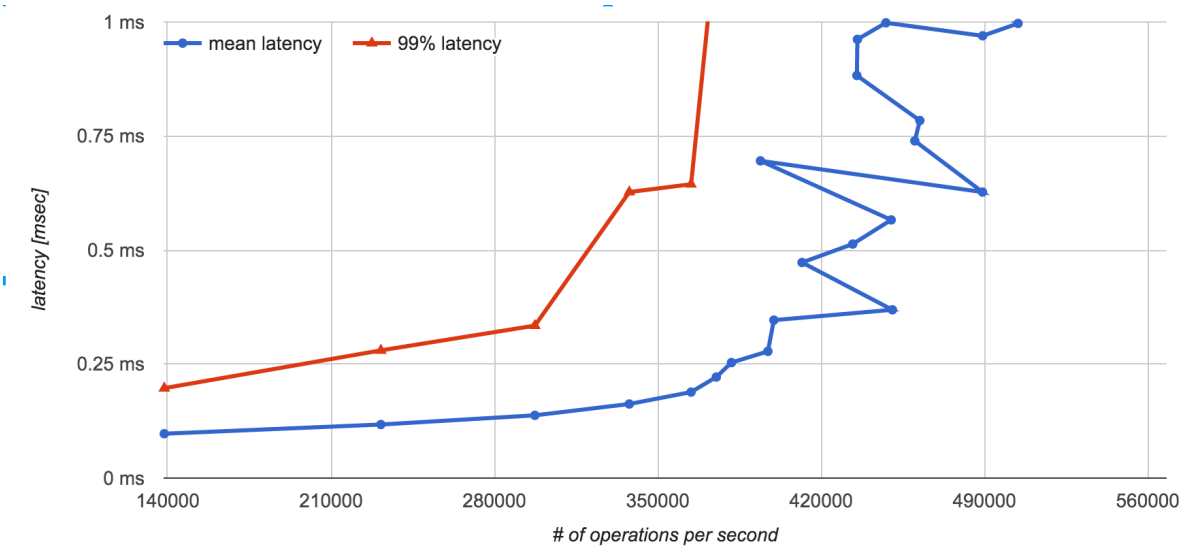


Figure 1: Mean latency and 99th percentile latency against throughput

### 5.1.2 Effect of Connections

To understand the effect of a large number of connections on the cache performance, Figure 2 shows the effect an increase in the total number of connections has on throughput, mean latency and the 99th percentile latency. The figure deliberately shows the behavior outside of the requires QoS requirements in order to better illustrate the impact on the cache under high load.

Firstly, Figure 2 displays the general trend an increased load has on throughput. As load increases, so does throughput. However, as the number of connections surpasses 100, the rate of increase in throughput for each increase the number of connections decreases. This is the saturation point of the cache, an increase in load yield disproportionate increase in throughput. Beyond the saturation point, the cache throughput fluctuates around 450,000 operations per second.

Secondly, the mean latency increases linearly with the number of connections (load). This is an expected behavior as the server experiences network stack queuing as well as increased resource requirements to process requests.

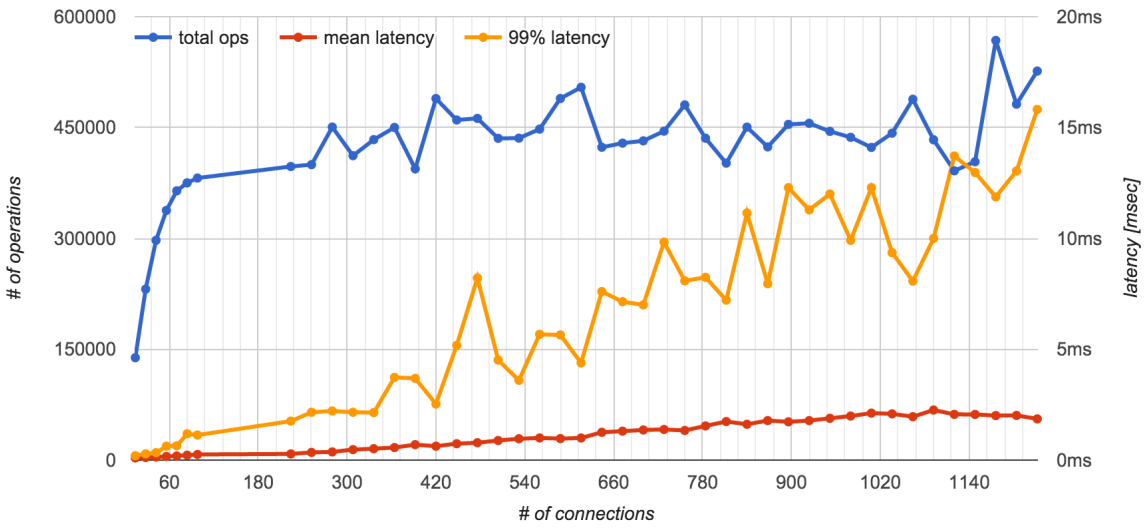


Figure 2: Throughput, Mean latency and 99th percentile against the number of connections

Thirdly, the 99th percentile latency increases linearly, with some fluctuations, against the increased server load. As the load gets higher, the fluctuation increases as well as the upper bound. This is due to some requests being queued for a long time before being processed, pushing the 99th percentile high.

We can observe that the server is capable of scaling much better until around 100 connections are reached. When the saturation point is surpassed, overall performance and quality of service degrades.

### 5.1.3 Server CPU

In order to be better understand the impact of memcached on the system, specifically the CPU usage, we consider a breakdown of CPU time spent in various areas of the operating system in Figure 3.

Firstly, we can observe that the effective footprint of memcached (`guest`) is relatively small compared to the rest. The CPU usage of memcached increases up until 100 connections at which point it remains fairly stable.

Secondly, the operating system (`sys`) increases as we increase the load. Fluctuations occur past 250 connections but the CPU usage by the system remains around 40 percent as the load is increased further. An increased number of connections requires additional resources to process incoming requests as well as process outgoing requests. The context switching from receiving and transmitting contributes to the high load from the system.

Thirdly, interrupt processing by the system (`irq`) decreases as we increase load, this is due to

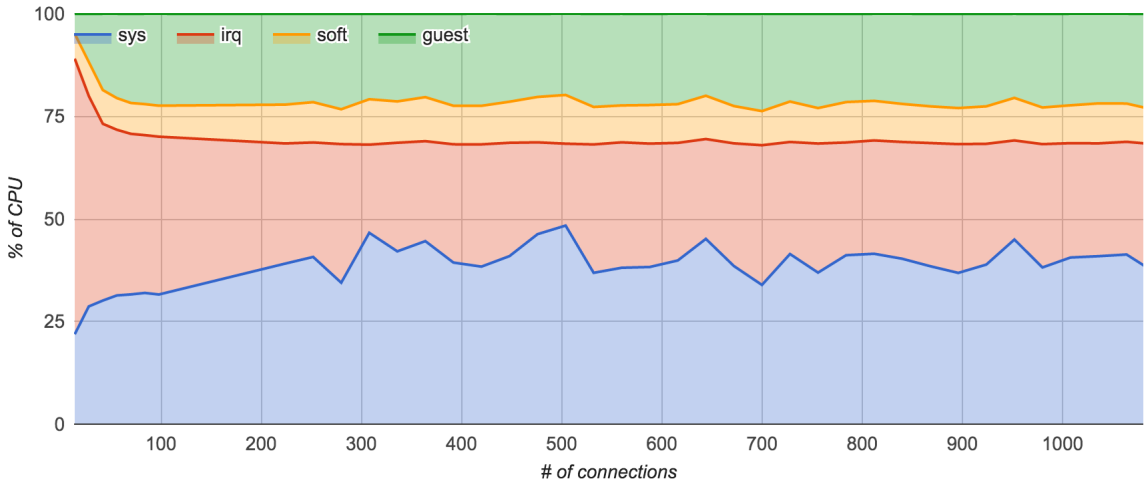


Figure 3: CPU Time against number of connections

having more CPU available and therefore being able to process a higher number of interrupts per unit of time. As resources are required by the system and the memcached application, this number of interrupts processed per unit of time decreases as reflected by the proportion of CPU.

Finally, software interrupt footprint (`soft`) remains relatively stable throughout. The software interrupts correspond to running threads of the memcached application (4 threads by default) and are used for context switching.

From the breakdown in Figure 3, we can conclude that memcached is not CPU heavy on its own. The large CPU footprint of running memcached under high load is tightly linked to performance of the network stack and the underlying hardware processing network requests rather than the application itself. This observation is consistent with findings in MICA [6].

## 5.2 Memcached Thread Scalability

Memcached, as a high performance object cache, is designed to be executed on a parallel 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 provision a larger number of threads for the application.

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 maximising throughput while minimizing 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 [5].

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 nsl200 -p 11120 -c 6 -t 2 -P memcache_binary
  --random-data
  --key-minimum=100
  --key-maximum=10000
```

The server in turn is configured as follows:

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

Where the number of threads is progressively increased.

### 5.2.1 Throughput & Latency

Figure 4 shows the relationship between throughput, mean latency and 99th percentile latency in relation to the number of threads used by a memcached application.

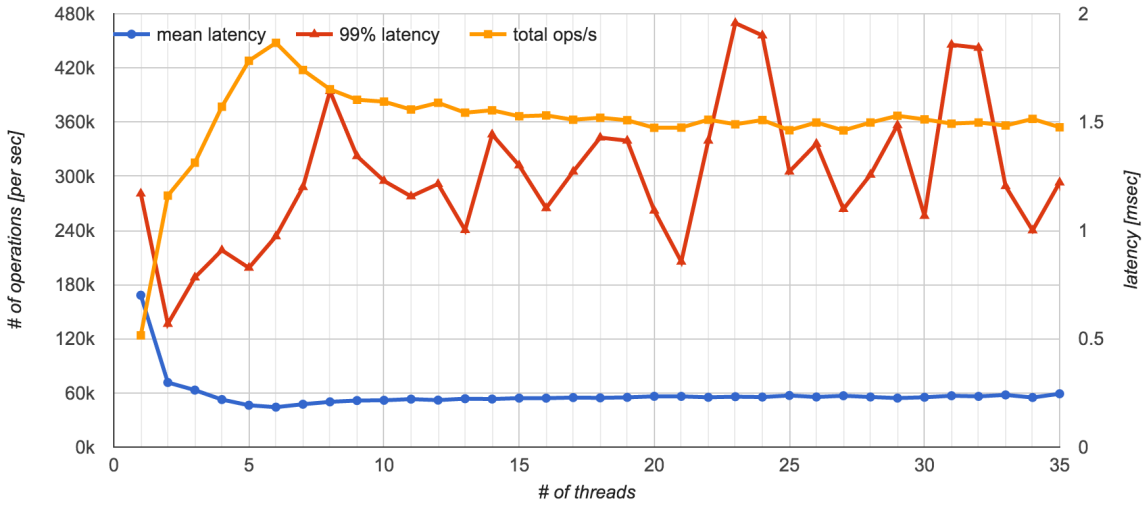


Figure 4: Memcached Thread Scaling

Firstly, we can observe that throughput increases as thread count increases until we reach 6 threads where it peaks at 450k requests per second. As we increase thread count further, throughput decreases. This behavior corresponds with our expectation that performance is maximized when there are as many threads as CPU cores.

Secondly, mean latency decreases as the number of threads is increased reaching a minimum of 0.1843 milliseconds at 6 threads. With more threads, the mean latency increases steadily.

Thirdly, the 99th percentile latency decreases as we increase the number of threads from 1 to 2, reaching a minimum and increasing as the number of threads increases. At 6 threads, we reach a 99th percentile latency of 0.973 which satisfies the QoS requirements under 1 millisecond.

Indeed, as expected we have been able to obtain the highest throughput and achieve the quality of service requirements with 6 threads, as many as CPU cores on the host. Beyond 6 threads, the overhead of context switching between threads increases processing time and reduces throughput.

### 5.2.2 CPU Time

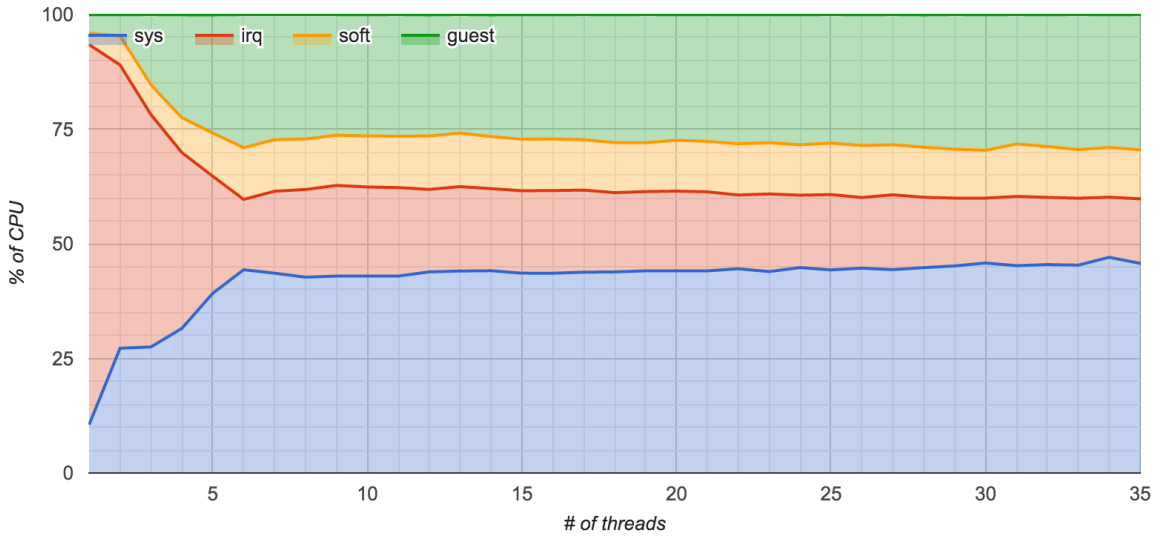


Figure 5: Memcached CPU Time against Number of Threads

Analyzing the CPU usage in Figure 5 as we increase the number of threads, we can observe that initially a large portion of the CPU time is spent servicing hardware interrupts (*irq*). Therefore, the OS is handling incoming traffic interrupts from the NIC. As the number of threads increases, an increasingly larger portion of CPU time is spent processing system calls and context switching (*sys*). This is reasonable as a larger number of threads will require context switching and concurrency management provided by the operating system. We can see that time spent processing hardware interrupts (*irq*) decreases which has the effect of increasing latency as packets remained queued up in the NIC for longer before the OS manages to schedule the interrupt to be serviced. Furthermore, we can observe that software interrupts (*soft*) CPU time progressively increases until we reach 6 threads

and remains stable as the number of threads grows further. The initial increase is reasonable as we are demanding more threads to processed simultaneously, past this point the percentage remains stable as we have reached a saturation point in terms of scalability and server performance. Finally, memcached (*guest*) follows a similar pattern as software interrupts. Usage increases until 6 threads are used and saturates further. This is further indicative of the inability to efficiently scale the number of threads past the point at which memcached uses the same number of threads as CPU cores.

### 5.2.3 Thread evaluation

Comparing results obtain from thread scalability with the results from the default configuration of memcached, we have been able to increase throughput from 375k to 450k requests per second while maintaining the desired QoS under 1ms.

## 5.3 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." [5] 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 a given process affinity by executing 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." [2]. We can discover memcached threads used for request processing using the following command where *tid* is the thread id discovered previously [2].

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

Given the best performance under QoS constraints of 1ms found in the previous section is memcached with 6 threads, the following benchmark will be using this best configuration in order the analyze the impact of thread pinning.

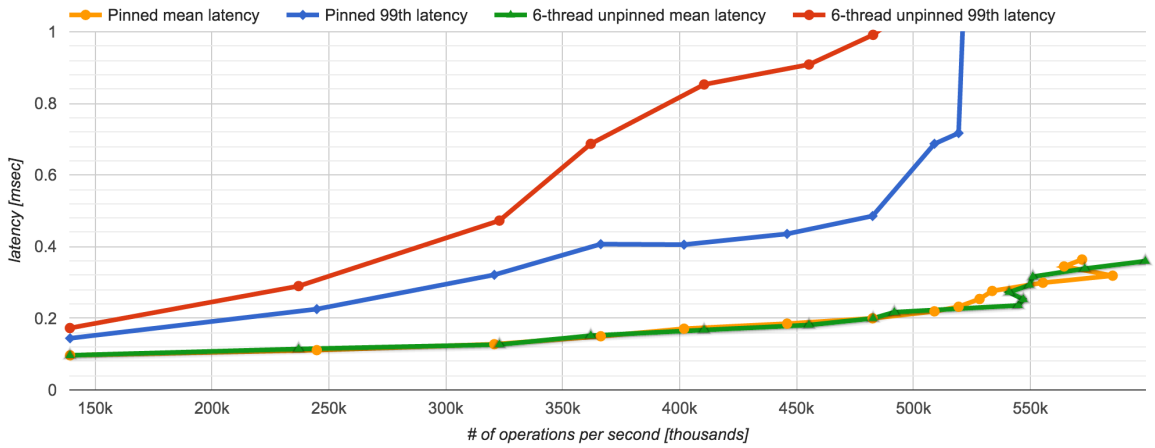


Figure 6: Memcached Pinned Threads vs Unpinned

### 5.3.1 Throughput vs Latency

Figure 6 shows the impact thread pinning has on mean and 99th percentile latency against throughput.

Firstly, the mean latency of both pinned and unpinned benchmarks remains very similar as throughput increases.

Secondly, the 99th percentile latency is lower in the case of pinned threads than unpinned. The pattern holds as throughput increases up until the required QoS boundary.

Furthermore, the throughput has also increased reaching 520k requests per second compared to 475k requests per second in the case of unpinned threads. This pattern is further confirmed by Leverich and Kozyrakis [5].

# 6 Redis

# 7 A Poetic name for comparison of memcached and redis should go here

# References


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

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

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

[5] 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.

[6] 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.

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

[8] Gregory F Pfister. *In search of clusters*. Prentice-Hall, Inc., 1998.

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

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