Evaluation of Redis Performing Asynchronous I/O With Concurrent Queues

Bryant Curto
University of Waterloo

ABSTRACT

Redis is currently the most popular key-value store even though it is largely single threaded. We modified Redis to send responses to clients asynchronously while the main thread performs other tasks by utilizing concurrent queues. Our modified Redis implementation was able to achieve an increase in throughput of up to 35% for one of our benchmarks and 117% for another at the cost of an increase in latency. Nevertheless, we believe that our findings reinforce the idea that Redis has untapped potential and it would benefit from taking greater advantage of more cores.

1 INTRODUCTION

Redis is an open source, production quality, in-memory data structure store that can be used as a key-value database [10]. It is used by many companies like Engine Yard, Github, Craigslist, Disqus, Digg, and Blizard and has been ranked as the most popular key-value database for over five years [4, 12]. It may come as a surprise that Redis is largely single threaded [9]. This paper aims to understand what performance gains can be achieved by enabling Redis to take greater advantage of more cores by enabling greater parallelism within the server. We achieve this using several concurrent queue implementations, one of which being a relaxed concurrent queue. The findings of this work are beneficial to any company that uses Redis for their high performance applications.

2 BACKGROUND

2.1 Redis

Redis can be used as a key-value store through which remote clients can store and retrieve entries. A client can map a key to a value in the database using the `SET` command and can retrieve the value associated with a key using the `GET` command. By default, Redis utilizes a single thread for accepting client connections, reading requests from client sockets, executing client commands, and writing responses to client sockets [9]. As of version 6, Redis can be configured to use multiple threads in order to perform I/O (i.e., reading and writing to client sockets). However, it is important to note that the execution of client commands is still performed using a single thread. When configured to use multiple threads, Redis cycles through stages of reading from client sockets in parallel, executing client commands synchronously, and writing to client sockets in parallel. (If configured to use a single thread, these parallel phases are simply synchronous phases as well.)

Redis' existing implementation for performing parallel I/O has several undesirable characteristics. First, the Redis server can't transition out of a parallel stage until all threads have finished their work. This design can be especially problematic as jobs are evenly divided amongst threads with no possibility of one thread stealing work from another. As a result, a single thread completing one or a sequence of abnormally large I/O jobs can cause the progress of the entire server to be halted.

However, this design has several beneficial results. By dividing execution of Redis into these stages, very little or no synchronization of shared state needs to be performed within each stage. This results in simpler code and fewer concurrency bugs.

We attempt to improve the performance of Redis by breaking this three phase cycle. Our modified Redis implementation transitions back and forth between synchronously reading from client sockets and executing client commands while a pool of threads perform writes to client sockets asynchronously.

2.2 Concurrent Queue

A concurrent queue is simply a thread-safe first-in-first-out data structure. However, for data structures and especially ordered data structures, scalability (the ability to increase in performance as the number of threads increases) is in conflict with correctness (the guarantee that only certain behaviors can occur) [5]. For example, assume that many threads want to dequeue items on a concurrent queue. For the queue to be strictly FIFO, meaning that elements are parceled out in order of age, their operations must take effect sequentially. As more threads try to enqueue at a given instant, the number of sequential operations that occur increases, resulting in poor scalability.

Relaxed data structures attempt to improve scalability by allowing more behaviors to be exhibited than their more strict counterparts. For example, on a call to dequeue, a k-relaxed concurrent queue ($k >= 0$) behaves like a concurrent queue but can return any of the $k + 1$ oldest elements.

3 RELATED WORK

Many have sought to enable Redis to utilize more cores of a machine.

Caciu et al. propose Node Replication (NR), a black-box approach for obtaining efficient concurrent data structures [6]. The method takes a sequential data structure and automatically transforms it into a NUMA-aware concurrent data structure satisfying linearizability. They apply NR to the data structures of Redis and achieve performance gains of up to 14x. To achieve these performance gains, NR implements a NUMA-aware shared log which it uses to replicate data structures across NUMA nodes. However, their adaptation of Redis to use NR serves as an example of the possible performance gains achievable by using NR. One is left to wonder if further performance improvements can be achieved by making more invasive modifications to Redis as we have done.

A high performance fork of Redis, named KeyDB, is also in development [15]. It has a focus on multithreading, memory efficiency, and high throughput. While KeyDB is open source, this is only one of three versions of the system – the other two require a license. Further, the open source version contains the smallest subset of the features KeyDB is able to support [14]. This is dissimilar from Redis, which is entirely open source. It is possible that this difference is the reason why Redis is a far more popular key-value store than KeyDB [2]. No matter the reason, Redis' popularity makes it an important target for optimization.

4 IMPLEMENTATION

4.1 Redis Modifications

For Redis to be able to write to client sockets asynchronously, many modifications were made.

In Redis version 6.2, server state is stored in a globally accessible `redisServer` instance. Each client connection is represented using a `client` instance, and all instances are accessible through the global `redisServer` instance. Each `client` instance contains buffers holding unexecuted commands from the corresponding client, unsent responses, and other information.

The first step in enabling Redis to write to client sockets in parallel while performing other tasks is to synchronize access to all shared state. In this instance, the shared state is the `client` instances. This was accomplished by first identifying all locations in the codebase where a thread acquires a reference to a `client` instance from the globally accessible `redisServer` instance. (This identification was performed by manually searching for lines in the codebase where any `redisServer` members were accessed.) A recursive lock was added to the `client` definition for synchronizing access to each `client` instance.

The second step is to enable the main thread, which reads client sockets and executes commands, to notify the thread pool as to which clients have awaiting responses. To do this, we evaluated the performance of our modified Redis implementation using several concurrent queue implementations. We used queue implementations from Scal, an open-source benchmarking framework that contains implementations of a large set of concurrent data structures [7]. Since Scal is implemented in C++ and Redis in C, we also developed a C interface to make use of Scal's queue implementations.

The first queue evaluated is referred to as the Lock-based Singly-linked List Queue by Scal. It is a simple queue implementation. Enqueuing an element entails acquiring a global queue lock, enqueuing the element on the linked-list queue, and releasing the lock. Dequeuing similarly entails acquiring a global lock, dequeuing the element from the linked-list queue, and releasing the lock. If the queue is empty, the caller is notified by the returned value. In this paper, the modified Redis implementation that uses this queue is referred to as the Lock-Queue Redis implementation.

The second queue evaluated is referred to as the Michael Scott (MS) Queue. It is an implementation of the non-blocking concurrent queue algorithm proposed by Michael and Scott [13]. The MS Queue has the same behavior as the Lock-based Singly-linked List Queue, but enqueue and dequeue operations utilize the compare-and-swap primitive instead of using locks. In this paper, the modified Redis implementation that uses this queue is referred to as the MS-Queue Redis implementation.

The third and final queue evaluated is referred to as the b-RR Distributed Queue (DQ). It is an implementation of the k-relaxed queue algorithm proposed by Haas et al. [8]. It is a distributed queue consisting of multiple FIFO queues, called partial queues, which are Michael Scott queues. Upon an enqueue or dequeue operation, one of the partial queues is selected for performing the actual operation without further coordination with the other partial queues. Selection of the partial queue for a given operation is done using a round-robin load balancer. As is implied by the evaluations performed in the DQ paper, we set the number of partial queues to our maximum core count (i.e., 32). (More specifically, the b-RR Distributed Queue is linearizable with respect to the sequential specification of a k-relaxed queue with k bounded in the number of round-robin counters, the number of partial queues, and the number of threads.) When dequeueing, all partial queues are checked up to two times before notifying the caller that the queue is empty. In this paper, the modified Redis implementation that uses this queue is referred to as the Distributed-Queue Redis implementation.

4.2 Validating Modifications

In order to have some degree of confidence that our modified Redis implementations do not have concurrency bugs, several checks were performed.

First, we disabled all codepaths not used by our permitted operations (i.e., `PING`, `GET`, and `SET`) that access members of the global `redisServer` instance. Executing any such codepath would result in Redis crashing.

Second, we ran GCC's ThreadSanitizer [3], a data race detector, on our modified Redis implementations. To do this, we recompiled our Redis implementations and their dependencies with the ThreadSanitizer option enabled and ran a subset of our benchmarks for four hours on each modified Redis implementation. Results indicate that we did not create data races.

Lastly, we modified our benchmarking tool to assert the validity of responses received by clients. When the response was expected to be of a fixed size in bytes, the benchmarking tool asserted that all received responses had the expected size. When the response was expected to have one of several sizes, the benchmarking tool asserted that all received responses had one of the expected sizes.

5 EXPERIMENTAL SETUP

5.1 Hardware

Redis and Redis clients are run on two separate hosts. We refer to the machine running the server as the server machine and the machine running the clients as the client machine. Both machines have two 2.70 GHz Intel Xeon CPU E5-2680 with 16 cores, 2 hyper-threads per core, and 64 GB of RAM. The server machine is running Ubuntu version 20.04.1 with Linux kernel version 5.4.0. The client machine is running Arch Linux with Linux kernel version 5.9.4. The machines communicate over a dedicated link using 40 Gbps network interfaces.

5.2 Configuration

Event logging and flushing of the database to disk are disabled on all Redis instances during all experiments. To ensure that the OS assigns Redis threads to cores consistently across all tests, we employ the following scheme using `taskset`. When the server utilizes 8 or fewer threads, all threads are pinned to all of the cores on the first socket and hyper-threading is disabled. When the server utilizes 16 or fewer threads, all threads are pinned to all cores and hyper-threading is disabled. When the server utilizes 32 or fewer threads, we left the OS to assign threads to cores as desired. Further, to ensure consistency across experiments, we fixed the frequency of the cores to the performance governor using `cpupower-frequency-set`.

5.3 Workload

To compare the performance of the Redis implementations, we simulated clients requesting entries from the key-value store where the value returned had one of various sizes.

Before each experiment was run, the key-value store of the Redis instance being evaluated was populated with one or more entries. The number of entries and the size of each entry is determined by the experiment.

The benchmarking tool was then used to create clients through which `GET` requests, specified on one of the previously described keys, were submitted to the Redis instance. The server would then respond with one of the corresponding, previously described values, thereby exercising the adaptation.

Redis uses a Request/Response protocol [11]. This means that, after a given client has submitted a request to the server, the client must wait for a response prior to submitting another request. As a result, we experimentally selected for the benchmarking tool to use 64 threads to make requests to the Redis instance using 800 clients.

We made a few modifications to the benchmarking tool as well. By default, the tool submits a fixed number of requests before stopping. We modified the tool so it would continuously send requests until a specified amount of time passed. Further, in order to allow the throughput to stabilize, we added an additional timer to delay the collection of statistics for a specified amount of time. Finally, we enabled the benchmarking tool to perform assertions on the server responses. This was done to help build confidence in the correctness of our Redis implementations as previously described.

6 EVALUATION

We compare the performance of the unmodified Redis implementation with that of each of our modified Redis implementations when we vary the number of threads. Note that, as previously explained, the unmodified Redis implementation cycles through stages of reading client sockets, executing client commands, and writing to client sockets. When writing to client sockets, all threads perform writes. For all three of our modified Redis implementations, one thread reads client sockets and executes client commands while all other threads write to client sockets.

Each experiment was run for 70 seconds with a delay in the collection of statistics for 10 seconds. Therefore, measurements were collected for a 60 second interval. In all throughput plots, 95% confidence intervals are displayed along with means. Each point in each plot represents a distinct execution of the benchmarking tool.

Bryant Curto
University of Waterloo

6.1 Zipfian Response Sizes

For our first experiment, we configured the workload so that the Redis implementation being evaluated responds to client requests with a response of size 2^N bytes, where N is in range 0 to 25. The distribution over the response sizes is Zipfian. Responses are very likely to be one byte in size and the probability of a response being larger decreases rapidly. This workload is used to understand what (if any) performance gains exist by enabling the server to proceed with receiving client requests and executing commands without having to wait for all client responses to be sent.

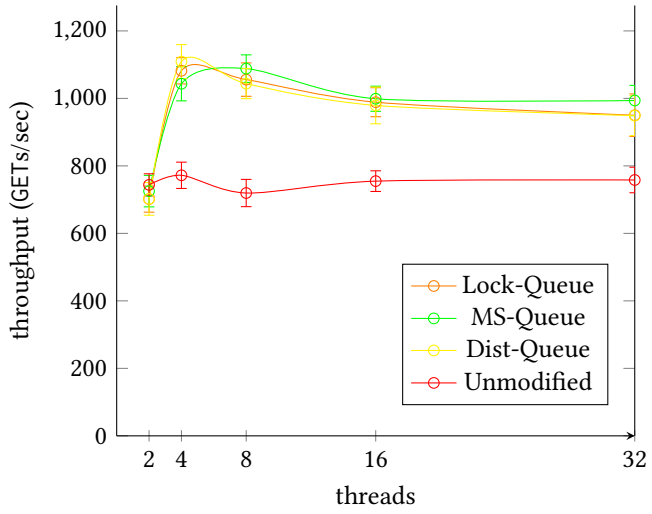


Figure 1: Throughput of responses when response sizes follows Zipfian distribution and Redis instance uses varying number of threads.

Looking at Figure 1, the throughput of the modified Redis implementations is higher than that of the unmodified Redis implementation for all setups, except when two threads are used. When two threads are used, the performance of all is on par. When four or more threads are used, the performance of the modified implementations are at most 35% higher than that of the unmodified implementation. The observed plateau in throughput suggests that Redis is bottlenecked. Considering that even the Distributed-Queue Redis implementation also plateaus, this suggests that the bottleneck is not caused by contention over the concurrent queue. However, more work is needed to determine what that bottleneck may be, as well as why performance of all implementations is on par when using two threads.

The difference in the throughput of the modified Redis implementations are not only similar, the difference is statistically insignificant since the confidence intervals overlap. This suggests that the choice of concurrent queue did not have a significant impact on performance. Instead, these

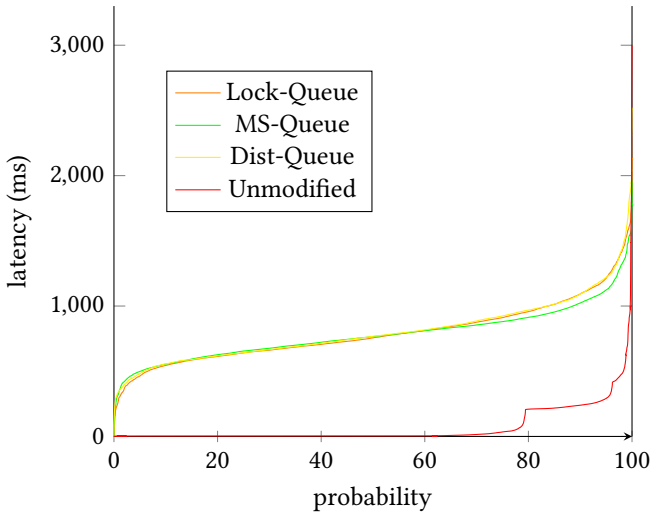


Figure 2: Latency of responses when response sizes follows Zipfian distribution and Redis instance uses 16 threads.

results suggest that enabling Redis to respond to clients asynchronously provided the performance improvement.

Looking at Figure 2, we see the cumulative distribution of latencies in milliseconds for all implementations using 16 threads. As may have been expected, the modified Redis implementations trade higher latency for higher throughput. Consider, the unmodified Redis implementation strives for low latency by periodically dedicating all of its resources (i.e., threads) to sending client responses. Meanwhile, the modified Redis implementations can and will queue client responses, which are sent only when the responses reach the head of the queue. By comparing the plots for the modified implementations and the unmodified implementation in Figure 2, one gets the impression that the average client response resides in the queue of any given modified implementations for a around 700 milliseconds.

The results shown in Figures 1 and 2 are surprising to us. We anticipated that we might observe lower latency and higher throughput in the MS-Queue Redis implementation as compared to the Lock-Queue Redis implementation since the prior is lock-free and the latter is not. However, we expected to observe lower latency and higher throughput in the Distributed-Queue Redis implementation as compared to the Lock-Queue and the MS-Queue Redis implementations because the Distributed-Queue Redis implementation uses a relaxed queue.

6.2 Constant Response Sizes

Using the previous setup, we now configure the workload so that implementations respond to client requests with a

response of a constant size. This workload is used to understand what (if any) performance gains exist by sending client responses asynchronously. This contrasts with the previous experiment in that the workload does not exploit the unmodified Redis implementation's naive allocation of jobs to threads when performing parallel writes to client sockets. The results from this experiment can be seen in Figure 3. (Note that more response sizes were evaluated than those shown. The trends observable in the presented results mirror the trends observable in those not presented.)

When responses are of a size 512 bytes or below, the behavior of the servers is unexpected – see Figures 3a and 3b. Throughput of the MS-Queue and Distributed-Queue Redis implementations change rather drastically as the number of threads is varied. The increase in performance of the MS-Queue Redis implementation using 32 threads as response sizes increase may indicate contention over shared memory. We would expect smaller responses to take less time and, therefore, result in more contention on the MS Queue. However, it is not clear why the Distributed-Queue Redis implementation using 16 threads performed so differently with these response sizes. The Lock-Queue Redis implementation performs poorly as the number of threads is increased, which is expected. We would expect sending responses of these sizes to be very fast, resulting in high contention on the lock used in the Lock-based Singly-linked List Queue. The unmodified Redis implementation performs consistently as the number of threads is increased.

When responses are of a size larger than 512 bytes, the behavior of the implementations is similar to that observed with the Zipfian workload. For a given response size, performance of the modified Redis implementations are similar as the number of threads is varied – see Figures 3c, 3d, 3e, and 3f. This suggests that writing threads may be spending much of their time writing to sockets instead of accessing the concurrent queue.

When response sizes are 2,097,152 (2^{21}) bytes or greater – Figures 3e and 3f – performance of the modified implementations are, in general, higher than that of the unmodified implementation. For these response sizes, throughput of the modified Redis implementations is as much as 117% higher than that of the unmodified Redis implementation.

Throughput appears to plateau as the number of threads is increased when response sizes are 8,192 (2^{13}), 2,097,152 (2^{21}), or 33,554,432 (2^{25}) bytes in size but not when they are 131,072 (2^{17}) bytes. Additionally, the throughputs of all implementations using two threads is the same for responses of all of these sizes except 2,097,152 (2^{21}) bytes.

The behavior observed using this workload cannot be easily explained and deserves further evaluation.

7 DISCUSSION & FUTURE WORK

The performance of the different modified Redis implementations is surprising, whether that be the similarity or dissimilarity between implementations or the trends observed. Our next step would be to determine the underlying causes. Further, it is possible that the difference in performance of our modified Redis implementations is drowned out by running our benchmarks across a network.

The experiments we ran only show some of the benefits of our modification. We assumed that it takes the implementation some amount of time to receive client `GET` requests and execute the client command (i.e., retrieve and prepare the data to be sent back to the client). Additional experiments should be run where the server would need to spend more time performing these steps. Doing so might show further benefits to sending client responses asynchronously.

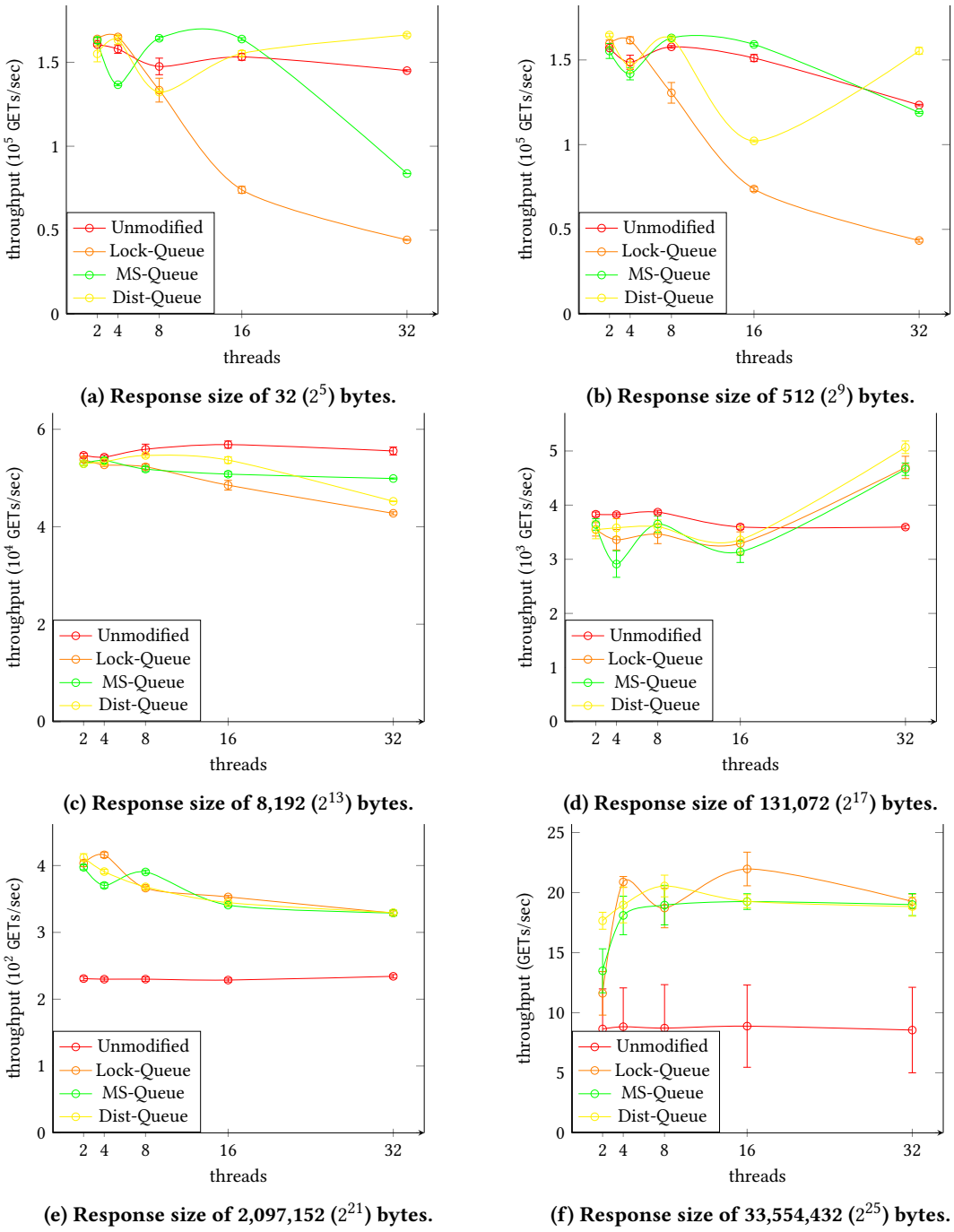
In our implementations, Redis sends responses to clients in parallel with receiving client requests and executing client commands. A next step in this work is to enable our Redis implementations to read client requests in parallel with writes of client responses and executions of client commands. The modifications needed to allow for asynchronous reads are very similar to allowing for asynchronous writes. The reason we initially augmented Redis to perform asynchronous writes instead of asynchronous reads is because the Redis developers suggest that parallel reads do not make a large impact on performance [1] (see `io-threads-do-reads`).

Our modifications of Redis only supports a subset of the operations supported by the original Redis implementation. The `PING`, `GET`, and `SET` commands are known to work with our modified implementations. However, many code paths were disabled to increase confidence that our modifications did not introduce concurrency bugs.

Given another chance, we would have spent more time analyzing the performance results and less time on creating several Redis implementations using different concurrent queues.

8 CONCLUSION

In this work, we adapt Redis to send responses to clients in parallel with reading client requests and executing client commands using concurrent queues. For one of our benchmarks, our modified Redis implementation was able to achieve an increase in throughput of up to 35% at the cost of an increase in latency. For another benchmark, our modified Redis implementation was able to achieve an increase in throughput of up to 117%. Further work must be performed to better understand and explain the behavior of our modified Redis implementations. Nevertheless, we believe that our findings reinforce the idea that Redis has untapped potential and it would benefit from taking greater advantage of more cores.



(a) Response size of 32 (2^5) bytes.

(b) Response size of 512 (2^9) bytes.

(c) Response size of 8,192 (2^{13}) bytes.

(d) Response size of 131,072 (2^{17}) bytes.

(e) Response size of 2,097,152 (2^{21}) bytes.

(f) Response size of 33,554,432 (2^{25}) bytes.

Figure 3: Throughput of Redis implementations when response size is constant.

9 AVAILABILITY

Our modified Redis implementations can be found (here) and our benchmarking tools can be found (here).

REFERENCES

[1] [n.d.]. https://github.com/redis/redis/blob/761d7d27711edfbf737def41ff28f5b325fb16c8/redis.conf. Accessed: 2021-04-25.

[2] [n.d.]. System Properties Comparison KeyDB vs. Redis. https://db-engines.com/en/system/KeyDB%3BRedis. Accessed: 2021-04-25.

[3] 2020. ThreadSanitizerCppManual. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual. Accessed: 2021-04-24.

[4] 2021. DB-Engines Ranking of Key-value Stores. https://db-engines.com/en/ranking/key-value+store. Accessed: 2021-04-25.

[5] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 487–498. https://doi.org/10.1145/1926385.1926442

[6] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[7] Andreas Haas, Thomas Hütter, Christoph M. Kirsch, Michael Lippautz, Mario Preishuber, and Ana Sokolova. 2015. Scal: A Benchmarking Suite for Concurrent Data Structures. In *Networked Systems*, Ahmed Bouajjani and Hugues Fauconnier (Eds.). Springer International Publishing, Cham, 1–14.

[8] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. 2013. Distributed Queues in Shared Memory: Multicore Performance and Scalability through Quantitative Relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '13)*. Association for Computing Machinery, New York, NY, USA, Article 17, 9 pages. https://doi.org/10.1145/2482767.2482789

[9] Redis Labs. [n.d.]. https://redis.io/topics/benchmarks. Accessed: 2021-04-25.

[10] Redis Labs. [n.d.]. Introduction to Redis. https://redis.io/topics/introduction. Accessed: 2021-04-25.

[11] Redis Labs. [n.d.]. Using pipelining to speedup Redis queries. https://redis.io/topics/pipelining. Accessed: 2021-04-25.

[12] Tiago Macedo and Fred Oliveira. 2011. *Redis Cookbook: Practical Techniques for Fast Data Manipulation.* " O'Reilly Media, Inc.".

[13] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[14] EQ Alpha Technology Ltd. & others. [n.d.]. Start Using KeyDB in Minutes. https://docs.keydb.dev/docs/using-keydb. Accessed: 2021-04-25.

[15] EQ Alpha Technology Ltd. & others. [n.d.]. Welcome to KeyDB's Documentation. https://docs.keydb.dev/docs/. Accessed: 2021-04-25.