# Evaluation of Redis Using io_uring Communication

Bryant Curto
University of Waterloo

## ABSTRACT

`io_uring` is a new asynchronous I/O framework on Linux that purports efficiency and a simple API. To evaluate these claims, Redis, a high-performance key-value store, is adapted to use `io_uring` to asynchronously communicate with clients. The performance of several variants of the Redis server adapted to use `io_uring` are then evaluated. One variant is found to achieve an increase in maximum throughput of up to 31% as compared to the unmodified Redis server.

## 1 INTRODUCTION

`io_uring` is an asynchronous I/O framework that recently appeared in the Linux kernel. It's aim is to offer an efficient implementation and a simple interface unlike previous available asynchronous I/O frameworks on Linux [5]. Interest in this framework has been rapidly increasing as too has the number of operations it supports [7].

This paper aims to understand what performance gains can be made using `io_uring` as compared to the standard `read` and `write` system calls in a high performance, system intensive application. To achieve this aim, Redis, a high performance, production quality, in-memory key-value store, is adapted to use `io_uring` in order to send responses to clients. Redis 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]. The findings of this work are beneficial to any company that use Redis, and anyone developing a high performance, system intensive application.

## 2 BACKGROUND

### 2.1 io_uring

Asynchronous I/O is the ability to perform some input or output operation, possibly to disk or a network, without having to wait for the operation to complete. This is valuable since I/O operations are frequently very expensive, resulting in wasted processor cycles waiting for the operation to complete. Since version 2.1, the Linux kernel has had an asynchronous I/O framework (dubbed aio) [5]. However, it has an unfriendly interface and limited guarantees on I/O operations actually being executed asynchronously which makes it unpopular for many systems developers [2].

`io_uring` is an asynchronous I/O framework, introduced in Linux kernel version 5.1, that aims to succeed where where aio failed. Each `io_uring` instance has two single producer, single consumer queues–a submission queue and a completion queue, respectively–implemented using ring buffers [2]. An application submits operations to be executed by the kernel by creating one or more submission queue entries and appending them to the tail of the submission queue. Once appended, the entries are submitted to the kernel with a call to `io_uring_enter`. The kernel pops entries off of the head of the submission queue for processing. Once an operation has been completed, the kernel appends a completion queue entry to the tail of the completion queue. Finally, the application pops entries off of the head of the completion queue to learn of an operation's completion.

By using memory barriers, entries can be added to the submission queue and removed from the completion queue without the use of locks. Using `io_uring_enter`, the application submits entries and can wait for a specified number of entries to be completed. The application can also continue executing and check the head of the completion queue for entries without having to perform another system call.

Using `io_uring`, an application can execute several I/O operations asynchronously with only a single system call.

### 2.2 Redis

Redis is an open source, production quality, in-memory data structure store that can be used as a key-value database [10]. It maintains the database in main memory and periodically writes a copy of the database to disk in order to guarantee persistence.

Clients communicate with the server over TCP. 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, a Redis server instance uses a single thread of execution to handle all client requests.

### 2.3 System Call

System calls are the method by which applications interact with the operating system kernel. They are used to request services offered by and implemented in the operating system kernel. Typically, system calls are invoked by writing arguments to specific registers or locations on the stack and then issuing a system call machine instruction. This results in execution switching from user-mode to kernel-mode. Once the system call has been completed, execution switches back from kernel-mode to user-mode.

However, system calls negatively impact the performance of system intensive workloads for two main reasons. First, there is a direct cost in switching from user mode to kernel mode. Second, there is an indirect cost in polluting of processor structures (e.g., caches) that affect both user-mode and kernel-model performance [14]. Further, Linux has employed a strategy named KPTI to mitigate the Meltdown security vulnerability that adds additional overhead to system calls [6]. The overhead of this mitigation was measured to be 0.28% by KPTI's original authors [8]. However, others have found applications for which overheads are as high as 30% [6].

## 3 RELATED WORK

Redis has previously been adapted to utilize emerging networking technologies and frameworks.

Tang et al. adapts Redis to use IP-over-InfiniBand (IPoIB) and Remote Direct Memory Access (RDMA), two modern network communication technologies, for receiving requests and sending responses to clients. They find that the average latency of `SET` operations of RDMA based Redis is about two times faster than IPoIB based Redis [15]. Meanwhile, Qi et al. designs a high performance Redis implementation by replacing use of sockets with Fast Event Driven RDMA RPCs (FeRR) [13].

RDMA allows one machine to directly access the memory of a remote machine without the need for involving the operating system of either host [9]. It enables zero-copy transfer of data, reduced latency, and reduced CPU overhead. While usage of these networking technologies and frameworks should be the goal, they are primarily targeted for use within data centers [3]. As a result, this work serves to help bridge the gap for those making use of available network communication technologies and frameworks outside of a data center environment until they become more ubiquitous.

## 4 EXPERIMENTAL SETUP

### 4.1 Hardware

The Redis server and Redis clients are run on two separate machines. 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 a 2.70 GHz Intel Xeon CPU E5-2680 with 16 cores 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.

### 4.2 Setup

To ensure that throughput is not bottlenecked by hardware resources, `vmstat` is utilized to monitor memory and CPU usage and `ifstat` to monitor network usage. For all experiments, hyperthreading is disabled on both machines to ensure that performance can not be influenced by the operating system assigning multiple threads to the same physical core.

For the Redis server, event logging and flushing of the database to disk are disabled. Redis version 6.0.9 is used for all experiments.

### 4.3 Workload

To evaluate the performance of the Redis server using `io_uring` to write to client sockets, the benchmarking tool that comes with the Redis server is used. The server is initialized with a key-value entry where the value has a specified size determined by the experiment. The benchmarking tool creates clients to submit `GET` requests specifying the key of the previously described entry. This results in the server responding with the associated value, 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, it must wait for a response prior to submitting another request. As a result, 64 threads are spawnd to make requests to the Redis server using 1000 clients, where clients are evenly assigned to threads. (These values were determined experimentally.) During the benchmark, each thread submits requests to the server using its assigned clients.

The benchmarking tool was adapted to log the number of requests made by all clients each second of a given experiment. Each experiment is run for 150 seconds. Only measurements from the last 120 seconds of each experiment are captured as client initialization will influenced the measurements.

## 5 METHODOLOGY

### 5.1 Unmodified Redis Server

In later discussions, the unmodified server refers to the Redis server that has had no other modifications beyond those described in the Setup. It uses socket multiplexing and non-blocking `write` system calls to respond to clients [1]. This means that the server only writes to a socket if that write can be done without causing the writing thread to block. This is especially important since the Redis server uses only a single thread of execution to handle client requests and responses.

### 5.2 Asynchronous Batching io_uring Redis Server

The async/batch server refers to the Redis server that has been augmented to use `io_uring` to perform asynchronous, batched writes to client sockets. Writes to client sockets are performed asynchronously, meaning that the thread submitting the write continues executing before the write has been completed. Additionally, writes to multiple client sockets are batched together and passed to `io_uring` for execution using a single system call.

Redis has a single thread for handling client requests and responses. As a result, after initialization, it executes in a loop, using socket multiplexing to take action only when an action (e.g., read or write) can be performed on a socket. In my implementation, batches of writes to client sockets are submitted once per cycle around this loop, which I refer to as the main loop.

## 5.3 Asynchronous io_uring Redis Server

The async server refers to the Redis server augmented to use `io_uring` to perform asynchronous writes to client sockets. Unlike the async/batch server, this server passes one write operation to a client socket to `io_uring` for asynchronous execution at a time. This results in one system call for every write.

By comparing the performance of this server to that of the async/batch server, one can determine what gains are made by making fewer system calls.

## 5.4 Sequential io_uring Redis Server

The sequential server refers to the Redis server augmented to use `io_uring` to perform synchronous writes to client sockets. Unlike the sync server, this server passes one write to `io_uring` and waits for completion before continuing execution.

The performance of this server can be used to understand the overhead of the implementation of the Redis server augmented to use `io_uring`.

## 6 EVALUATION

The results of the evaluation can be seen in Figure 1. It shows the maximum throughput of each variant of the Redis server for a variety of response sizes: the unmodified server, the async/batch server, the async server, and the sequential server. All means are presented with 95% confidence intervals, but the size of the confidence interval makes them difficult to see.

All variants of the Redis server outperform the sequential server, which submits each write to a client's socket to `io_uring` and then waits for completion. This can likely be attributed to the overhead in putting the writing thread to sleep after submitting the write to `io_uring` and waking the thread up once the write has been completed.

The async server outperforms the unmodified server by up to 8.7% for responses of size $2^6$ to $2^9$ bytes. The difference in throughput can be understood as the gains made from executing write calls asynchronously via `io_uring`.

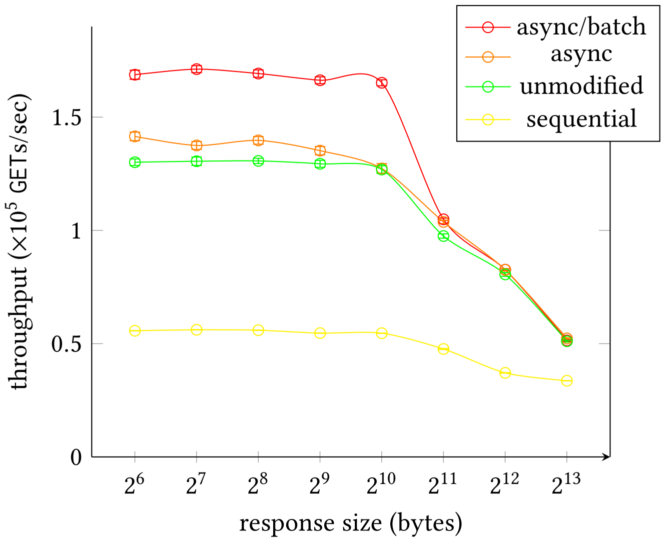

**Figure 1: Throughputs of Redis server variants with 95% confidence intervals.**

For responses of size $2^{10}$ to $2^{13}$ bytes, both systems have throughput that is on par.

The throughput of the async/batch server performs significantly better than both the unmodified and async servers for responses of size $2^6$ to $2^{10}$ bytes. For this range of sizes, the async/batch server outperforms the async server by up to 29%. This result can be understood as the gains made from batching writes to client sockets and thereby reducing system calls. Additionally, this server outperforms the unmodified server by up to 31%. This result can be understood as the gains made from executing batched writes to client sockets asynchronously using `io_uring`. Similar to the async server, throughput is on par with the unmodified server for response sizes of $2^{11}$ to $2^{13}$ bytes.

For larger response sizes, throughput is similar for the unmodified, async, and async/batch servers.

Figure 2 shows the average number of write operations to client sockets submitted to `io_uring` with a single call to `io_uring_enter` by the async/batch server. The average number of batched writes starts off around 15 writes per batch for responses of size $2^6$ to $2^8$ bytes. It then rapidly increases to about 35 for $2^9$ bytes and increases again to nearly 42 for $2^{10}$ bytes. Finally, it drops and plateaus at nearly 1.5 for responses of size $2^{11}$ to $2^{13}$ bytes. This has been made more clear by a blue line at $y = 1$.

These results help explain the similarity of the async/batch server's throughput as compared with that of the async server for the larger response sizes. The async server can be understood as the async/batch server but with a batch size always equal to one. Further, these results suggest that better

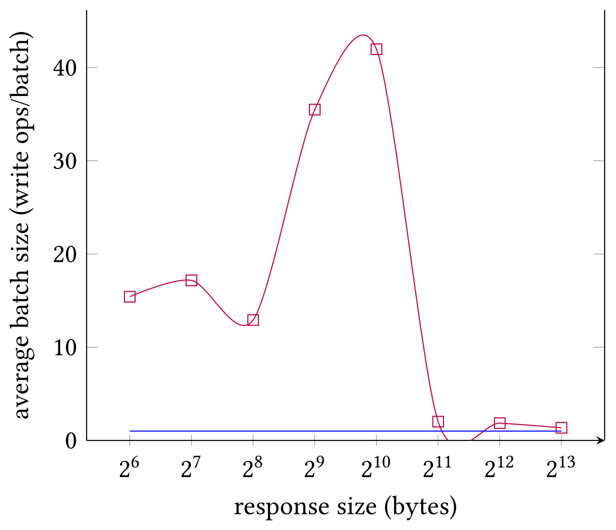

**Figure 2: Average number of write operations to client sockets batched into a call to `io_uring_enter` by the async/batch Redis server.**

performance can be achieved by controlling the number of entries submitted in each call to `io_uring_enter`.

## 6.1 Pipelining

In order to better understand how the Redis server might be affected by performing asynchronous, batched writes to client sockets, the performance of the async/batch server that has been further augmented to support pipelining of commands is evaluated. Redis uses a Request/Response protocol [11]. This means that, after a given client has submitted a request to the server, it must wait for a response prior to submitting another request. With pipelining, the client can submit several requests at once without waiting for the server to respond. The server will send responses in the order that the client submits the requests.

Enforcing this ordering constraint becomes a challenge when sending responses to clients asynchronously. In order to support pipelining, writes to a given client socket submitted to `io_uring` in a given batch and in different batches must be executed in order.

`io_uring` enables this to be done using the `IOSQE_IO_LINK` and `IOSQE_IO_DRAIN` flags for the submission queue entries [2]. The `IOSQE_IO_LINK` flag signals that a submission queue entry is a link in a chain. All sequential submission queue entries with this flag set and the following entry constitute a chain. A chain is executed in order from head to tail. This flag is used to ensure that writes to a given client's socket submitted in a single batch are executed in order. The `IOSQE_IO_DRAIN` flag signals that all submission queue entries ahead of the current entry in the queue must be completed before any entries behind the current can begin being executed. This flag is used to ensure that writes to a given client's socket submitted in different batches are executed in order.

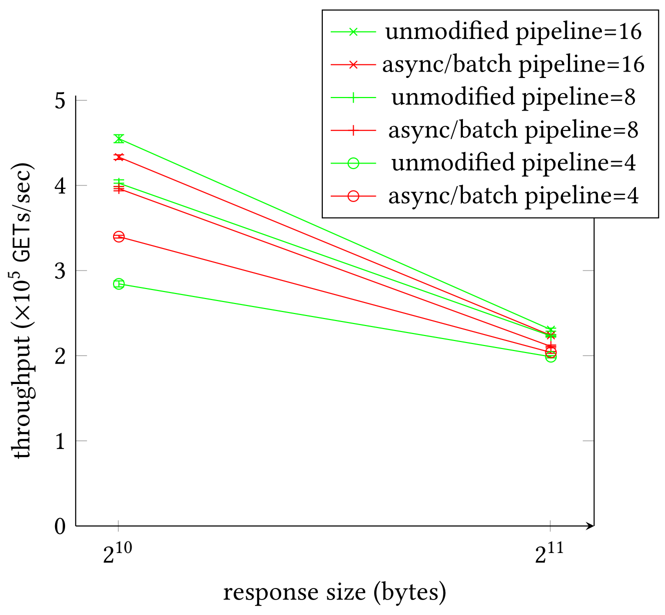

**Figure 3: Average number of writes to client sockets submitted to kernel using a single `io_uring_enter` system call by the Redis server.**

The throughput of the unmodified and async/batch servers are presented in Figure 3 for a variety of response sizes and pipeline depths.

With a pipeline depth of four, the async/batch server's throughput is over 19% higher than that of the unmodified server for a response size $2^{10}$ bytes. When the pipeline depth is eight, both servers have approximately the same performance. Finally, when the pipeline depth is sixteen, the async/batch server has throughput nearly 5% lower than that of the unmodified server. These results suggest that `io_uring` alone should not be used to enforce ordering of write operations to a given client's socket. The server might have better performance if it enforces ordering of these asynchronous writes by selectively submitting write operations to `io_uring`.

For any of the given pipeline depths, both servers perform on par with response sizes of $2^{11}$ bytes. This is the same phenomenon observed in Figure 1.

## 7 FUTURE WORK

In the implementation, writes are submitted to the `io_uring` framework every cycle around the server's main loop. This results in batch sizes being largely left to chance. Considering the results presented in this paper, a next step in this work is to evaluate the performance when submitting batches at regular intervals or after some threshold batch size has been reached. Additionally, batching will likely result in higher latencies and it would be interesting to investigate how batching affects these values.

In the implementation, the Redis server was not adapted to read from client sockets using `io_uring`. The challenge in performing asynchronous reads of client sockets using `io_uring` is in determining where to store the read data. When a given client's socket is ready to be read from, the Redis server reserves space on a buffer owned by the client and reads as much data as will fit in the reserved space. Performing this using `io_uring` first requires allocating a buffer with specific alignment constraints. Then, a read operation of the client's socket must be submitted to `io_uring` such that the result is written to the previously allocated buffer. This operation should be submitted prior to the point at which the client's socket becomes readable. Once the read has occurred, the server must then wait until it can retrieve the read data from the `io_uring` completion queue. For efficiency, the server might want to keep a linked list of buffers per client in order to avoid copying the data to a different location.

## 8 CONCLUSION

In this work, the Redis server is adapted to make use of Linux's `io_uring` framework in order to write responses to Redis clients. Through this adaptation, the performance benefits that can be gained using asynchronous writes to client sockets with fewer system calls are examined. It is shown that, by using `io_uring`, the maximum throughput of the Redis server can be increased by as much as 31% as compared to the unmodified Redis server. This work demonstrates what performance gains can be achieved using the `io_uring` framework in a high performance, system intensive scenario.

## 9 ACKNOWLEDGMENTS

I would like to acknowledge Professor Benard Wong, who provided me much help and encouragement.

## 10 AVAILABILITY

Server code can be found (here) and benchmarking code can be found (here).

## REFERENCES

[1] [n. d.]. Redis Clients Handling. https://redis.io/topics/clients. ([n. d.]). Accessed: 2020-12-08.

[2] 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf. (2019). Accessed: 2020-11-19.

[3] 2019. *White paper: Introducing 200G HDR InfiniBand Solutions.* Technical Report 060058WP Rev 1.4. Mellanox Technologies.

[4] 2020. DB-Engines Ranking of Key-value Stores. https://db-engines.com/en/ranking/key-value+store. (2020). Accessed: 2020-12-09.

[5] Suparna Bhattacharya, S. Pratt, Badari Pulavarty, and J. Morgan. 2010. Asynchronous I/O Support in Linux 2.5.

[6] Jonathan Corbet. 2017. KAISER: hiding the kernel from user space. https://lwn.net/Articles/741878/. (2017). Accessed: 2020-12-08.

[7] Jonathan Corbet. 2020. The rapid growth of io_uring. https://lwn.net/Articles/810414/. (2020). Accessed: 2020-11-19.

[8] D. Gruss, Moritz Lipp, M. Schwarz, Richard Fellner, Clémentine Maurice, and S. Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.

[9] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 295–306. https://doi.org/10.1145/2740070.2626299

[10] Redis Labs. [n. d.]. Introduction to Redis. https://redis.io/topics/introduction. ([n. d.]). Accessed: 2020-12-08.

[11] Redis Labs. [n. d.]. Using pipelining to speedup Redis queries. https://redis.io/topics/pipelining. ([n. d.]). Accessed: 2020-12-08.

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

[13] Xuecheng Qi, Huiqi Hu, Xing Wei, Chengcheng Huang, Xuan Zhou, and Aoying Zhou. 2020. High Performance Design for Redis with Fast Event-Driven RDMA RPCs. In *International Conference on Database Systems for Advanced Applications.* Springer, 195–210.

[14] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10).* USENIX Association, USA, 33–46.

[15] Wenhui Tang, Yutong Lu, Nong Xiao, Fang Liu, and Zhiguang Chen. 2017. Accelerating Redis with RDMA Over InfiniBand. In *International Conference on Data Mining and Big Data.* Springer, 472–483.