Improving Throughput of BigData Applications

Janardhana Reddy Naredula[#1]

Walmart Labs, Bangalore, India

[1]naredula.jana@gmail.com

Abstract **:** The paper describes various performance problems and solutions to improve throughput of BigData Application like Redis, Kafka, memcache, Cassandra, ElasticSearch,..etc. Most of the solution to the problems are achieved by some of the techniques like bypassing linux kernel, minimizing system calls, efficiently using the multi core machine using asynchronous programming, one thread per core, DPDK, .. etc. Modern machines are very different from those of just 10 years ago. They have many cores and deep memory hierarchies (from L1 caches to NUMA) which reward certain programming practices and penalizes others, Unscalable programming practices (such as taking locks) can devastate performance on many cores. Shared memory and lock-free synchronization primitives are used in solving some of the problems.The paper was concluded with the test prototype of Redis with efficient network path that resulted 37X perf improvement over the baseline.

Keywords: Transactions Per Second(TPS), AppendOnly Log, Mutation operations, Non-mutation operations, Instructions Per Cycle(IPC) [1], Inter Component Communication(ICC), Data Plane Development Kit(DPDK), Big Data Application.

1. Introduction

BigData Application like Redis, Kafka, Cassandra, ElasticSearch, RocksDb .. etc. are mainly used for storing and retrieving Key-value store. It contains two types of operations namely Non-mutation and Mutation operations. Non-mutation operations includes retrieving, searching data. Mutation operations includes writing, updating data to mostly immutable files. The file operation on immutable files can be synchronous or asynchronous. Non-mutation operation does not involve disk io if the data is cached, so the TPS of non-mutation operations is relatively higher when compared to mutation operation.

Common Design Patterns in BigData Applications

Following are common design patterns used in big data applications like Redis, Kafka, Cassandra, Elasticsearch, RockDB, ..etc. Based on these patterns as assumptions rest of paper was constructed.

1) Most of the input/output was done using Immutable files only.
2) Four different types of file operations are involved:
   a) Appending data to translog file as part of write request.
   b) Reading immutable files sequentially as part of read/write request.
   c) Flushing files to disk periodically.
   d) Reading from one immutable files and creating new immutable file periodically as part of housekeeping task.

   Here type-a, type-b are high priority when compared to the rest of the two types since it impacts the latency of user requests, type-c and type-d can be done as part of housekeeping with less priority.
3) Linux system call part of posix api's are used to communicate with OS for networking and file computations.
4) The machines used are usually large with multiple core like 72, and a large amount of RAM like 256G, and good network and disk speeds.

2. Problem Statement

Design of BigData applications on the linux system as performance limitations due to generic api from linux kernel. The following are the bottlenecks from the point of linux system:

1. **Network IO path:** Syscall Interface with OS to receive/send network messages need lot of cpu cycles. In the current linux system BigData Applications uses epoll, read, write system calls to receive/send the packets. Lot of cpu cycles are consumed in executing these system calls for each message, due to this TPS will be lowered.
2. **Disk Path:**
   a. **Write Path**: Syscall Interface with OS to flush the append log changes during mutation operation is a synchronous operation, and application need syscalls to communicate with OS. The syscalls can be write, mmap, fsync, msync,..etc.
   b. **Read Path:** This is a synchronous operation, and disk reads are lot slower when compared to memory read, due to this user level threads will get blocked.

user level threads will be blocked if there is not sufficient memory, due to this application calls fadvise calls to evict the memory.

3. **Multi-core machine inefficiency:** Multi-threaded code does not effectively make use of modern multi core machine because of the following reasons :
    a. **Synchronous Operations:** Most of the operations are blocking in nature, and need multiple threads per cpu core because threads blocks due to synchronous operations. having multiple threads will lead to context switches. context switches and system calls slows down the cpu speed, or bring down the Instruction Per Cycle(IPC[1]).
    b. **Implementation in High Level language**: Application like Cassandra, kafka, Elasticsearch are written in high level languages like Java. High-level languages such as Java and similar modern languages are convenient, but each comes with its own set of assumptions which conflict with the performance requirements.

3. Solution: Efficient Diskpath without kernel Change:

Direct IO: With DirectIO pagecache of kernel can be bypassed. Here the scheduling and buffering of pages will be done at the user space. By doing the scheduling and buffer implementation in user space as the following advantages:

A. Page cache inefficiency will be removed: some examples are 1) if the application need only 100 bytes but kernel will get 4k in to the memory. 2) what pages need to evacuate and what pages need to retained will be efficient in user space.
B. system calls will be minimized.
C. scheduling suitable for the application will be used.

The disadvantages is application need to implement entire logic.

4. Solution: Efficient Diskpath with kernel Change (Non-Posix API)

Current File Write Path: Fsync and write system calls combination is first method to implement Durability in the Database. Second Implementation method is using mmap and msync syscalls. Here we cover only the immutable writes. Second method is more efficient then first.

Current File Read Path: File Read using read system call is first method. Second method is using mmap system call and reading from memory directly. second method is efficient since it decreases number of system calls if the file fits in memory. In both the methods fadvise system call can be used to evict the un-required pages from memory to create sufficient free memory or to make the page cache of OS efficient.

A. Problems in the current Posix File API:

1. Calling synchronous Fsync system call for every mutation operation or at the end of few batched operations. The system call is expensive because of switch from userspace to kernel and it is a synchronous call.
2. Absence of low cost notification from OS to the userspace and vicerversa related to the amount of data flushed to disk.
3. Do not have priority among the writers. Most of the database like cassandra, Elasticsearch, ..etc has two types of writes, one is writing to translog that need at high priority and other is low priority file merging as part of housekeeping.
4. Explicit Calling of fadvise syscalls to evacuate pages of files that are needed from memory to create free memory.
5. Lazy write wastes the disk bandwidth. It is not suitable for immutable file writes that will be written only once.

B. Algorithm using New Non-Posix File API:

Shared memory region is created using ioctl call for the file by app to communicate between user level thread and OS. Shared memory for each open file is divided into two sections. “User to OS” section updated by user space thread and read by the OS. *OS to user* section updated by OS and read by user level threads. Below are the contents of the shared memory for each opened file:

- User to OS section:
    - Last-byte-written: This gives offset of last byte written to the file by the user. Data can be written using write call or using mmap method. This is an indication to OS during flushing of file content to disk. Since it is an immutable file, so this value will be continuously increasing.
    - Last-byte-not-needed: This gives indication to OS to free the pages from the pagecache till “Last-byte-not-needed”. This avoid calling fadvise syscall by the user. OS can scan this offset whenever there is a memory pressure or during housekeeping.
    - Last-byte-needed: This is an indication to OS for the readonly file. The read-ahead code will pick this and get the pages from the disk accordingly. If page is not present in memory then there will be page faults or blocking read calls that will slow down the application.
    - file Priority: priority of the data to flush. If there are multiple immutable files across the system, then

priority helps to flush the data in the priority order. Application set priority as high if it wants to flush the data immediately as soon as possible even if the data size is small. This property is very useful, with the posix api application usually calls fsync when the data size is sizable, but in the Non-posix changing the priority will make the OS picks the data as soon as possible without any system call at the highest granularity, this as huge impact on efficiency and latency.

- OS to user Section:
  - Last-byte-flushed : This is last byte or highest offset in the file flushed to disk by the OS. Here file is assumed to be immutable file.

C. Techniques used in the above Algorithm:

The non-posix file api address the above problems with the following techniques:

1. Almost Zero system calls to increase IPC: After initialization, Non-posix API tries to use almost zero system calls. This is acheived by using the shared memory communication between user level threads and kernel. Along with this polling method is used on both sides to avoid wait or sleep.
2. Early write to optimize disk io: Early Write means writing to disk immediately after complete writing block size data into memory. linux being generic system, it does not do early write because pages of the file can be overwritten in the near time and too frequent communication with OS need system calls. But in this design most of the files are immutable. Waiting does not help, flushing the data immediately after crossing block size will help in improving the disk io. OS will uses polling and shared memory to detect and write to disk as soon as possible.
3. Prioritised IO to decrease latency: Not all writes or reads need to be done immediately, but need to be done eventually. There are two types of tasks, first type is io waiting by the client request, second is regular housekeeping tasks like merging immutable files,..etc. so first task need to be completed as soon as possible to reduce the latency, and at the same time second tasks need to processed whenever the disk is idle without waiting from the user app or from the OS. This design pattern suits for immutable file and reading sequentially.
4. Evacuate not-required pages Efficiently: Linux evict the pages based on the LRU algorithm, but LRU does not suitable for all cases. Suppose if a page in a file is read, and no longer needed in the near future, in such cases application can signal the OS using the fadvise calls. But fadvise is expensive since it needs to be called regularly. This is addressed using shared memory communication.
5. Disk IO bandwidth bottleneck*:* In the write path, the API will be more efficient if disk io bandwidth is not the bottleneck, this can happen in the following cases:
   a. If the speed gap between volatile memory and disk/Non-volatile memory decreases.
   b. If there are a large number of disks attached to machine, so that overall disk bandwidth increases.
   c. If the user space thread as less writing when compared to other computations like read,..etc.
   d. asynchronous disk io requirement is less when compared to housekeeping disk io.

   In all the above cases, bottleneck will be shifted from io speed to the communication between user and kernel threads.
6. *Hugepage advantages:* In write path, mmap can be used to avoid write system call. but when the application touches a single byte in file buffer, OS need to flush entire 2M page this will be huge penalty. With new API only the smallest block like 512(size of block on physical device) can be flushed, this is possible with the *LastByteWritten* from shared memory as mentioned in the previous section, this is not possible in existing posix api due to absence of file offset in mmap.
7. ReadAhead : This software construct is already present in the OS to speed up user read request. Here the some differences:
   a. OS detects automatically based on the read pattern, but here the app knows it is going to read all the content of file once.
   b. Read-ahead need lot of free pages and disk bandwidth, so read-ahead need some coordination with evacuation-technique for free pages and prioritised read requests along with other io requests.

D. **Summary of changes needed in Application**

The following are the changes needed in the application to implement non-posix API:

1. Setting shared memory section for file: Ioctl syscall is used to set up the shared memory section. This is done during file opening.
2. ioctl is used to start and stop the polling inside the kernel. This can be done during initialization or periodically.
3. Fdatasync, mysnc, fadvise syscalls are completely avoided. Both user level threads and Kernel will be

polling and updating the shared memory to communicate to other side.

With Non-posix API, user level threads will be populating into volatile memory using mmap without any slowdown from system calls or blocking from OS. On the other hand, kernel threads will be flushing dirty pages in a priority manner into non-volatile disk without any interruption. Here both kernel and user level threads will be running in parallel with shared memory as the communication between them, both the threads will be polling, due to this IPC will be high.

E. All put together:

Early disk writes, read ahead advance, memory evacuation, priority io, regular polling, almost zero system calls, Hugepages usage will make entire io faster and efficient.

The above api will make application threads and OS threads to execute in parallel with almost zero system calls.

G. Prototype Test Results: Yet to publish.

5. Solution: DPDK for efficient Network Path

DPDK is proven and as delivered better network throughput in Networking workload like Network Function Virtualization(NFV). In BigData applications, Network path is one of the major bottlenecks especially for non-mutation operations. To transmit the packets at a high rate, the application need to bypass the system calls and kernel tcp/ip. DPDK is implemented as a user space library, It contains userspace tcp/ip stack and uses techniques like shared memory, batching, hugepages, continuous spin of threads. DPDK route the network packets efficiently from NIC to BigData applications and vice versa. With DPDK the networking stack of OS is completely bypassed.

A. Advantages of DPDK

The DPDK library in user space can receive/send the packet at high rate using poll mode driver inside the kernel, the following are the advantages:

1. Zero system calls: The interface between proxy and OS, and between proxy and server does not involve any system call.
2. Zero memory copy: The buffer from NIC-proxy-server and vice versa are done through shared memory inside the huge pages.
3. Uses of HugeTLB pages and bulk copy: Buffers are pre allocated inside the Huge page memory, and transmitted in bulk instead of one packet each.

B. Disadvantages and Mitigation

At low load, poll-mode driver inside the kernel and userspace will consume cpu. This can be mitigated by converting polling mode into blocking mode when the load is less dynamically.

6. Prototype of Shared memory Vs Socket

Test Description: The Ring buffer inside the shared memory is used to exchange messages between two entities. Shared memory will also be used to receive/send the network packets from DPDK instead of socket interface. The entity can be process, thread from within the same process , threads from two different process , between OS and user space threads.

Advantages of shared memory interface with DPDK compare to linux sockets are:

1. System calls are not needed to exchange messages.
2. Memory copy is not needed when compared to sockets.
3. Bulk copy or batching of messages can be possible in the shared memory.
4. one thread per CPU core will be used instead of blocking system calls. This decreases latency and increases TPS.

A. Prototype using Redis to Demo the impact of DPDK vs Socket:

Redis is superior in performance when compared to other Big Data applications like cassandra, kafka, elastic search ,..etc because of the following reasons:

1. Redis uses asynchronous programming model with one thread per cpu.
2. Redis is written in C, so the memory and buffer management is very efficient.

"Redis with Socket"is a baseline code. "Redis with shared memory" is patched by removing socket interface and added shared memory interface. "Redis with Socket" versus "Redis with shared memory" is compared with various degree of batching of requests[3]. "Redis with socket" is unmodified version receives the packet using the socket interface from linux kernel , "Redis with Shared memory" is a modified version to measure the impact of receiving the packets using DPDK instead of socket. In the below table-1, Shared memory version has out performed the socket version by a 37X times with one request is sent per network packet(i.e batch=1). Shared memory as variation of with and without copy, without copy means buffers from shared memory is reused to avoid the extra memory copy. Zero copy as shown improvement on top of the shared memory improvements. Zero copy is not possible with the socket. Other implementations of shared memory are LMAX[6].

TABLE I

SHARED MEMORY VERSUS SOCKET COMPARISON

SNO	Type	TPS	Description
1	Redis with Socket (batch=1)	54K	**Batch=1** : The ratio of difference is High (1:37), because of lot of round trips or less batching in socket.
	Redis with Shared memory (batch=1)	2000k	
2	Redis with Socket (batch=10)	576k	**Batch=10** :The ratio of difference is Medium (1:5) ,because round trips is reduced by 10.
	Redis with Shared memory (batch=10)	3000k	
3	Redis with Socket (batch=50)	1800k	**Batch=50** :The ratio of difference is relatively less(1: 1.8), because round trips is reduced by 50.
	Redis with Shared memory (batch=50)	3300k	
4	Redis With Shared memory + zero copy	TODO	
5	Server-client With shared memory (batch=10)	3.5M	Better with Zero copy
	Server-client With shared memory + zero copy(batch=10)	4.5M	
6	Server-client With shared memory (batch=50)	8.3M	Better with Zero copy
	Server-client With shared memory + zero copy(batch=50)	12.5M	
batch = number of requests per network packet.			

B. **Analysis of Test Results**

1. Shared memory performs better in batch=1 when compared to batch=10: The more the message exchanges the better the improvement, this is due to savings from system calls. In batch=1, there will be 10 times more system calls when compared to batch=10.
2. Zero Copy: Reusing the buffers without freeing the buffer causes less memory copies. Memory copy is an expensive operation, due to this TPS is higher.

7. Solution: Asynchronous programming

A. **One thread Per CPU core**

Programming a server which uses a process (or a thread) per connection is known as synchronous programming, because the code is written linearly, and one line of code starts to run after the previous line finished.The solution, which became popular in the following decade, was to abandon the inefficient synchronous server design, and switch to a new type of server design the asynchronous, or event-driven, server. An event-driven server has just one thread per CPU. This single thread runs a tight loop which, at each iteration, checks, using epoll for new events on many open file descriptors, e.g., sockets. For example, an event can be a socket becoming readable (new data has arrived from the remote end) or becoming writable (we can send more data on this connection). The application handles this event by doing some non-blocking operations, modifying one or more of the file descriptors, and maintaining its knowledge of the state of this connection.

8. Other Optimizations

A. **NUMA Awareness**

NUMA awareness can make cpu cache friendly. There will be a lot of churn of data inside the cache. With proper pinning of threads to the appropriate core will avoid the pollution of L2, L3 caches.

B. **Java Off-Heap Memory**

Since most of the NoSQL are implemented in Java, Performance of JVM will be critical. One of the problems in Java is Memory management during the processing of request and response, there will be a lot of memory buffers are created and destroyed during the life cycle of request response cycle, this creates extra pressure on the JVM garbage collector, and causes periodic cpu and memory spikes. This can be minimised by using the reusable off-heap memory. Off-heap memory can be used for the following two purposes:

1. The off-memory also well connects with the shared memory described in the earlier section. with this the network requests coming from outside the jvm can receive/transmit without memory copy by the off-heap memory module.
2. off-heap memory can used instead of jvm heap, so that the to java objects in the request-response can be reused.

9. Putting All together

Putting all the above optimizations together and applying on NoSQL databases like Redis, Kafka, Cassandra, ElasticSearch,..etc. This as shown in. Figure-1. optimizations are represented in oval shape. The following are advantages when compared to the non-optimized version:

1. **Network Path:** DPK and network proxy are used for network path as shown in the figure. NoSQL databases uses network proxy instead of socket layer.

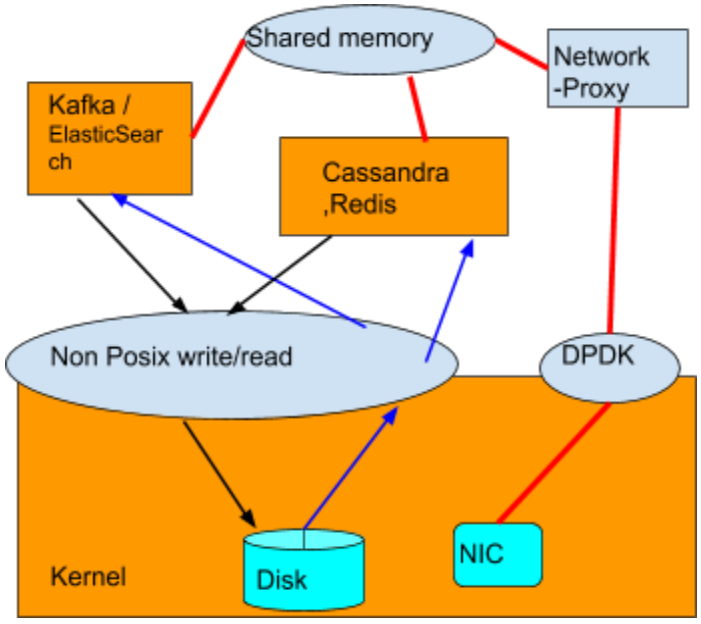

FIGURE-1

2. **Efficient Disk Path with Kernel change:**
   a. **Disk Write Path:** Cassandra, Kafka can flush all files like append log, commit log, sstables using non-posix api. Both applications do not need any system calls to flush the data. Huge TLB pages can be efficiently do mmap the commit logs. OS can flush portion of large page instead of complete page with the help of offset inside the fsync shared memory. Priority of flushing helps Cassandra to flush commit log before sstable files.
   b. **Disk Read Path:**Read path can be improved by caching the hot pages in the memory without depending on linux pagecache, To do this unwanted pages need to be removed from the memory using fadvise.
3. **Efficient Disk Path without Kernel change:** Using Direct IO and without using pagecache, Here the scheduling and buffering of IO will be done at the user space.
4. **Inter Component Communication(ICC):** Shared memory can be used to communicate between proxy, Kafka, Cassandra and other app without any system calls, this can be done with zero copy.
5. **Asynchronous Programming:** The application needs to be event driven so that one thread per CPU core can be implemented to make cpu efficient.

10. Summary

Improvements in Disk path, network path and communication between the various components of NoSQL Databases or BigData Applications are described in this paper.

Inefficiency in write disk path, read disk path, network path , communication between components , Asynchronous programming are described in this paper. The Root cause and corresponding solutions for the above are summarised below:

1. **Almost Zero system calls:** Non-posix Fsync for write path, Shared memory for ICC and DPDK for network path avoids system calls completely. Due to zero system calls IPC and TPS of the components will improve drastically. There will be a small percentage of system calls during file or socket setup.
2. **Efficient ICC:** Shared memory and DPDK have Zero buffer copy. Polling mode in DPDK, shared memory and Non-posix fsync makes communications faster and efficient.
3. **Parallelism between user level and kernel level threads:** user level and kernel threads will be executed in parallel without blocking, this is due to polling and shared memory communication.
4. **Multi Core Friendly:**
   a. Polling mode without system calls is well suited for multi core when compared to single core.
   b. NUMA friendly: polling mode and less system calls makes more NUMA friendly and cpu cache friendly.
   c. HugeTLB can be efficiently used with Non-posix fsync, shared memory and DPDK.
   d. one thread per core with the asynchronous programming.

References

[1] Instruction Per Cycles paper , https://github.com/naredula-jana/Jiny-Kernel/blob/master/doc/Perf_IPC.pdf.

[2] JinyKernel: https://github.com/naredula-jana/Jiny-Kernel/.

[3] Shared Memory Implementation: https://github.com/naredula-jana/PerfTools/tree/master/SharedMem

[4] DPDK : https://software.intel.com/en-us/articles/dpdk-performance-optimization-guidelines-white-paper

[5] DPDK : http://www.dpdk.org

[6] LMAX disruptor: “https://lmax-exchange.github.io/disruptor/