Request-aware Cooperative I/O Scheduling for Scale-out Database Applications

Hyungil Jo[1], Sung-hun Kim[1], Sangwook Kim[1,2], Jinkyu Jeong[1], and Joonwon Lee[1]

[1]Sungkyunkwan University
[2]Apposha

Abstract

Interactive data center applications suffer from the tail latency problem. Since most modern data center applications take the sharded architecture to serve scale-out services, a request comprises multiple sub-requests handled in individual back-end nodes. Depending on the state of each back-end node, a node may issue multiple I/Os for a single sub-request. Since traditional I/O scheduling operates in an application-agnostic manner, it sometimes causes a long latency gap between the responses of sub-requests, thereby delaying the response to end-users. In this paper, we propose a request-aware cooperative I/O scheduling scheme to reduce the tail latency of a database application. Our proposed scheme captures request arrival order at the front-end of an application and exploits it to make a decision for I/O scheduling in individual back-end nodes. We implemented a prototype based on MongoDB and the Linux kernel and evaluated it with a read-intensive scan workload. Experimental results show that our proposed scheme effectively reduces the latency gap between sub-requests, thereby reducing the tail latency.

1 Introduction

Modern data center applications provide highly interactive services to end-users. In interactive applications, providing a consistently low latency in responding to user-requests is one of the important concerns of service providers [7]. In a decade, several studies have conducted to reduce tail latency on individual building blocks of data center applications, such as data center networks [10, 9], application protocols [17, 19], request scheduling [14], and layered I/O stacks [18, 16]; most of these relieved the tail latency problem directly or indirectly. However, such studies do not carefully consider the cases that latency of a request involves causally-related multiple I/Os on multiple nodes.

Database applications such as MongoDB [2] and Cassandra [12] are representative data center applications because they provide consistent data stores for other applications such as web servers. To cope with the growth of database size and user demands, modern database applications support scale-out architectures, such as sharding [5]. On a sharded database, a request such as a search query splits into multiple sub-requests that are handled in a group of servers [8], and the system will wait for the completion of whole sub-requests. Each server handles an assigned sub-request through a number of internal behaviors such as cache lookup, index tree traversal, and interaction with the kernel for issuing I/Os. As a result, the original request must wait for the completion of all internal behaviors involved in all sub-requests. Since internal behaviors are not determined until a request arrives, this uncertainty can lead to latency fluctuations.

With an application study reported herein, we found several conditions that exacerbate tail latency in database applications (§3).

- An original request finishes only after receiving completion messages of all sub-requests.
- A node issues multiple I/Os for a single sub-request, and the response of the sub-request can be delayed by the contention with I/Os caused by other concurrent sub-requests handled in the same node.
- Application-agnostic I/O scheduling can delay the response of a sub-request. For example, fair I/O bandwidth allocation in the block layer significantly increases the latency of each sub-request since it only considers fairness among the threads while ignoring fairness among the user requests.

Based on the observations, we conclude that, if I/Os on multiple nodes caused by one request are *cooperatively scheduled* by reflecting the application's semantic, it relieves the tail latency by avoiding the above conditions. We choose the *request arrival order* as a useful application semantic that can be exploited for I/O scheduling.

Thus, in this paper, we propose a request-aware cooperative I/O scheduling scheme to reduce the tail latency of requests. The proposed scheme comprises two parts. First, the proposed scheme propagates the request arrival order from the application to the I/O schedulers in the back-end nodes of an application (§4.1). Second, the I/O scheduler in each node exploits the propagated request arrival order to make decisions for scheduling I/Os. By doing so, each I/O scheduler can enforce the request arrival order in its local I/O scheduling (§4.2).

As a case study, we used MongoDB as a database application [2] and examined its read path from the front-end of the application to the I/O scheduler in an OS. Then, the proposed scheme was implemented in the application as well as in the Linux kernel. With a read-intensive scan workload, we show that the proposed cooperative I/O scheduling scheme effectively reduces the tail latency of requests.

2 Background

In modern applications such as database applications, a request such as CRUD (create, read, update, delete) is composed of a number of internal behaviors [15]. This may affect the predictability of the latencies of responding to users since the completion of an original request is delayed until all behaviors are complete. Due to the scale-out architecture of modern applications, such behaviors caused by a single request are handled at multiple nodes, and therefore, it sometimes exacerbate the unpredictability in the latency of requests. Thus, we investigate internal behaviors in MongoDB, a popular distributed database application, and the Linux kernel to find a method that can serve predictable latency of request.

2.1 Behaviors in Application

With sharding, an application splits its key space into multiple partitions and creates a number of back-end *shard*s on multiple nodes to deal with each partition. Since an application creates shards on different nodes for load balancing, an application requires a responsible entity, such as a front-end proxy (e.g., a query router) or an out-of-band coordinator, to serve transparent application services. In sharded applications, depending on the type of request (e.g., scan), a request is divided into a number of sub-requests, which are distributed into multiple shards as shown in Figure 1.

Depending on the state of each sub-request, a shard can have different behaviors. For example, database applications commonly index their blocks in storage by using index trees such as B-tree or LSM tree [13]. When an application gets a request to retrieve a record and it is not in the cache, it traverses an index tree to find the block location of a record. While traversing an index tree, an application accesses several internal nodes of the index tree, and it may issues several I/Os if accessing nodes are not in the cache. While I/Os seem individual from the perspective of the I/O layer (e.g., the I/O stack in the kernel), those are correlated to a request from the perspective of the application.

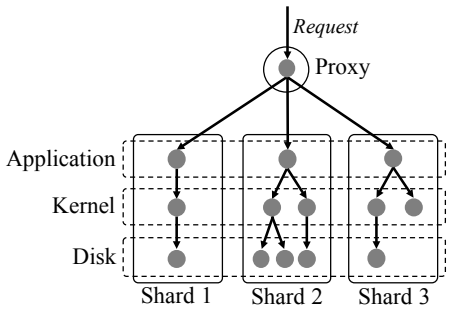

Figure 1: **Causal relationships in a request:** depending on the state of each shard, the length of the critical path in a sub-request is decided. In this figure, a sub-request on shard 2 has the longest critical path.

When a sub-request on each shard is completed, a shard transfers the result to the proxy. When the results of all sub-requests arrive at the proxy, the proxy aggregates them and then responds to the original request.

2.2 Behaviors in Kernel

In each shard, the OS kernel handles I/Os on behalf of an application. A typical I/O stack in modern OSes is composed of a number of independent layers [11]. If a read I/O hits in the kernel caching layer, it is served without accessing storage. Otherwise, a file system submits a block read request to the block layer, and the block layer makes a read I/O request and admits it into a request queue of a block device. Lastly, an I/O scheduler dispatches an I/O request from a request queue to the block device depending on its local policy, such as earliest deadline first or fair scheduling [1].

However, it does not directly result in low latency of the original request. This is because an I/O scheduler does not recognize the scale-out structure of the application and thus it misses knowledge of correlations across I/Os. Accordingly, the I/O scheduler merely handles I/O requests on a best-effort basis in the layer and sometimes cannot make the best decision to minimize the latency of a request. For example, individual I/O requests for traversing an index tree can be mixed and re-ordered depending on the I/O scheduling policy.

3 Key Insights

By searching the behaviors of MongoDB and the Linux kernel, we observed two key causes of latency fluctuation

in a request.

Causal relationship: In sharded applications, since each request is handled according to the aggregator-leaf model, a request and its sub-requests are causally related. Similarly, a sub-request in a back-end node may issue a number of I/Os; thus, a sub-request and its issuing I/Os are also causally related. Causal relationship implies that a parent must wait for completion of children tasks (Figure 1). Furthermore, by the transitive law, a request and I/Os issued by its sub-requests have causal relationships.

Variable critical path: An application and the kernel commonly have their own cache. A request percolates down to lower layer only if a cache miss occurs. The critical path length of a request is decided based on the state of the cache. Since the state changes over time and in response to running workloads, an application cannot determine the completion time of a request. Moreover, some internal behaviors such as index tree traversal can incur additional I/Os in the critical path.

Based on these observations, we found that completion time of a request is determined by the sub-request that has the longest critical path. To reduce the tail latency of requests, we suggest a strategy for an I/O scheduler that considers such internal behaviors of an application. Briefly, the strategy forces that scheduling of I/O requests to follow the request arrival order captured in the application, and the I/O scheduler schedules I/O requests in the same causal relationship in a batched manner.

4 Design and Implementation

To enforce our strategies for I/O scheduling in distributed nodes, the context of a request should be propagated through the full parallel I/O path of the application and the OS kernel. To this end, the proposed cooperative I/O scheduling scheme is composed of two parts: *request context propagation* and *cooperative I/O scheduler*. We summarized the roles of these two components as follows:

- **Request context propagation** is a mechanism that delivers the context of a request, especially *causal relationships* (Figure 1), from an application's proxy to I/O schedulers in the shards.
- **The cooperative I/O scheduler** in each shard schedules I/O requests by reflecting the arrival order of the original requests at the proxy.

We used MongoDB [2] and the Linux kernel to implement our scheme. Request context propagation components are implemented into both MongoDB and the kernel, and the cooperative I/O scheduler was implemented as an I/O scheduler in the kernel. In the rest of this section, we describe implementation details of the two components.

4.1 Request Context Propagation

MongoDB has a number of components to support sharding. The front-end proxy (hereinafter, "proxy") in MongoDB, named *mongos*, receives requests from clients and sends sub-requests to back-end shards (hereinafter, "shards"). The proxy waits for completion of whole the sub-requests and then aggregates results of sub-requests to return the results to the clients. A configuration server stores metadata that describes the layout of the partitioned key space and the sharding architecture. Each shard has an instance of *mongod* that actually handles operations in an original request, such as CRUD. A mongod process is multi-threaded, and each thread is in charge of handling one request. Each component of MongoDB operates independently and interacts with other components by exchanging messages.

When a request arrives, mongos interprets it and encodes a request into a message, which is of a form that each shard can handle. Then, mongos determines which shards are appropriate and forwards the message to each of these shards.

We add a module to mongos that captures the request arrival order and embeds it in each message. When a request arrives, a module creates a *request-context object* and assigns a unique *request id* to the request. A module maintains a request-context object until a request is completed. When mongos creates a message, it includes the request id in the message. By doing so, mongos can propagate the request arrival order to the shards.

Once the message arrives at each shard, mongod interprets the message and invokes the appropriate service such as query processing. While a thread in mongod handles a message, I/Os should be accompanied with the corresponding request id of the message. To this end, we implemented two system calls, `ctx_begin()` and `ctx_end()`. The former is called with a request id before handling a message, and the latter is called when message processing is complete. In mongod, I/Os issued between the two system calls are tagged with the corresponding request id. As a result, the request context can be propagated to the kernel. The kernel allows only one request context to each thread at once since each thread handles an assigned sub-request synchronously. Whenever the form of an I/O is changed (e.g., from a bio to an I/O request), the request id is also transferred to the changed form until the I/O reaches the I/O scheduler.

4.2 Cooperative I/O Scheduler

The goal of our cooperative I/O scheduler is to enforce the order of I/O scheduling to follow the request arrival order captured at the proxy. Since each I/O request is tagged with a corresponding request id, our scheduler can schedule I/O requests in the order of the request ids. We implemented our cooperative I/O scheduler based on

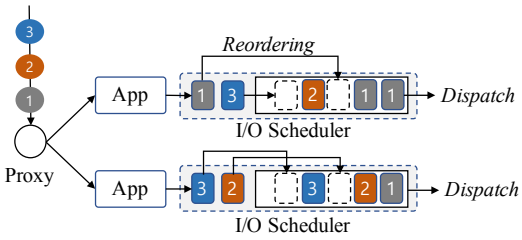

Figure 2: **Request-aware cooperative I/O scheduling:** circles and rounded rectangles represent user and I/O requests, respectively. The number in each circle indicates the request arrival order.

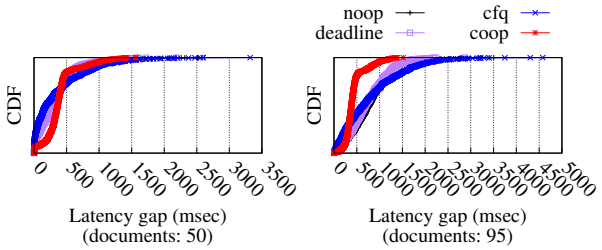

Figure 3: **Latency gap between sub-requests:** each value is the difference between the minimum and the maximum latency of sub-requests.

the noop scheduler of the Linux kernel. While the original noop scheduler schedules all I/O requests in a first-in first-out (FIFO) manner, the cooperative I/O scheduler reorders the received I/O requests according to request id, as depicted in Figure 2.

Basically, I/O requests in the request queue are in the form of a linked list and are isted in order of request id. When a new I/O request arrives, it should be inserted in the correct position, between the last I/O request having the same request id and the first I/O request having a larger request id. This search operation may increase the queueing time. Since I/O requests irrelevant to applications are assigned a default request id (i.e., 0), the search operation can also waste time in traversing I/O requests with the default request id.

To avoid unnecessary traversal, the cooperative I/O scheduler maintains a lookup table to track outstanding request contexts. Each entry contains a request id, a pointer to the last I/O request in the same request context, and several statistical information. When an I/O request arrives, the cooperative I/O scheduler looks up the table and find an entry which has the matched request id and then inserts the arriving request at an appropriate position in the request queue by following the pointer to the entry. An entry of the lookup table is allocated when an I/O request with a new request id arrives and is freed upon the completion of the last I/O request which is implied by the `ctx_end()` call. Since maximum size of the lookup table is decided by the application's parallelism level (in our case, it is less than 500 in each shard), negligible additional memory overhead is incurred by using the cooperative scheduling scheme.

5 Evaluation

To evaluate the request-aware cooperative I/O scheduling scheme, we used four VMs on two physical machines to run mongod, and two physical machines to run a mongos and a client. Each machine had two Intel Xeon E5-2650 CPUs, 32 GB of memory, and two SAS SSDs, and the machines were interconnected by 10 Gbit Ethernet. Each VM had two virtual cores, 2 GB of memory and a dedicated SSD for database storage. Physical resources were not overcommitted in each physical machine to avoid resource contention. For prototyping, we used Linux 4.8.2 and MongoDB 3.2.10.

We defined and ran a synthetic workload using the Yahoo Cloud Serving Benchmark (YCSB) [6] as follows:

- **Scan:** The workload performs a range query that finds N documents having a value *greater-than-or-equal-to* a specific value. Note that N is a random variable ranging uniformly from 1 to 100.

A workload runs on a data set composed of forty million 1 KB documents. We ran YCSB on a client machine which generates concurrent requests by 64 client threads. For comparison, we used three I/O schedulers in the Linux kernel (noop, deadline, and completely-fair queue (cfq)). Hereafter, we denote the cooperative I/O scheduler as *coop*.

Figure 3 shows the latency gap between the first and last response of sub-requests of each request. We present two cases from our experiments: a query for 50 documents as a median case and a query for 95 documents as an extreme case. As shown in the figure, when the number of documents to fetch increases, the latency gap significantly increases in the original Linux I/O schedulers. Due to the increased number of documents, internal behaviors, such as index tree traversal and cache misses, generate more I/Os. However, our scheme is successful in reducing the latency gap by respecting the global scheduling order.

We present throughput and latencies of the scan workload in Table 1 and Figure 4, respectively. Four I/O schedulers show similar throughput because most requests are served from in-memory cache. Since applications incur I/Os when cache misses occurred, effect of I/O scheduling is notably shown in the tail latency. Deadline and noop showed moderate response times, whereas cfq aggravated the tail latency. Since cfq applies fair I/O scheduling to threads, a sub-request can be delayed due to reordering of I/O requests. The cooperative I/O scheduler, however, outperforms other schedulers in terms of

Configuration	noop	deadline	cfq	coop
Throughput (ops/sec)	147.92	150.25	145.40	150.76

Table 1: **Throughput of the scan workload.**

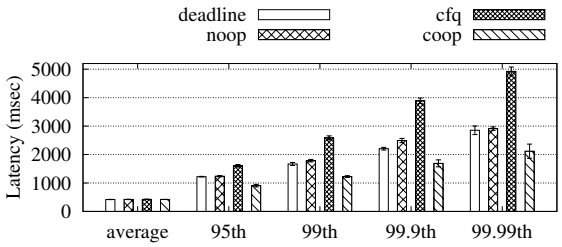

Figure 4: **Average and tail latencies of the scan workload.**

the tail latency, reducing tail latency up to 57% better than that of cfq and 32% better than those of noop and deadline. Although our scheme does not strictly co-schedule I/Os associated with the same request over the multiple nodes, respecting the global order in each node is effective in synchronizing sub-request handling.

6 Discussion

While the prototype implementation showed that the request-aware cooperative I/O scheduling is effective in reducing tail latency, the prototype implementation only considers read-oriented workloads. To make our scheme more effective for all types of workloads, we need to consider the following aspects.

6.1 Complexity in Write Path

Applications and kernel I/O stacks extensively use write merging that cause the difficulty in assigning proper request id to a merged I/O request. For example, database applications use a group commit that flushes transactional logs made by multiple threads together [3]. Each transactional log contains updates by a single update request by a user. In addition, when database applications run on a file system supporting crash consistency, another write entanglement occurs in the file system journaling layer [20, 11]. Hence, write I/Os caused by a group commit are caused by multiple requests. Assigning a request id to a merged I/O should be done carefully. Otherwise, tail latency can be aggravated because of starvation of I/Os or violation of the request order.

6.2 Background Jobs

Most applications have maintenance jobs running in the background. For example, Cassandra [12] conducts *compaction* on data files to guarantee low latency of read requests, and MongoDB makes a snapshot periodically to support snapshot-level durability [4]. Since such jobs incur heavy I/Os, I/Os associated with user requests can be affected [20, 11]. One problem is that it is difficult for the I/O scheduler to schedule background jobs since they are not initiated by user-requests. Previous work [11] suggests that prioritizes I/Os for foreground jobs from I/Os for background jobs and also prioritizes low-priority I/Os when the runtime dependencies between I/Os is found. However, this suggestion might be ineffective since it only supports two level prioritization whereas we also require mapping of background I/Os to the dependent *request context* in runtime.

6.3 Replication

Database applications usually adopt data replication for high availability. On an application with data replication, request latency can be prolonged due to the replication protocol. Applications commonly use quorum-based replication protocols for data consistency. In that case, a write request is propagated to replication nodes and then is complete when the majority return the results. For a read request, an application executes a quorum read if it is a masterless system, or just reads from a single node if it has a primary node. If the scheduling order of I/O requests from concurrent requests are stirred in any replication nodes, it can delay the completion of all requests. We believe that our cooperative I/O scheduling scheme relieves the tail latency problem in replication nodes by enforcing global scheduling order.

7 Conclusions

In this paper, we investigated the I/O path from a database application to the kernel I/O stack to understand the causes of tail latency. Based on our observations, we designed a request-aware cooperative I/O scheduling scheme and implemented a prototype in a read path. With a synthetic scan workload, we demonstrated that our prototype effectively reduces the tail latency of the workload. We plan to reflect considerations noted in Section 6 to our prototype and to search for other applications that can benefit from the request-aware cooperative I/O scheduling scheme.

8 Acknowledgements

We would like to thank reviewers for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2014R1A2A1A10049626) and by Samsung Electronics. Joonwon Lee is the corresponding author.

References

[1] Completely fair queueing. `https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt`.

[2] MongoDB for GIANT ideas. `https://www.mongodb.com/`.

[3] MongoDB journaling. https://docs.mongodb.com/v3.2/core/journaling/.

[4] MongoDB WiredTiger storage engine. https://docs.mongodb.com/v3.2/core/wiredtiger/.

[5] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[6] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*.

[7] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[8] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*.

[9] DOGAR, F. R., KARAGIANNIS, T., BALLANI, H., AND ROWSTRON, A. Decentralized task-aware scheduling for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*.

[10] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing flows quickly with preemptive scheduling. In *Proceedings of the 2012 ACM Conference on SIGCOMM (SIGCOMM'12)*.

[11] KIM, S., KIM, H., LEE, J., AND JEONG, J. Enlightening the i/o path: A holistic approach for application performance. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*.

[12] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[13] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[14] REDA, W., CANINI, M., SURESH, L., KOSTIĆ, D., AND BRAITHWAITE, S. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems (EusoSys'17)*.

[15] SHEN, K. Request behavior variations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*.

[16] STEFANOVICI, I., SCHROEDER, B., O'SHEA, G., AND THERESKA, E. sroute: Treating the storage stack like a network. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*.

[17] SURESH, L., CANINI, M., SCHMID, S., AND FELDMANN, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*.

[18] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. Ioflow: A software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

[19] WU, Z., YU, C., AND MADHYASTHA, H. V. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*.

[20] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*.