# SyncGC: A Synchronized Garbage Collection Technique for Reducing Tail Latency in Cassandra

Seungwook Han
LINE
seungwook.han@linecorp.com

Sungjin Lee
DGIST
sungjin.lee@dgist.ac.kr

Sangwook Shane Hahn
Seoul National University
shanehahn@davinci.snu.ac.kr

Jihong Kim
Seoul National University
jihong@davinci.snu.ac.kr

## ABSTRACT

Data-center applications running on distributed databases often suffer from unexpectedly high response time fluctuation which is caused by *long tail latency*. In this paper, we find that long tail latency of user writes is mainly created by the interference with garbage collection (GC) tasks running in various system layers. In order to address the tail latency problem, we propose a synchronized garbage collection technique, called *SyncGC*. By scheduling multiple GC instances to execute in sync with each other in an overlapped manner, SyncGC prevents user requests from being interfered with GC instances, thereby minimizing their negative impacts on tail latency. Our experimental results with Cassandra show that SyncGC reduces the 99.99th-percentile tail latency and the maximum latency by 35% and 37%, on average, respectively.

## 1 INTRODUCTION

Modern large-scale data center services are built over a complex distributed system so that their important requirements such as high availability and high scalability can be efficiently supported. Although managing a data-intensive system in a distributed manner has many advantages in practice, applications often suffer from unexpected response time outliers, which cause severe degradations on user experience.

Even though there can be many different factors, *long tail latency* is considered to be one of the primary sources for response time variations [4, 10, 12, 13]. In a distributed environment, when a single user request $R$ is split into multiple subrequests $subR_i$ which are processed by several distributed nodes, the response time of $R$ is decided by the longest subrequest response time. In other words, long tail latency occurs when one of the nodes where subrequests are spread out is not responding quickly over other nodes [1, 11, 14]. Since a user request can be finished only after all the subrequests are completely served by the nodes, the slowest node eventually decides the overall system responsiveness.

The fundamental solution that addresses long tail latency is to assign subrequests to nodes that are ready to serve new requests immediately [4, 8, 9]. This solution, however, is infeasible because keeping track of all the available nodes in large-scale distributed systems is impossible in practice. As an alternative solution, therefore, optimizing individual nodes so that they are able to provide consistent response times is widely studied by many researchers [7, 10, 15].

In this paper, we identify the root cause of long tail latency of a popular distributed database, Cassandra, by profiling its execution behavior from the *worst-case* latency analysis and propose new optimization techniques to improve tail latency of Cassandra. Particularly, we focus on analyzing long tail latency of writes. There have been many attempts to get rid of tail latency for reads because of their higher impact on user-perceived response times [15, 16]. However, as workloads in data centers have shifted to write-oriented one, eliminating tail latency for writes is becoming a challenging issue [1, 3, 17]. For example, for every write transaction, Cassandra creates at least three copies of replicas and distributes them across nodes, along with an additional erasure code. The write transaction is finished after the original data, the replicas as well as the erasure code are completely stored in distributed nodes. If one of the nodes is delayed, the entire write transaction has to be postponed, which leads applications to experience high fluctuation in transaction processing.

Our analysis with Cassandra reveals that high fluctuation of write requests is mainly caused by the lack of free buffer

*APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6006-7/18/08...$15.00
https://doi.org/10.1145/3265723.3265737

space in the main memory and is further exacerbated by interference with garbage collection (GC) [18]. When write requests come, Cassandra temporary buffers them in the main memory, which is managed by a Java Virtual Machine (JVM). While JVM makes it easier to develop application services in a quick manner, it occasionally triggers JVM-GC in an attempt to create free heap space by removing deleted objects and performing compaction. JVM-GC stops all the running Java threads and thus causes long pause times. For example, if JVM-GC occurs while user writes are being served, the requests have to be suspended until JVM-GC finishes.

Furthermore, once the heap space is fully filled with data, Cassandra tries to make a room in the main memory by flushing out the dirty data to persistent storage. This flush operation may also be suspended by another garbage collector on a storage side. Flash-based SSDs, which are widely deployed in data centers these days, should maintain enough free NAND space to quickly absorb the data being flushed. If not, it triggers SSD-GC to reclaim free NAND space. SSD-GC involves lots of I/Os, so it delays the flush operation for a long time, causing long tail latency [7].

In order to alleviate the problem above, we propose a novel synchronized garbage collection technique, *SyncGC*, which synchronizes the executions of two GC instances (i.e., GC for a JVM and GC for an FTL), so that they execute simultaneously during the same period of time. The two GC instances run in separate machines (i.e., in the x86 host and in the SSD). Therefore, by intentionally scheduling them to run in an overlapped manner, the probability that the garbage collectors interfere with the flushing operations can be minimized. This simple synchronization, however, causes frequent invocations of GC which particularly have a negative impact on the lifetime of SSDs. To avoid this, we also propose a new GC policy that prevents unnecessary garbage collection on the SSD side.

We have implemented SyncGC in Cassandra on top of a real-world SSD which is customized to expose synchronization interfaces to the host. Our experimental results using the YCSB benchmark [6] show that 99.99th-percentile and the maximum latency has been reduce by 35% and 37%, on average, respectively.

This paper is organized as follows. Section 2 briefly explains how Cassandra deals with write requests from clients and gives results showing the problem of long tail latency in Cassandra. After analyzing why tail latency occurs under write-intensive workloads, in Section 3, we explain the details of the proposed SyncGC technique. We demonstrate experimental results with YCSB in Section 4. Section 5 concludes the paper with future directions.

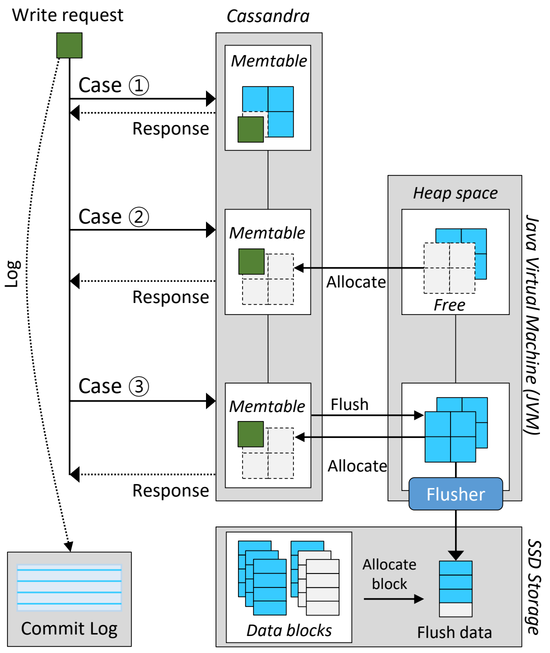


**Fig. 1. Write process in Cassandra**

## 2 ANALYSIS OF TAIL LATENCY

Figure 1 shows how Cassandra handles write requests, particularly focusing on two system components: a main memory and an SSD. Cassandra is written in the Java programming language, so its heap space is managed by a Java Virtual Machine (JVM). Cassandra is based on the LSM-tree algorithm [19], so it internally maintains a sorted list of key-value pairs in the data structure, called a *memtable*, in the main memory [2, 3]. When the memtable becomes full or reaches a certain threshold, key-value pairs are persistently flushed to the SSD at once.

When a user write request comes, Cassandra writes a log to the commit log in storage to maintain data consistency, and then stores the requested data in the previously allocated free space of the memtable. The write request is then finished, and a result is sent to a client (the case ① in the Figure 1). If there is no free space available in the memtable, Cassandra asks more memory space for JVM to create a new memtable. After the free-space allocation, the requested data is stored in the new memtable, and the user request is then finished (②). If Cassandra fails to find any available space in the heap, it starts to flush pending dirty data kept in the memtables to the SSD. After the dirty data are persistently written, the memtables are released, and the allocated memory returns to the free memory pool in the heap (③). This flush operation takes a long time because it involves disk I/Os. Thus, Cassandra runs a flush thread that evicts the memtables in background when the amount of accumulated data reaches a configured threshold value.

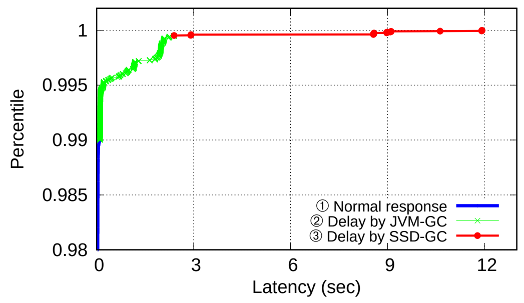


**Fig. 2. Cumulative distributions of long latency**

In order to understand the latency characteristic of Cassandra, we collect response times of write requests while the YCSB benchmark runs [6] and plot the CDF graph of them, which is shown in Figure 2. We observe that Cassandra seriously suffers from long tail latency. We highlight the graph with three different colors. The blue-line indicates response times of the requests that are immediately served by Cassandra with no delays (i.e., the case ① in Figure 1). 99% of the requests are categorized as the blue, and their average latency is shorter than 0.1 second. The green line represents response times of the requests which are delayed while allocating more free space in the heap (②). Overall, the allocation of free memory space is done quickly, but it sometimes takes a relatively long time which reaches 2 seconds in the worst case. Moreover, we observe that few requests (colored in red) are suspended for a very long time (~ 12 seconds) to wait for the flush thread to empty the heap space by flushing out dirty data (③).

After close examinations, we find that two garbage collectors running in the main memory, JVM-GC, and in the SSD, SSD-GC, cause the tail latency.

## 2.1 Impact of JVM-GC on Tail Latency

Like many enterprise/distributed systems (e.g., Hadoop [20], Spark [21], HBase [22], Neo4j [23], Kafka [24], Solr [25], and ZooKeeper [26]), Cassandra is built upon the Java environment because of its benefit of faster time-to-market. Cassandra delegates memory management entirely to JVM, enabling enterprise developers to easily manage the heap without any concerns about explicit memory free, memory leaks, and dangling pointers. The most important feature of the JVM memory management system may be garbage collection. JVM-GC reclaims memory space occupied by deleted objects. And, if the heap is severely fragmented, it also performs compaction to create a large and continuous free space. Since JVM-GC is a time-consuming job, if possible, JVM attempts to run it in background to hide associated overheads.

While it works well in most cases, under memory intensive applications, on-demand JVM-GC is unavoidable. For example, in Cassandra, Java service logics and the memtable

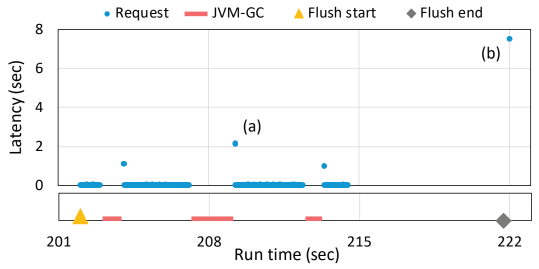


**Fig. 3. Latency fluctuation caused by JVM-GC**

compete for the limited heap space, repeating memory allocation and free, which leads the heap space to be fragmented severely. To create free heap space by compaction, JVM-GC has to be invoked in foreground, and it results in the long suspension of running Java threads, which is often called a *stop-the-world* (STW) pause.

Figure 3 illustrates the response time delays occurred while serving user requests. In Figure 3, the x-axis is the time (unit: second), and the blue-line represents the response times of user requests. The red-line indicates whether JVM-GC is running or not. At around 202, 207, and 213 seconds, JVM-GC is triggered on demand to reclaim free heap space, which typically requires 1-2 seconds. Since user requests have to wait for JVM-GC to finish, they are delayed by the same amount. Note that Figure 3 shows the start and end times of the flush operation that will be explained later.

Some might think that the response time fluctuation can be mitigated by applying better JVM-GC policies. OpenJDK supports various JVM-GC policies, such as Parallel-GC, CMS-GC, and G1-GC. Parallel-GC utilizes available cores in the system to run multiple GC instances in parallel, and is used as a default GC policy. CMS-GC attempts to reduce long pause times of full JVM-GC by dividing it into multiple subphases. G1-GC further improves CMS-GC to bound pause times within a specific value and also employs better compaction algorithms. Table 1 compares the average pause times of the three GC policies. CMS-GC and G1-GC show less GC overheads than Parallel-GC, but still show pretty long pause times. Moreover, CMS-GC and G1-GC incur more frequent GC invocations, which means that they often interfere with user requests. As a result, the latency fluctuation by JVM-GC cannot be completely removed, regardless of the type of GC policies.

**Table 1: GC pause times and invocation counts of various JVM-GC policies**

| Type | Parallel-GC | G1-GC | CMS-GC |
|---|---|---|---|
| Pause time (Avg.) | 458ms | 203ms | 59ms |
| Count | 66 | 123 | 158 |

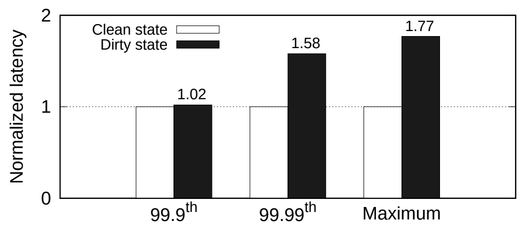


Fig. 4. Impact of SSD-GC on tail latency

## 2.2 Impact of SSD-GC on Tail Latency

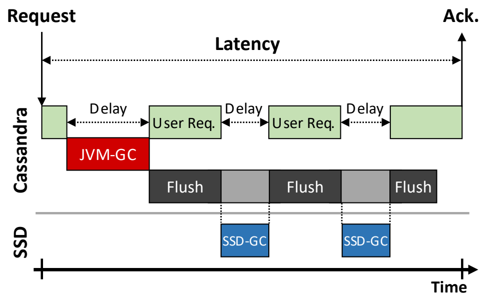
Another source that greatly increases overall tail latency is SSD garbage collection. If Cassandra finds that it is impossible to create sufficient free heap space though JVM-GC, it begins to empty the memtables by flushing pending data to the SSD. This flush operation is, however, often delayed by SSD-GC. To quickly write the data being sent from Cassandra, the SSD should maintain enough free NAND blocks. If free NAND blocks are exhausted, the SSD triggers GC, suspending the flush operation until enough NAND blocks are created. For example, in Figure 3, the flush operation starts at 201 second, but it does not finish until 222 seconds because of the delay by SSD-GC. Since SSD-GC requires much longer time than JVM-GC, its impact on tail latency is more significant. In the above example, the response time of a user request blocked by SSD-GC increases to 7 seconds.

In order to confirm the impact of SSD-GC, we carry out additional experiments with a clean SSD and a dirty SSD. Here, the clean SSD is a fresh-out-of-box (FOB) SSD only with free NAND blocks. Since enough free blocks are available, SSD-GC is never triggered while running a benchmark. The dirty SSD is an SSD that has been used for a long time and thus has many dirty blocks that require SSD-GC in the near future. Figure 4 shows that, with the clean SSD, long tail latency is not observed – the 99.9th and 99.99th percentile latency is almost the same as the maximum latency. On the other hand, compared to the clean SSD, the dirty SSD shows 1.58x and 1.77x longer latency, respectively, for the 99.99th percentile and the maximum.

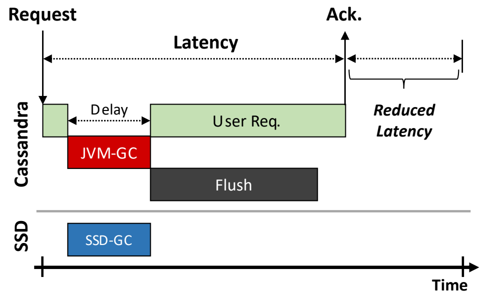
## 3 DESIGN OF SYNCGC

We have seen that two garbage collectors, JVM-GC and SSD-GC, badly affect tail latency. The ultimate solution to get rid of the interventions by GC is to completely hide or eliminate GC overheads themselves, but it is practically infeasible as we already dicussed in Section 2.1. Rather than eliminating GC overheads, SyncGC attempts to minimize its negative impact by scheduling multiple GC instances to run in an overlapped manner.

Figure 5 illustrates a basic idea of SyncGC, comparing it with the conventional Cassandra. As shown in Figure 5a, JVM-GC and SSD-GC are triggered depending on their own



(a) Baseline



(b) SyncGC

Fig. 5. Comparison of baseline and SyncGC

internal conditions, without any awareness of the status of another garbage collector. Thus, right after user requests are interrupted by JVM-GC, they are halted by SSD-GC again. SyncGC hides such frequent GC interruptions by means of explicitly scheduling SSD-GC in sync with JVM-GC as in Figure 5b. The negative impact of JVM-GC does not disappear even with SyncGC, but long latency caused by SSD-GC can be hidden behind JVM-GC.

While it is conceptually simple, in order to realize the idea of SyncGC in Cassandra with an SSD, the refactoring of the existing GC modules, in addition to defining new interfaces for synchronization, is necessary. In the following subsection, we explain this issue in detail.

## 3.1 Overall Architecture of SyncGC

Figure 6 shows an overall architecture of SyncGC, along with its operational flow. A SyncGC module is implemented as part of Cassandra inside JVM, and is designed to interact with the two GC modules in the JVM and the SSD. When JVM-GC is triggered, the SyncGC module is informed that a new GC operation just started (① in Figure 6). Then, SyncGC decides whether or not SSD-GC is required, and, if so, it estimates how many NAND blocks should be garbage collected. This is the responsibility of the block reclamation policy in SyncGC (②). The number of blocks to be reclaimed is then delivered to the FTL inside the SSD, and the FTL keeps performing SSD-GC until the required free blocks are obtained (③). After JVM-GC is finished (④), the SyncGC asks for the

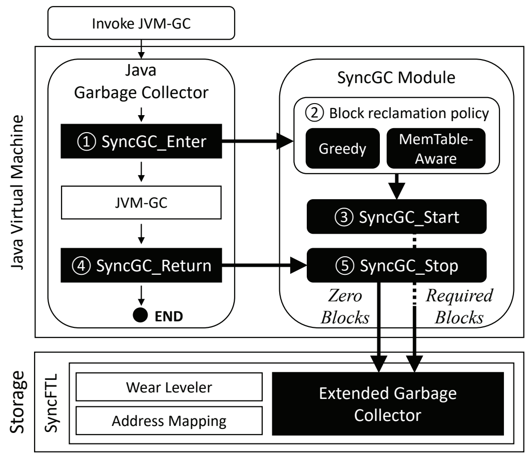


**Fig. 6. Overall architecture of SyncGC**

FTL to terminate SSD-GC since JVM-GC stopped and user requests will be issued soon (⑤).

Creating interfaces between the SyncGC and the JVM-GC modules is straightforward since these are both implemented in the same virtual machine. At the beginning of the `PSScavenge::invoke()` function which is invoked when JVM-GC starts, we just add few lines of code that call the function `SyncGC_Enter()`. Letting the SyncGC module know when JVM-GC stops can be done in a similar manner. We add few code lines that invoke `SyncGC_Return()` at the end of the `PSScavenge::invoke()` function.

Making SyncGC communicate with the SSD FTL is rather complicated because they run in separate systems. We use the SCSI generic I/O (SG_IO) interface that enables user applications to directly send custom SCSI commands to a device though the `ioctl()` system call. For the sake of simplicity, only one custom SCSI command is added, which delivers the number of NAND blocks to be reclaimed as a command parameter. This SCSI command is differently encapsulated by two functions of the SyncGC module, `SyncGC_Start()` and `SyncGC_Stop()`. For the `SyncGC_Start()` function, the parameter value is set larger than 0, so it automatically triggers SSD-GC to create free blocks. On the other hand, the parameter value is always 0 for `SyncGC_Stop()`. Since free blocks are not necessary anymore, the FTL terminates SSD-GC if it is running. To add the new SCSI command, the modification of the SSD controller that runs the FTL is required. We have modified Samsung's SM843T SSD controller [5], but there is no serious modification of the existing FTL module – it is extended to start or stop garbage collection upon receiving SyncGC commands.

## 3.2 Block Reclamation Policy

The block reclamation policy of SyncGC is responsible for deciding the number of free blocks to be garbage collected

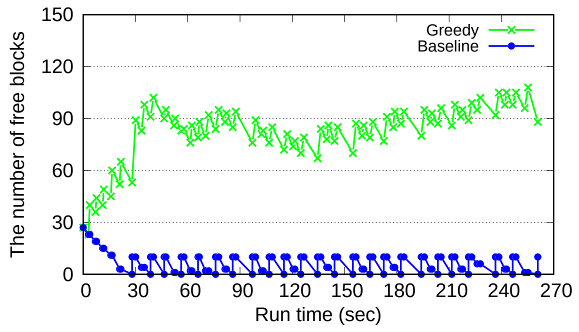


**Fig. 7. Free blocks created by Baseline and Greedy**

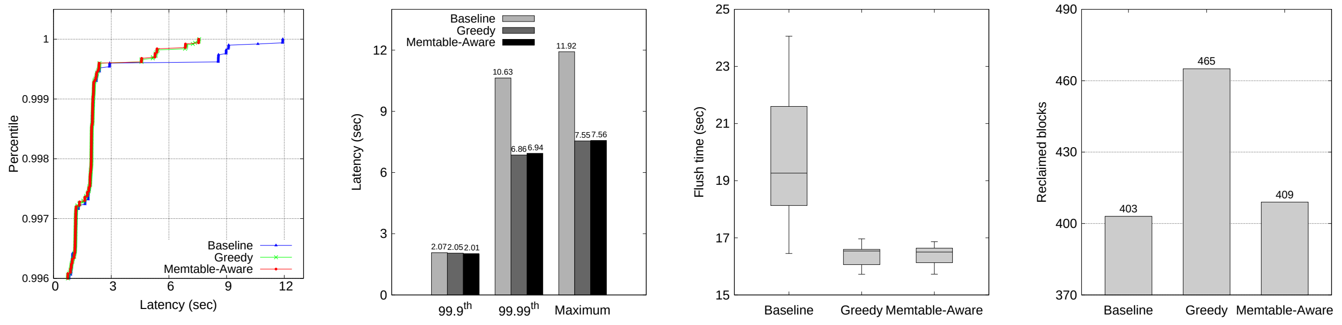
by SSD-GC. We discuss two possible policies below, comparing their pros and cons.

**Greedy policy:** It reclaims as many NAND blocks as possible while JVM-GC runs. The greedy policy is simple yet effective in terms of minimizing the interventions by SSD-GC. SyncGC just needs to call `SyncGC_Start()` with the parameter of $\infty$ and invoke `SyncGC_Stop()` to stop SSD-GC. With the greedy policy, a plenty of free NAND blocks can be reclaimed, which enables us to complete the flush operation with minimal delays. Figure 7 shows experimental results that compare the greedy policy with the baseline (original Cassandra). While the baseline often experiences the shortage of free blocks, the greedy policy maintains large enough free blocks to absorb data being flushed by Cassandra.

The major drawback of the greedy policy is that it results in a quick wear-out of NAND flash, shortening the SSD lifetime. This is because it forces the FTL to select premature NAND blocks as victims for GC. For example, suppose that an NAND block with hot data (which are frequently updated) is selected as a victim. The hot data in the block are moved to other NAND blocks, but they are likely to be invalided soon after their movement. Consequently, moving the hot data turns out to be useless, and extra writes performed for moving them just waste the limited P/E cycles of NAND blocks.

**Memtable-aware policy:** In order to solve this problem, we propose an alternative approach, called a *memtable-aware policy*. This policy is based on an idea that if there exist sufficient free NAND blocks to accommodate the memtables that will be flushed by Cassandra, it is unnecessary to reclaim more free NAND blocks. For example, if the total size of the memtables is 3 GB, at most 3 GB free space needs to be prepared on the SSD side. In other words, reclaiming more free blocks are unnecessary and just wastes the SSD lifetime. However, if free NAND space smaller than the memtables is maintained, the performance degradation caused by on-demand SSD-GC is unavoidable.

Therefore, a key challenge with the memtable-aware policy is how to accurately estimate the amount of the pending



**(a) CDF of response times** **(b) 99.9/99.99/Max latency** **(c) Memtable flush times** **(d) Block erase counts**

**Fig. 8. Experimental results with SyncGC**

data accumulated in the memtables which will be evicted soon from the main memory. This is done by keeping track of the status of the memtable management module of Cassandra. We can easily monitor how many data are newly written to the memtables. The size information is delivered to the SyncGC module, so that it uses this number as the parameter for `SyncGC_Start()`.

## 4 EVALUATION

In order to evaluate the effectiveness of SyncGC, we have implemented the SyncGC module in Cassandra 2.2.8 running in the Linux kernel 4.4.0-38. Our host machine is equipped with an Intel i7-2600 CPU with 16 GB DRAM. The storage device we use is Samsung's SM843T, which is customized to support the synchronization interface for SyncGC. OpenJDK 1.8.0 is set to have 12 GB of the heap space, and ParallelGC is used as a default garbage collector. The maximum memtable size is 3.33 GB.

We have used YCSB (Yahoo! Cloud Serving Benchmark) to generate workloads. Since we are interested in optimizing the tail latency of write requests, we select the Workload A of YCSB which is an update heavy workload. To create more intensive workloads, four threads run simultaneously which issue 50,000 requests each.

**Impact on tail latency:** Figure 8a shows a CDF graph that compares tail latency of the three policies, `Baseline`, `Greedy`, and `Memtable-Aware`, which represent the original Cassandra, SyncGC with the greedy policy, and SyncGC with the memtable-aware policy, respectively. As shown in the figure, `Greedy` and `Memtable-Aware` effectively eliminate the tail latency. We further analysis the latency of the three policies for the 99.9th, 99.99th percentiles, and maximum latency in Figure 8b. As expected, two techniques employing SyncGC outperform `Baseline` – they show 35% and 37% shorter latency for the 99.99 percentile and the maximum, respectively. This is due to the fact that SyncGC eliminates the intervention by SSD-GC while flushing out the memtables, thereby enabling us to create free space quickly. In case of the 99.9th percentile, however, all three methods show similar response times. This is because, even though SyncGC prevents user requests from being delayed by SSD-GC which is quite long, the interference with JVM-GC cannot be avoided.

**Reduced flush time:** We compare the memtable flush times of the three policies in Figure 8c. Both `Greedy` and `Memtable-Aware` greatly reduce the time taken for flushing the memtables – the average flush time of `Baseline` is 19.77 seconds, while those of `Greedy` and `Memtable-Aware` methods are 16.42 and 16.41 seconds, respectively. Particularly, we also observe that the flush time becomes much more stable with SyncGC. The standard deviation of `Baseline` is about 2.27 seconds, while `Greedy` and `Memtable-Aware` is 0.36 and 0.32 seconds, respectively.

**Impact on block erasure count:** Finally, we measure the number of block erasure counts that are performed by `Baseline`, `Greedy`, and `Memtable-Aware`, which is illustrated in Figure 8d. In case of `Baseline`, SSD-GC is executed at the moment only when no blocks are available to write data. That is, in `Baseline`, the execution of SSD-GC is delayed as much as possible until SSD-GC is actually necessary. Unlike `Baseline`, `Greedy` creates as many free blocks as possible by performing SSD-GC whenever JVM-GC runs. This aggressive GC policy, however, results in the largest number of block erasures among all of the evaluated policies. `Memtable-Aware` shows the similar block erasure count as `Baseline` by reclaiming only the limited number of free blocks that will be used soon. Even if `Memtable-Aware` intentionally regulates SSD-GC, it is not badly affected from the lack of free NAND blocks because it always keeps large enough free blocks to absorb data being flushed. For example, in Figure 8b, the tail latency of `Memtable-Aware` is almost equivalent to that of `Greedy`.

## 5 CONCLUSION

In this paper, we analyzed long tail latency of Cassandra and found that user writes were often delayed by garbage collectors running in the main memory and the SSD, which resulted in high fluctuation of write response times. To alleviate the problem of long tail latency, we proposed SyncGC that simultaneously performed JVM-GC and SSD-GC, hiding SSD-GC overheads behind JVM-GC. Our experiments showed that SyncGC reduced the 99.99th-percentile latency and the maximum latency by 35% and 37%, respectively. As our future work, we plan to combine SyncGC with better JVM-GC policies, such as G1-G1 and CMS-GC, which offer shorter pause times but require a fine-grained GC synchronization control and evaluate the reduction of tail latency in a distributed Cassandra environment.

## 6 ACKNOWLEDGMENTS

We would like to thank anonymous referees for valuable comments that greatly improved our paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2015M3C4A7065645 and NRF-2018R 1A2B6006878.) The ICT at Seoul National University provided research facilities for this study. Sungjin Lee was supported by the NRF grant funded by the Korea government (Ministry of Science and ICT) (NRF-2017R1E1A1A01077410) and the DGIST R&D Program of the Ministry of Science and ICT (18-EE-01). *(Corresponding Author: Jihong Kim)*

## REFERENCES

[1] G. DeCandia *et al.,* "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. of the Symposium on Operating Systems Principles*, pp. 205-220, 2007.

[2] F. Chang *et al.,* "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, Issue. 2, 2008.

[3] A. Lakshman *et al.,* "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, Issue. 2, pp. 35-40, 2010.

[4] Y. Xu *et al.,* "Bobtail: Avoiding Long Tails in the Cloud," in *Proc. of the Networked Systems Design and Implementation*, pp. 329-341, 2013.

[5] SAMSUNG 843T Data Center Series, http://www.samsung.com/global /business/semiconductor/file/media/SM843T_Product_Overview-0.pdf

[6] B.-F. Cooper *et al.,* "Benchmarking Cloud Serving Systmes with YCSB," in *Proc. of the ACM Symposium on Cloud Computing*, pp. 143-154, 2010.

[7] S. Yan *et al.,* "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," in *Proc. of the USENIX Symposium on File and Storage Technologies*, pp. 15-28, 2017.

[8] A. Vulimiri *et al.,* "Low Latency via Redundancy," in *Proc. of the ACM Conference on Emerging Networking Experiments and Technologies*, pp. 283-294, 2013.

[9] L. Suresh *et al.,* "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," in *Proc. of the Networked Systems Design and Implementation*, pp. 512-527, 2015.

[10] J. Dean *et al.,* "The Tail at Scale," *ACM Communications*, vol. 56, no. 2, pp. 74-80, 2013.

[11] M. Alizadehdeh *et al.,* "Data Center TCP," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 63-74, 2010.

[12] M. Kambadur *et al.,* "Measuring Interference Between Live Datacenter Applications," in *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 51, 2012.

[13] J. Li *et al.,* "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," in *Proc. of the ACM Symposium on Cloud Computing*, pp. 1-14, 2014.

[14] V. Jalaparti *et al.,* "Speeding up Distributed Request-response Workflows," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 219-230, 2013.

[15] W. Reda *et al.,* "Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling," in *Proc. of the European Conference on Computer Systems*, pp. 95-110, 2017.

[16] D. Zats *et al.,* "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *Proc. of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 139-150, 2012.

[17] R. Sumbaly *et al.,* "Serving Large-scale Batch Computed Data with Project Voldemort," in *Proc. of the USENIX conference on File and Storage Technologies*, pp. 18-30, 2012.

[18] A. Liljencrantz *et al.,* "How Not to Use Cassandra," Cassandra Summit, 2013. https://www.youtube.com/watch?v=0u-EKJBPrj8.

[19] P. O'Neil *et al.,* "The Log-structured Merge-Tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351-385, 1996.

[20] V.-K. Vavilapalli *et al.,* "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *Proc. of the Symposium on Cloud Computing*, no. 5, 2013.

[21] M. Zaharia *et al.,* "Apache Spark: a Unified Engine for Big Data Processing," *ACM Communications*, vol. 59, no. 11, pp. 56-65, 2010.

[22] HBase. `https://hbase.apache.org`, accessed May 28, 2018.

[23] Neo4j. `https://neo4j.com`, accessed May 28, 2018.

[24] Kafka. `https://kafka.apache.org`, accessed May 28, 2018.

[25] Solr. `https://lucene.apache.org/solr/`, accessed May 28, 2018.

[26] ZooKeeper. `https://zookeeper.apache.org`, accessed May 28, 2018.