High-Performance Multi-Stream Management for SSDs

Yongjae Chun, Kyeore Han and Youpyo Hong *

Division of Electronics and Electrical Engineering, Dongguk University-Seoul, Seoul 04620, Korea; genie224@dongguk.edu (Y.C.); vwno134@dongguk.edu (K.H.)
* Correspondence: yhong@dongguk.edu; Tel.: +82-2-2260-3818

Abstract: Owing to their advantages over hard disc drives (HDDs), solid-state drives (SSDs) are widely used in many applications, including consumer electronics and data centers. As erase operations are feasible only in block units, modification or deletion of pages cause invalidation of the pages in their corresponding blocks. To reclaim these invalid pages, the valid pages in the block are copied to other blocks, and the block with the invalid pages is initialized, which adversely affects the performance and durability of the SSD. The objective of a multi-stream SSD is to group data by their expected lifetimes and store each group of data in a separate area called a stream to minimize the frequency of wasteful copy-back and initialization operations. In this paper, we propose an algorithm that groups the data based on input/output (I/O) types and rewrite frequency, which show significant improvements over existing multi-stream algorithms not only for performance but also for effectiveness in covering most applications.

Keywords: flash memory; solid-state drive; multi-stream; garbage collection; write amplification

1. Introduction

Solid-state drives (SSDs) are rapidly replacing hard disc drives (HDDs) in many applications owing to their advantages in terms of speed, power consumption, and size [1]. However, in-place overwrite functionality is not allowed, and erase operations must precede new writes in SSDs because of the inherent features of flash memory, which constitutes SSDs [2,3]. In addition, erase operations are possible only in block units, so a block may contain many invalid pages before deletion of the entire block of data [4]. To reclaim such wasted storage space, valid pages from multiple blocks are copied to a new block, and the old blocks are initialized by complete deletion of their content; this process is called garbage collection (GC). This means that the actual number of write operations applied to an SSD is greater than that intended by the host. Therefore, GC adversely affects the lifetime and performance of a SSD owing to the extra write operations, even though it improves storage utilization [5–7]. A simple yet effective solution to handle this is to expect the lifetime of data, which is the interval between their creation and the time the data are invalidated due to deletion or modification, and group pages with similar lifetimes to minimize copying pages with long lifetimes. This is the basic concept of a multi-stream drive, where a stream refers to a block that ideally contains pages with similar lifetimes. However, this simple idea is challenging to implement because the lifetime of a page cannot be accurately predicted [8,9].

In implementing multi-stream, hardware limitations such as fast memory capacity lead to an upper limit of 4 to 16 available streams [10]. A stream identifier (stream ID) that identifies such finite number of streams is included in NVMe (NVM Express) standards and can be transmitted from the host to the SSD. Therefore, it is possible to classify data in a host where various hints exist from application to operating system level. Recent multi-stream methods have been tried in diverse stages, and there are two issues here: the performance itself and the extent to which it works.
In previous works [11–13], various standards for classifying data based on low-level information have been suggested. Each of these methods, which correlate the lifespan of files with logical address, file system, and system call, differs in scope and effectiveness. The logical address and lifetime of a file show a weak relationship in many applications, and the data categorization at the file system level depends on the specific type of data of a unique write pattern. In addition, numerous combinations of system calls are less applicable as they are applied to SSDs with finite available streams.

In this paper, we propose a widely applicable and effective multi-stream method that applies data classification criteria at a low level. Our method groups specific types of data of which their roles in the file system lead to regular write patterns. In addition, data showing an irregular lifespan in the classification increase multi-stream performance through additional classification based on file characteristics and operation types, and this base information is also extracted at the operating system level to ensure versatility. Lifetime calculation that is selectively applied to specific data types can improve multi-stream efficiency, and the classification based on numerical data maximizes the use of available streams.

Compared to Auto-stream [11], our method shows a more similar lifetime of the files in each category, which leads to better performance. Unlike FStream [12], our method further classifies the data with the most irregular lifetimes based on low-level information, thereby ensuring applicability and increasing efficiency of streams. Whereas PCStream [13] groups files in a narrow range using combinations of system calls, each group in our method covers a wide range of files. Therefore, our method can be implemented in most SSD controllers that have a limited number of streams.

Our study aims to reduce the internal copying operation of the GC process, the core purpose of multi-stream, and to present the applicability of the proposed method through experiments with various workloads.

2. Related Work

One of the successful multi-streaming algorithms is based on logical block addressing (LBA) of data [11,14], whose primary scheme involves dividing the entire LBA space into a finite set and tracking information that may contribute to lifetime, such as reference count or recent access time. This approach is effective when the LBA generation scheme is favorable such that the relevant files have contiguous addresses, which is not generally true; therefore, the applicability of this approach is limited.

Another simple yet powerful multi-stream management technique is based on grouping data based on input/output (I/O) types [12]. Specifically, separate streams are allocated to user data, journal, inode, directory, and others in the Linux ext4 file system [15]. The major factor in the success of this approach is the short and monotonic lifetime of the journal data, where journal information is written to a predetermined logical space in a circular pattern with a relatively short lifetime, and the amount of journal data typically set by the user is significant. Metadata, such as inode and directory information, however, do not show clear patterns in terms of lifetimes; the heavy dependence on the features of the journal data implies the limitation of this approach. Multi-streaming performance also degrades significantly in cases that do not involve journal data.

Kim et al. [13] proposed a multi-stream management technique that utilizes various types of information from the host at higher levels of abstraction. However, the applicability of any scheme that is extensively dependent on the host may be limited, especially for hardware solutions. In this paper, we propose a new multi-stream algorithm based only on the data types and operation types of the data to be stored. Furthermore, this information is available to SSD controllers, so the proposed algorithm is applicable in most cases.
3. Motivation

3.1. I/O Type

File systems generally have their own unique I/O type categories. For example, in Linux ext4 file systems, the I/O types include user data, journal, inode, directory, or other miscellaneous information in the kernel at the highest level. Such categorizations hint at the lifetimes of the corresponding files, which is the idea behind [12]. However, such coarse categorizations are not accurate for estimating the lifetimes of user data, which show heterogeneous write patterns, and multi-streaming algorithms based on such categorizations may lead to poor performance. Hence, our first motivation involves applying fine-grained categorizations where the distinctions impact the lifetime estimations of the corresponding files. Specifically, we propose additional classification of the user data based on their write operation types, namely synchronous-create, synchronous-append, and asynchronous.

3.2. Synchronous Write Operation

Synchronous write means that read/write operations are fully executed through the disk in the order generated by the host. For asynchronous writes, on the contrary, the data may be reorganized in a temporary buffer for more efficient storage [16]. For instance, multiple writes to the same file may be performed in the temporary buffer before the file is eventually stored in the disk. Asynchronous writes typically enable longer lifetimes owing to this process. Therefore, dividing the write operations into synchronous or asynchronous types improves the accuracy of multi-stream processing.

3.3. Characteristics of Append Operation

Another motivation of our work is that the manner in which a file is written to the storage is closely related to its lifetime. Specifically, we claim that an important factor contributing to the lifetime of a file depends on whether the write is of the create or append type. As the names suggest, create writes a new file and append additionally writes to an existing file. Unless the existing file size is a precise multiple of the block size, an append-type write operation requires merging existing and new data, which leads to overwrites in the block in that is being merged. This means that the data before merging are deleted, thereby accounting for the lifetime changes. The lifetime of a file that experiences overwriting can be estimated using the overwrite frequency; simply speaking, the file that is overwritten often can be expected to have a short lifetime. This applies commonly to files in the append mode and is the third motivation of our work.

4. Proposed Multi-Stream Algorithm

One of the distinct features of an operating system is its file system containing various data types. For example, in the Linux ext4 file system, which is the target of this work, the data types at the highest level in the kernel may include user data, journal, inode, directory, or other miscellaneous information. As each data type has its unique function, it can be inferred that there is a strong correlation between the data type and its lifetime. However, separation of the stream solely based on the data types offers performance benefits only when the proportion of journal data is significantly large, because only the journal data have a unique and monotonic overwrite pattern. Hence, we propose a multi-stream management approach that utilizes the data type and I/O type of files from the host in a hybrid manner. The key idea here is to apply top-level partition to user data, journal data, and metadata; furthermore, the user data, which typically constitute the largest portion of data, are divided according to the write operation type: synchronous-create, synchronous-append, and asynchronous.
By measuring a few parameters associated with the write operations, a reasonably accurate method of lifetime estimation is possible for the synchronous-append case, as follows. We measure a file-append interval time, \( I \), as

\[
I = \frac{(I_r \times N + (T_c - T_r))}{N + 1}
\]

where \( I_r \) is the recently recorded interval, \( T_c \) is the current time, \( T_r \) is the time of the most recent modification, and \( N \) is the total number of write operations applied to the data. A larger \( I \) value thus means a longer lifetime. We propose to measure and update such parameters, including \( I \) values, for each synchronous write operation such that append-write data with similar \( I \) values are divided into append streams.

To allocate a file with an append attribute to its corresponding stream, the entire time interval must be partitioned such that the files are evenly distributed. This is not a straightforward task because the distribution of \( I \) values varies significantly over workloads and times, as shown in Figure 1. In Figure 1, we calculated \( I \) values for every append-write in the first 20 min of each workload and counted the number of \( I \) values in each range whose length divides the mean of all \( I \) values in a workload by 50.

![Figures showing frequency of append-writes for various databases with different file-append interval times.](image)

**Figure 1.** Interval distributions in various workloads.

We propose a statistical approach to set the interval ranges of the append-writes that are expected to be stored in each stream. In the proposed method, the interval ranges of streams are determined after a specific quantity of interval samples is collected. We assume that the interval distribution of the subsequent append-writes follows the normal distribution with the mean and standard deviation of collected interval samples. Interval samples are collected until the total size of append-write requests exceeds “block
size \times \text{number of append streams}” in the case of the initial setting, and the interval ranges are redistributed whenever the size of additional append-write data exceeds “block size \times \text{number of append streams}”.

Specifically, the entire time domain is partitioned according to boundary values, each with a different cumulative probability given by \(\frac{N}{\text{number of append streams}}\) (\(N = \text{all natural numbers less than the number of append streams}\)) in the normal distribution. However, if one or more boundary values are less than 0, the negative boundaries are replaced with values that equally divide the time range below the minimum number of positive boundaries. For a more detailed explanation, Appendix A describes the algorithm for setting the time range of each append stream.

To estimate the validity of our approach, we investigated append interval time, which means that the time interval between the two most recent writes requests the recently appended file. Table 1 provides the mean and standard deviation of all append intervals within each workload. Figure 2 shows the conformity between the distribution of observed append intervals and the normal distribution with the mean and standard deviation from Table 1. Except for MySQL and Dbench, where append intervals are short so that partitioning the time domain on append streams is less important, each of the workloads shows append interval distribution similar to their normal distribution.

Table 1. Mean and standard deviation of append intervals.

| Statistic     | Varmail | Varmail_nj | MySQL | Cassandra | SQLite | Dbench |
|---------------|---------|------------|-------|-----------|--------|--------|
| Mean          | 20.989  | 7.961      | 0.006 | 1.259     | 0.016  | 0.026  |
| Standard Deviation | 8.579  | 3.622      | 0.243 | 4.418     | 0.008  | 0.016  |

Figure 2. Correlation between observed append intervals and normal distribution.
5. Evaluation

5.1. Experimental Setup

We modeled SSDs using C code to estimate the efficiency of the multi-stream algorithm. We designed a simulator to model the operations of SSD running trace files, which consist of the commands including LBA, operation type, data type, size, and arrival time of each actual write request from the host. The information regarding synchronous/asynchronous and create/append characteristics that determine stream ID of the file is extracted from the Linux kernel and also recorded to each commend. As for the environment to create the trace file, we performed diverse benchmark programs to Samsung T5 SSD 1 TB, through the Linux kernel version 3.10.0 under Intel(R) Core(TM) i7-9700K CPU with 32 GB RAM.

The key point of the simulator is the implementation of the flash translation layer (FTL) [17] on which the multi-stream function is mounted. In the simulator, each write request from the host machine is translated into several one-page-size writes distinguished by logical page numbers. These individual pages are allocated to the physical addresses of the blocks corresponding to each designated stream. To simulate a situation where the write amplification factor (WAF) is directly affected by the lifetime of each file, we emulated a single channel SSD with four append streams, and Table 2 shows the features of the emulated NAND flash memory.

| Table 2. Parameters of the NAND flash memory. |
|-----------------------------------------------|
| **Parameter**                       | **Value**    |
| Write page operation time             | 230 µs       |
| Read page operation time              | 25 µs        |
| Erase block operation time            | 0.7 ms       |
| Page size                            | 4 kB         |
| Block size                           | 128 pages    |
| Total number of blocks                | 524,288      |

The GC invocation and target block selection schemes have significant impacts on the WAF. In our experiments, the background GC is performed during idle time on the blocks with the highest number of invalid pages among blocks with at least 60% invalid pages [18]. The foreground GC is invoked upon arrival of new data if the SSD is 75% full to free up space for the new data. The WAF is calculated using the following formula.

\[
\text{WAF} = \frac{\text{number of arrived pages} + \text{number of copied pages}}{\text{number of arrived pages}}
\]  

The WAF of the proposed multi-stream algorithm is compared with those of the Single Stream, multiple queue (MQ) algorithm [11], and FStream [12] for various workloads, such as the Yahoo! Cloud Serving Benchmark (YCSB) [19] on Cassandra [20], Sysbench [21] on MySQL [22], Varmail of Filebench [23], phoronix-test-suite [24] on SQLite [25], and Dbench [26].

5.2. Workload Analysis

We conducted experiments with workloads having different profiles, such as mail servers and databases. Because the access patterns affect the lifetimes of the files significantly, we investigated the setup and I/O request characteristics for each workload. Varmail is a workload that mimics the data transaction pattern of a mail server. The workload creates files according to predetermined sizes and numbers and performs append and delete operations on randomly selected files. In our experiments, the number of files, file creation size, and append-write size were set to 7500, 40 kB, and 16 kB, respectively. We also conducted experiments using Dbench built on the phoronix-test-suite using the 6-clients mode, which generates I/O requests on the disk using filesystem calls.
To evaluate the efficiency of each multi-stream scheme for commonly used database management programs, we executed the Sysbench and YCSB applications on MySQL and Cassandra databases, respectively. Sysbench simulates a specific test profile called online transaction processing (OLTP) on a MySQL database with some characteristics, such as the number of tables. We performed the workload on a database consisting of 16 tables with other default settings, and only log files were written to a disk in a sequential write pattern owing to the characteristics of the in-memory database. YCSB is also a database performance evaluation program in which each test is divided into the load and run phases. We performed a specific benchmark called workload A for as much as 10,000,000 keys for each phase, consequently writing various files, including Commitlog, SSTable, filters, and other index files. We also conducted experiments on a widely used database engine, SQLite, using the phoronix-test-suite application to measure the insertion time of a certain amount of data. We selected the test configuration called 8 threads/copies, which resulted in random-write patterns.

We analyzed more details on the I/O types for each workload, as shown in Table 3. Each workload contains large amounts of journal data, except for the Varmail workload, which disables journaling (denoted by Varmail_nj), and each workload includes characteristic write request patterns, such as append-only in MySQL.

Table 3. Distribution of input/output (I/O) types (%).

| Category       | Varmail | Varmail_nj | MySQL | Cassandra | SQLite | Dbench |
|----------------|---------|------------|-------|-----------|--------|--------|
| Journal        | 81.53   | -          | 56.60 | 65.53     | 55.55  | 82.88  |
| Inode          | 0.53    | 28.37      | 0.01  | 0.93      | 0.01   | -      |
| Directory      | 3.93    | 39.90      | 0.01  | 0.64      | 0.01   | 0.07   |
| Misc. meta     | 0.03    | 0.03       | -     | 0.31      | -      | -      |
| Sync.-create   | 4.94    | 14.52      | -     | 19.69     | 11.21  | 8.23   |
| Sync.-append   | 8.97    | 17.15      | 43.38 | 2.85      | 33.22  | 8.62   |
| Asynchronous   | 0.07    | 0.03       | -     | 10.05     | -      | 0.20   |

We investigated the write patterns of the data, as shown in Tables 4 and 5. In Varmail and Varmail_nj workloads, synchronous-create and synchronous-append show cohesive LBA, respectively, even though most data writes are random. MySQL shows a sequential write pattern for a narrow range of logical block area, and Cassandra’s large files are divided into smaller sequential writes. In addition, SQLite and Dbench include large numbers of random writes.

Table 4. Size characteristics of data write requests.

| Average Size (KB) | Varmail | Varmail_nj | MySQL | Cassandra | SQLite | Dbench |
|-------------------|---------|------------|-------|-----------|--------|--------|
| Sync.-create      | 10.79   | 10.53      | -     | 425.29    | 2.19   | 41.07  |
| Sync.-append      | 6.49    | 8.34       | 8.27  | 21.85     | 4.10   | 4.84   |
| Asynchronous      | 22.48   | 28.15      | -     | 187.06    | -      | 318.41 |

Table 5. Logical block addressing (LBA) characteristics of data write requests.

| Characteristics    | Varmail | Varmail_nj | MySQL | Cassandra | SQLite | Dbench |
|--------------------|---------|------------|-------|-----------|--------|--------|
| Seq. write (%)     | 5.80    | 0.26       | 98.36 | 79.16     | 2.26   | 2.11   |
| Seq. LBA (%)       | sync.-cr | 60.26      | 78.25 | -         | 79.22  | 11.89  |
|                    | sync.-app | 32.56      | 39.91 | 96.32     | 15.91  | -      |
| Number of used block groups | 30 | 46 | 2 | 129 | 24 | 15 |
To simulate various workloads in our trace-driven emulator, we acquired information on the commands issued by the host using blktrace [27], which provides traces on the LBA, size, and timestamps. In addition, ftrace [28] is used to trace the kernel function calls, which distinguish synchronized operations and create/append characteristics.

5.3. Results

5.3.1. Interval Range Setting for Append Stream

Figure 3 shows distributions of error rates associated with the mean of intervals in the interval range setting for append streams. Because the interval range setting predicts a distribution of subsequent append intervals based on recent append intervals as mentioned in Section 4, the accuracy of the prediction affects the performance of the proposed method. The error rate for each interval range setting was calculated using the following formula.

\[
\text{Error rate (\%)} = \left( \frac{\text{mean}_{\text{recent}} - \text{mean}_{\text{curr}}}{\text{mean}_{\text{recent}}} \right) \times 100
\]

(3)

where \(\text{mean}_{\text{recent}}\) is the mean of intervals calculated in the recent interval range setting, and \(\text{mean}_{\text{curr}}\) is the mean of intervals from the current interval range setting.

![Figure 3. Distributions of error rates associated with mean of intervals in interval range setting.](image-url)
In Figure 3, our approach shows low error rates for the workloads, except for MySQL, in which most append intervals are short, and Cassandra, which has a small amount of append-writes, as shown in Table 3.

5.3.2. WAF

WAFs based on various multi-stream algorithms are presented in Figure 4. The MQ shows lower WAFs in Varmail and SQLite workloads compared to Single Stream owing to the characteristics of the journal, which shows a circular write pattern in a specific LBA area. Additionally, continuous LBA of synchronous-create and synchronous-append in Varmail and Varmail_nj contribute to classification of these two types of data into different streams. This tendency results in a lower WAF of MQ in Varmail_nj than FStream, which performs better overall. The proposed algorithm reduces the WAF by 12% over MQ in Varmail and 11% over FStream in Varmail_nj by not only separating the synchronous-create and synchronous-append operations but also effectively classifying synchronous-append with various intervals.

![Write amplification factor (WAF) comparison.](image)

As shown in Table 5, in the MySQL workload, data are written to the SSD in journal-like write patterns. As a result, the MQ performs poorly as it cannot separate journal and synchronous-append into different streams. On the other hand, the proposed algorithm and FStream effectively reduce WAF because the data and journal are stored in different streams. SQLite is a random write-intensive workload whose write patterns of the data are in contrast with that of the journal. The MQ tends to separate these I/O types and reduce WAF up to 8% compared to the Single Stream. In this situation, multi-stream algorithms show similar performances because of the short lifetimes of the data, as shown in Figure 1.

The MQ algorithm performs poorly for the Dbench workload, which has a large amount of random writes, thereby exposing the weakness of LBA-based streaming. However, the proposed method reduces the WAF by 10% compared to the MQ with robust performance even for irregular write patterns. As shown in Tables 3 and 4, the large amounts of synchronous-create in Dbench workload allow FStream to show similar WAFs as the proposed algorithm.

Cassandra has a distinctive feature where the file sizes are exceptionally large, occupying as many as 2 to 13 blocks each, as shown in Table 4. In such cases, most blocks are occupied by a single file with one data type; this results in both Single Stream and MQ being approximately as effective as the other approaches because the different types of data are not mixed in the same block. For this reason, the potential advantage of multi-stream
processing is limited unless a single file is stored over multiple channels in a distributed manner, which is not the experimental setup considered for this study.

6. Conclusions

A multi-stream management algorithm that utilizes information on the data type and operation type associated with the stored data on the SSD is presented in this paper. Only for particular types of data are the expected lifetimes computed and used to further refine the accuracies of stream partitions. The goal of this selective refinement approach is to minimize the computation cost while maximizing the stream classification accuracy. The combined strategy of using data type, operation type, and expected lifetimes is expected to cover a wide range of applications. Unlike most existing multi-stream algorithms that are solely based on file types or logical addresses, the proposed algorithm is proven to be not only effective for improvement but also for robustness to be applied to most workloads with variable profiles.

Author Contributions: Conceptualization, Y.C., K.H. and Y.H.; methodology, Y.C., K.H. and Y.H.; software, Y.C. and K.H.; validation, Y.C., K.H. and Y.H.; formal analysis, Y.C., K.H. and Y.H.; investigation, Y.C. and K.H.; resources, Y.C. and K.H.; writing—original draft preparation, Y.H.; writing—review and editing, Y.C., K.H. and Y.H.; visualization, Y.C. and K.H.; supervision, Y.H.; project administration, Y.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Acknowledgments: This work was supported by ‘Research on NOC application for power-efficient backbone in SDC’ funded by Samsung Electronics and the Dongguk University Research Fund of 2021. The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

The algorithm shows the interval ranges setting algorithm, which includes three global variables, namely, the boundary values of the interval ranges, accumulated append-write size, and interval-queue. The accumulated append-write size is used to determine when to set the interval ranges and the interval-queue stores the interval values as samples. These two variables are updated for each append-write request (lines 1–2), and the interval ranges are set by the function “SET_INTERVAL_RANGE” if the accumulated size of the append-write request exceeds the threshold (lines 4–5). Specifically, after calculating the mean of all interval values in the interval-queue, the standard deviation is calculated using the sum of squared deviations of each interval and number of entries (lines 9–18). At this time, the entry used for calculating the deviation is removed from the queue, that is, the used interval is not reused. The interval distribution of append-write requests is predicted until the next interval setting based on the normal distribution using the mean and standard deviation. Therefore, to divide the append-write requests evenly into all append streams, the boundary values are set such that the cumulative probability in all intervals are equal (lines 19–23). Since the interval values cannot be negative, the negative boundary values are replaced with values that equally divide the range below the minimum positive value of the boundaries (lines 25–28). After setting the intervals, the accumulated append-write size is initialized to 0, such that the next setting proceeds independently of the latest setting (line 6).
Algorithm A1. Interval range setting in append-write streaming

Input:
1. \( S_r \): recent append-write request size
2. \( I_r \): time interval between two most recent write requests for the recently appended file

Result:
Interval ranges are set by boundary values \( B[\text{Num_of_streams}] \).
/*e.g., if using 4 append streams, ranges are \([0, B[1]), [B[1), B[2]), [B[2), B[3]), [B[3), \infty)\]*/

Global Variables
1. \( S_{ac} \): accumulated size of append-write requests (initialized to 0)
2. \( \text{Interval-queue} \): queue in which intervals are stored
3. \( B[\text{Num_of_streams}] \): boundary values of interval ranges (initialized to 0)

Procedure \( \text{MAIN}(S_r, I_r) \)
1. enqueue \( I_r \) to \( \text{Interval-queue} \);
2. \( S_{ac} = S_{ac} + S_r \);
3. /*set interval ranges, if cumulative append-write size exceeds threshold*/
4. if \( S_{ac} \geq \text{Block_size \times \text{Num_of_streams}} \) then
5. \( \text{SET_INTERVAL_RANGE}(\) \);
6. \( S_{ac} = 0 \);
7. else
8. maintain current interval ranges;
end procedure

Procedure \( \text{SET_INTERVAL_RANGE}(\) \);
9. /*calculate mean \( \mu \) of intervals in \( \text{Interval-queue} \)*/ \( \mu = \frac{\text{sum of every interval in \( \text{Interval-queue} \)}}{\text{number of entries in \( \text{Interval-queue} \)}} \);
10. /*calculate standard deviation \( \sigma \) of intervals in \( \text{Interval-queue} \)*/
11. define \( \text{sum_of_squared_deviations}(=0) \) and \( \text{num_of_intervals}(=0) \);
12. while \( \text{Interval-queue} \neq \text{EMPTY} \) begin
13. get an entry (denoted by \( E \)) from \( \text{Interval-queue} \);
14. \( \text{sum_of_squared_deviations} += (\mu - \text{interval value of } E)^2 \)
15. \( \text{num_of_intervals}++ \);
16. remove \( E \) from \( \text{Interval-queue} \);
17. end while
18. \( \sigma = \sqrt{\frac{\text{sum_of_squared_deviations}}{\text{num_of_intervals}}} \);
19. define random variable \( x \), following a normal distribution with \( \mu \) and \( \sigma \), so that probability \( P(x) \) is \( \frac{1}{\sigma \sqrt{2\pi}} \exp\left[\frac{-(x-\mu)^2}{2\sigma^2}\right] \);
20. /*set boundary values at specific quantiles in the normal distribution*/
21. for \( i = 1; \text{Num_of_streams} > i; i++ \) begin
22. update \( B[i] \) for the value that meets the following condition.
23. end for
24. /*Replace negative boundaries such that the minimum of positive boundaries is equally divided*/
25. for \( i = \text{Num_of_streams}-1; i > 0; i-- \) begin
26. if \( B[i] < 0 \) then
27. \( B[i] = B[i+1] \times \frac{i}{i+1} \);
28. end for
end procedure
