Impact of garbage collection algorithms and over-provisioning sizes on flash-based solid-state drives

Hossein Ghotbaddini
University Of Toronto
h.ghotbaddini@mail.utoronto.ca

Haoran Liu
University Of Toronto
haoranliu.liu@mail.utoronto.ca

ABSTRACT

Solid-state drives (SSDs) made out of Flash devices have gained a lot of prominence in recent years due to their increased performance and endurance. A common but critical question one can ask is which SSD configuration would be more efficient for a specific type of application. Trying to find the answer by running an application on many SSDs with different configurations wouldn't be the best idea because first of all, they're not necessarily accessible, and secondly, real I/O would take a lot of time. Hence, here we built a **Flash-Based SSD Simulator** which can be configured to have different sizes of Flashs, Blocks, and Pages, and can run many types of Garbage Collectors with arbitrary frequency. We also implemented the feature of having an Over-Provisioning region of any size and the option of supporting multiple CPUs. To show the usage of this tool and the fact that it really works, we tested it on several block traces with various configurations and compared those results to get the same conclusions our peers got in their related papers.

1. INTRODUCTION

The garbage collection algorithm, which specifies the strategy of choosing a block to erase, impacts the performance of an SSD[8]. A GC algorithm will select a victim block and copyback all the valid physical pages to a block with erased physical pages, and perform erase operation on the victim block to prepare it for new writes. The performance of GC algorithms is defined with WAF(write amplification factor) [9]. Since flash-based solid-state drives can wear out under erase operations, a GC algorithm is often designed to improve GC performance, while avoiding over-wearing a particular SSD block[8].

In this project, we've implemented the following GC Algorithms and compared their performance.

- Greedy GC Algorithm [1,2,3]: The Greedy Algorithm iterates through the dirty blocks and selects the block with the most invalid page count for garbage collection. This algorithm has a time complexity of O(n).
- FIFO GC Algorithm [1,3,4,5]: Performs garbage collection on the first block in the queue, which was pushed into the queue after it was filled with data. This is the simplest GC algorithm, and has a time complexity of O(1).
- Window GC Algorithm [1,6]: Keeps track of a sliding sequential subset of blocks and selects the block with the most invalid blocks for garbage collection. Window GC algorithm tradeoff the greedy GC algorithm's performance for the time efficiency of the FIFO GC algorithm.

The simulator runs in CLI, takes in drive configuration parameters, an I/O trace file as input, and writes the simulation results to an output file. The output of the simulator includes

overall statistics like read(R)/write(W)/discard(D) operation counts, the number of times Garbage Collection was run, total copyback counts, average GC time, and the WAF(Write Amplification Factor), as well as individual CPU R/W/D operation counts, and individual blocks' statistics, including invalid page number and erase count. With the simulator, we ran simulations of fio[7] generated I/O traces with various garbage collection algorithms while having different over-provisioning sizes to study their impact on SSD performance and lifespan.

2. PROBLEM DESCRIPTION

A SSD has B physical blocks, each block contains P physical pages. There are also logical blocks which are blocks on the OS side. In our model the size of logical blocks and physical pages are equal (but can be configured arbitrarily). To write a logical block to the SSD, it would be assigned to a physical block to store the data. This translation between OS operations on logical blocks to inner-SSD operations on physical blocks and physical pages, is implemented in a software layer called the Flash Translation Layer (FTL). We implemented a page-mapped FTL, which means that we keep a map between logical blocks and physical pages to find the right physical page whenever we want to write, update, read, or discard a logical block. Although it could also be done in the physical block level or hybrid form, by implementing the address mapping in the page level, it potentially achieves the best I/O performance. Since a SSD reserves blocks for GC and bad block management, its advertised capacity as viewed by the logical address space is smaller than its physical capacity. We call the proportion of difference between logical and physical capacity, the Over-Provisioning (OP) factor. Hence, the logical address space only contains $BP(1 - OP)$ logical blocks. The value of the OP can be configured as well.

A physical page can be valid, invalid, or erased at any time. Initially, the status of all pages is erased. A physical page should be erased to be able to write data on it. We cannot write on valid or invalid physical pages. In a SSD we cannot erase a single physical page. The only option is to erase the whole physical block, which means the status of all physical pages in it would be reset to erased. After writing data on an erased physical page, it becomes a valid physical page. To update a logical block, we should assign it to a new physical page and write the new data on it. The old physical page would be marked as invalid then, and it wouldn't be mapped to the logical block anymore. To discard a logical block, we need to unassign it, and mark the old physical page as invalid. We assume that erasing a physical block can be completed instantly, such that arrivals do not queue up to be placed. The arrivals of new logical blocks which must be written to storage can be described by arbitrary process and cpu, but we assume that the number of valid pages never exceeds $BP(1 - OP)$ such that an arriving logical block can always be stored.

With page-mapping FTL, we consider the following write and GC implementations, which are also used by the referenced analytical studies. At any time, there is one special physical block for each cpu, called the Write Frontier (WF) (Different variable name in the implementation: curren_state.block). All writes related to a specific cpu, such as external writes (Workload) and internal writes (GC), are directed to the corresponding WF. Logical blocks are sequentially written to the corresponding WF. When all erased physical pages in a WF are used up, the WF is sealed and no more writes for that specific cpu are permitted until an erased physical block is selected as the new WF for that cpu. Hence, a SSD contains these three types of physical blocks: (1) the WF (containing valid, invalid, and erased physical pages), (2) sealed physical blocks (with no erased physical pages), and (3) erased physical blocks (with all physical

pages being erased). When the average number of invalid physical pages per sealed physical block passes a certain threshold (which should be configured), one of the various types of GC will be triggered to perform the following steps: (1) select a sealed physical block, (2) write (copyback) all valid physical pages from the selected sealed block to the WF belonging to the same cpu as the sealed physical block, and (3) erase the selected physical block. To fully utilize the available spare physical blocks, one can set the threshold as small as possible, so GC is triggered only when the SSD no longer contains any erased physical block after a WF is allocated, and GC stops when at least one erased block is reclaimed.

We should mention that OP is also a critical parameter to GC performance and has a significant impact on the performance of SSD accordingly. Decreasing OP leads to having more available storage space, but it increases write amplification factor since each physical block contains fewer valid physical pages on average. We will show this by comparing the resulting statistics of the tests we have done.

3. PERFORMANCE ANALYSIS

Here we'll demonstrate the simulation results from two block traces generated with similar configurations on two different sized drives:

The first I/O trace was generated using fio[7] with the random read-write pattern on a 128MB logical size virtual SSD. And the second I/O trace was generated with the same configurations on a 256MB logical size virtual SSD. Both trace files contain I/O traces from two CPUs.

The result in figure 1, contains the simulation result of running greedy and FIFO algorithms with an over-provisioning region of 7%, 14%, and 28% respectively, for a 128MB logical size, 4k block size, and 512 sector size drive configuration. The different statistics are plotted against the over-provisioning percentage.

Results in figure 2 were obtained with a similar configuration of figure 1, from a 256MB logical size drive I/O trace.

From the plots, we can see that with a higher over-provisioning region size, both greedy and FIFO algorithms are performing better. A larger over-provisioning size yields more physical blocks in the drive, so there are more buffer spaces for GC algorithms to do copyback before erasing blocks. In some edge cases when there are only a few pages left in the entire drive, and the user is trying to update an old file, a drive with zero over-provisioning size will need to perform garbage collection for every few pages being written to allocation new block for incoming operations because the current block is almost filled with copyback content after each garbage collection, which generates a huge amount of erasing operation and impact the SSD's performance and lifespan. On the other hand, over-provisioning will allocate more physical blocks than the logical size, so the GC algorithms could have buffer space to accommodate the copyback data and yield much less erase operations in the edge case mentioned above. From the simulation results, we can see that the erase count is lowered and the write amplification factor is smaller, which has a positive impact on the SSD lifespan, and because we have a lower total copyback count, less of the drive resource is spent on garbage collection, which would positively impact the performance of SSD.

We can also see that the greedy GC algorithm yields a better result than the FIFO GC algorithm in general. It searches for the block with the least valid pages to perform garbage

collection[1,2,3], which minimizes the amount of copyback needed for each GC operation and essentially frees more physical pages for incoming data.

However, the runtime of the greedy GC algorithm is worse than the FIFO algorithm, for each garbage collection operation, the greedy algorithm needs to iterate through every written block and look for one with the least number of valid pages, which has a time complexity of O(n), where n is the number of blocks. The FIFO algorithm yields a much better average runtime because it's doing garbage collection on the first unerased written block, which has a time complexity of O(1).

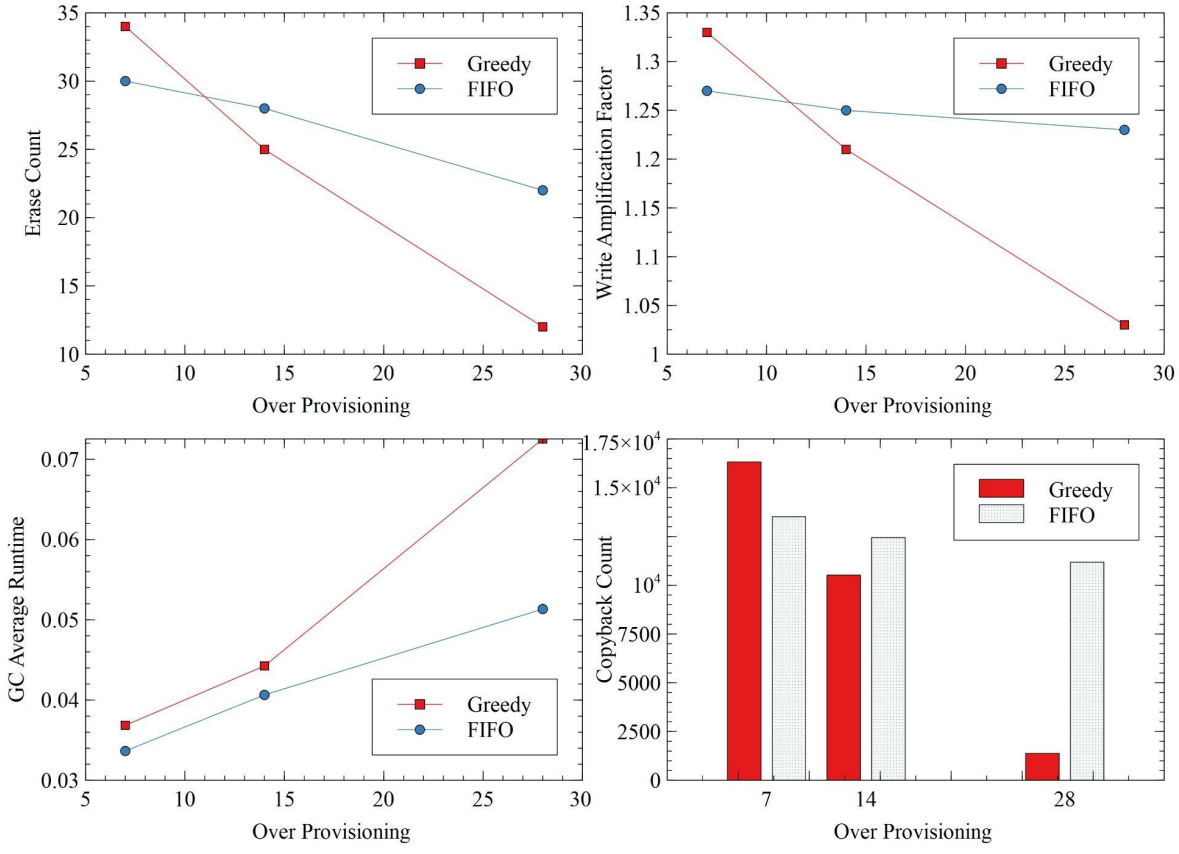


Figure 1

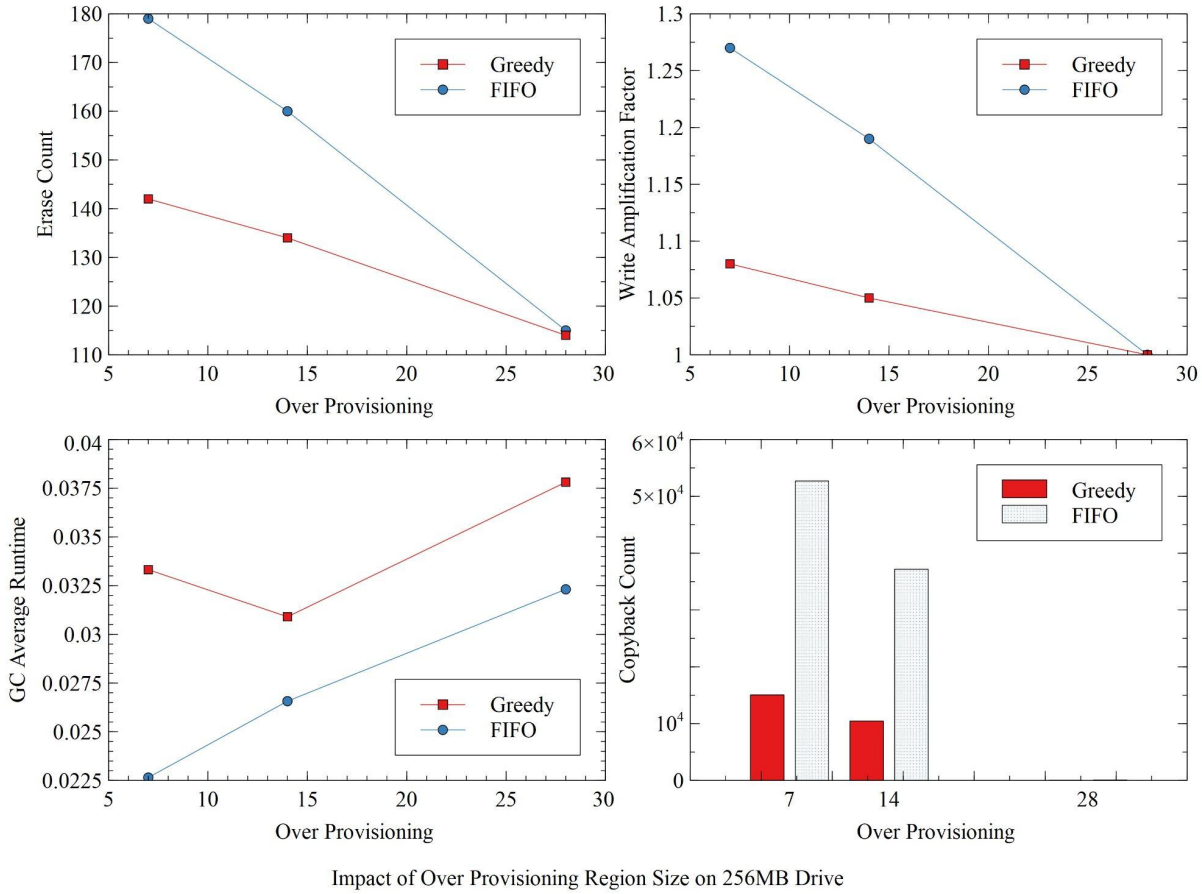


Figure 2

In figures 3 and 4, the window GC algorithm simulation statistics are plotted against the window size, together with the statistics of the greedy algorithm and FIFO algorithm under the same drive configuration, 7% over-provisioning size.

Window GC algorithm tradeoff some of the performance of greedy GC algorithm for the resource efficiency of FIFO GC algorithm, to achieve a balance of performance and simplicity. From the plots, we can see that the performance of the window GC algorithm improves with higher window size. As the window size increases, a larger subset of blocks is used to find the block with the most invalid pages, which has a higher chance to contain a block with a higher

invalid page count. When the window size is the same as the total flash size, we achieved a similar result as the greedy algorithm at the cost of average run time, since they are both performing the greedy GC algorithm on a global scale.

During experiments, we found out that the window size for the window GC algorithm can't be too small as well. When the written content size is bigger than the configured window size, every block in the window would have zero invalid pages. If we perform garbage collection on one of the valid blocks, the current block will be filled and then pushed into the window, therefore falling into an infinite loop of copyback and erase. This would also wear out blocks in this subset quicker than other blocks.

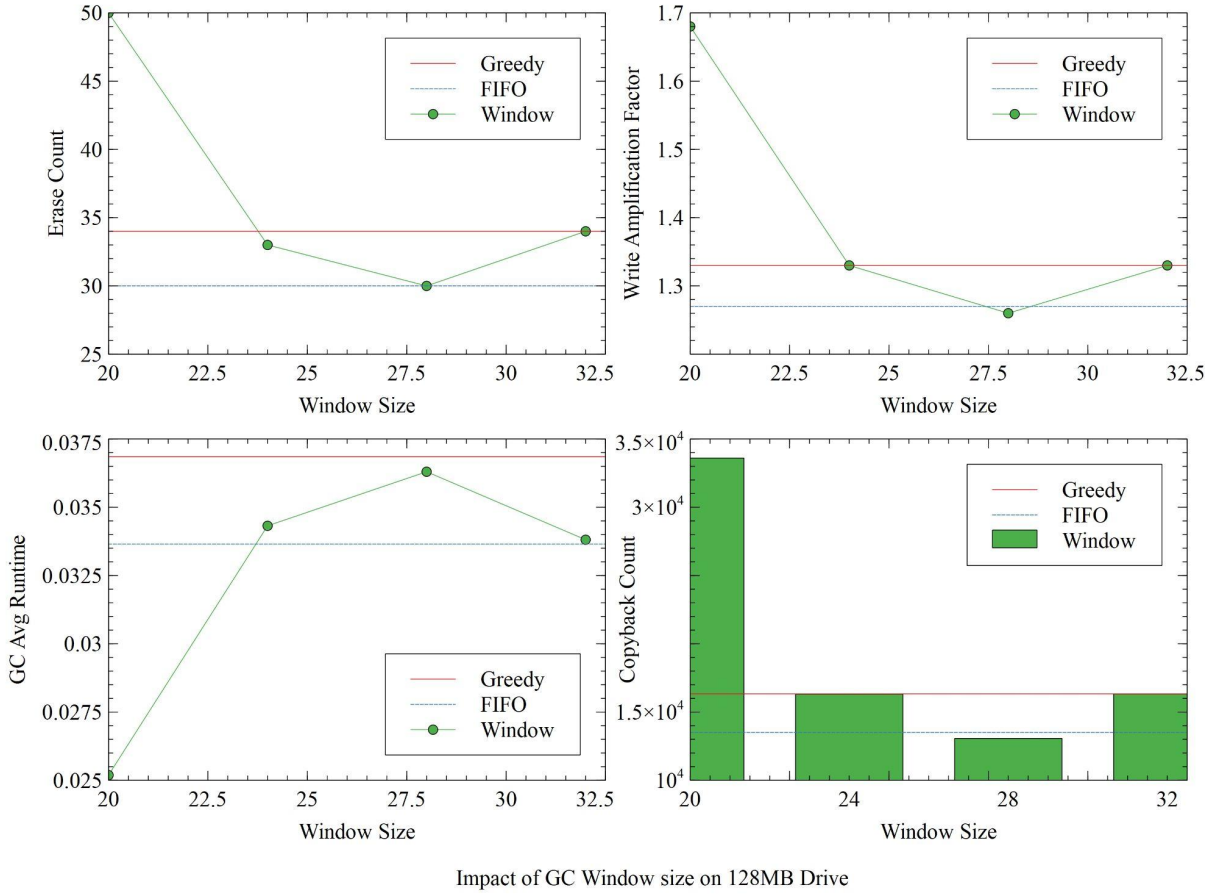


Figure 3

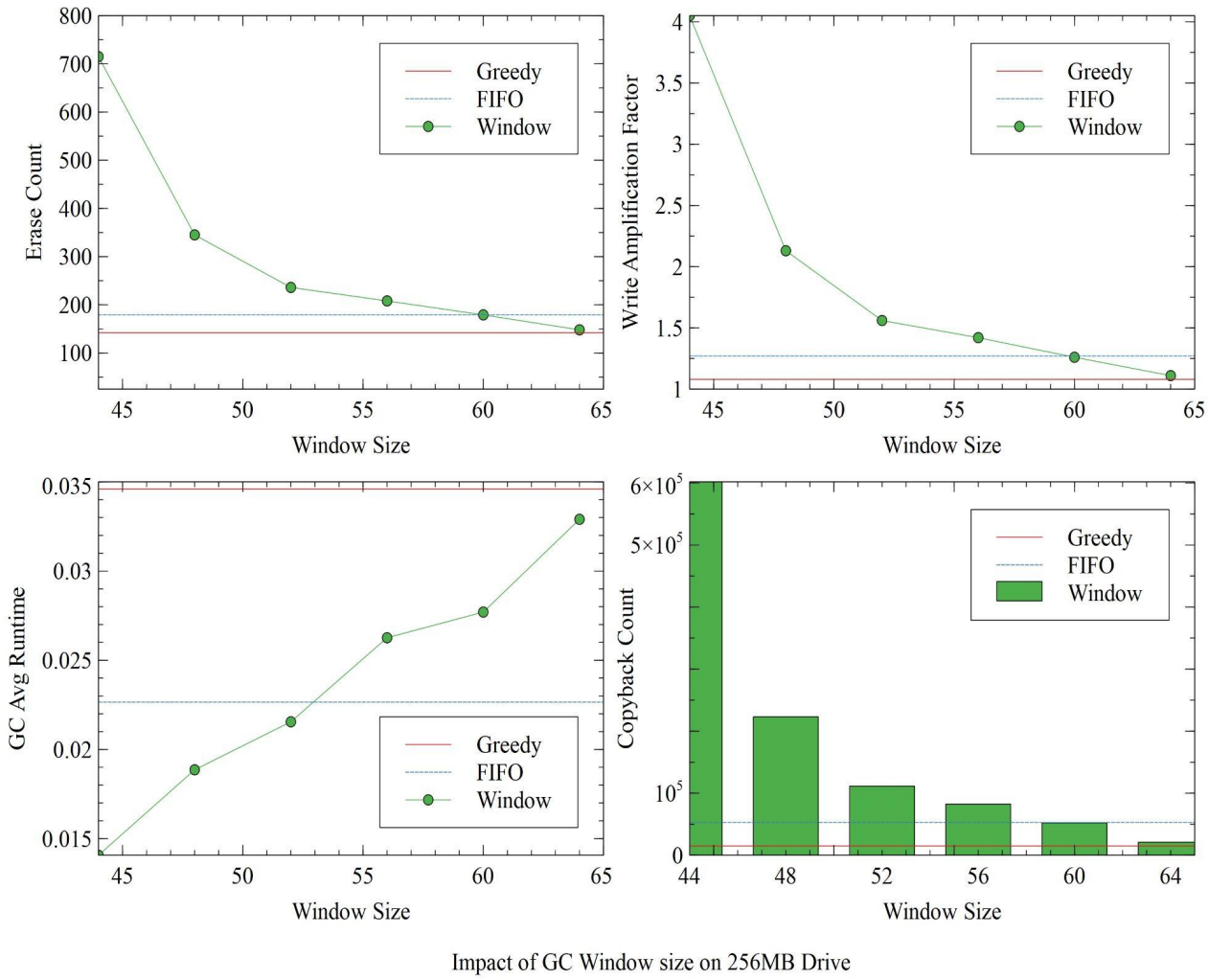
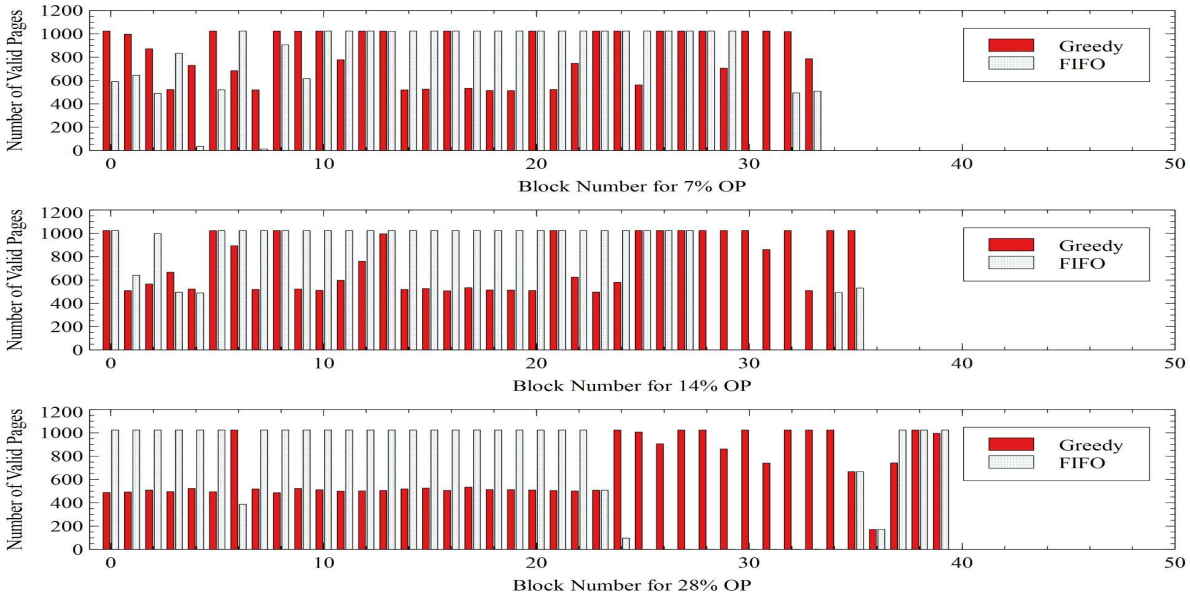


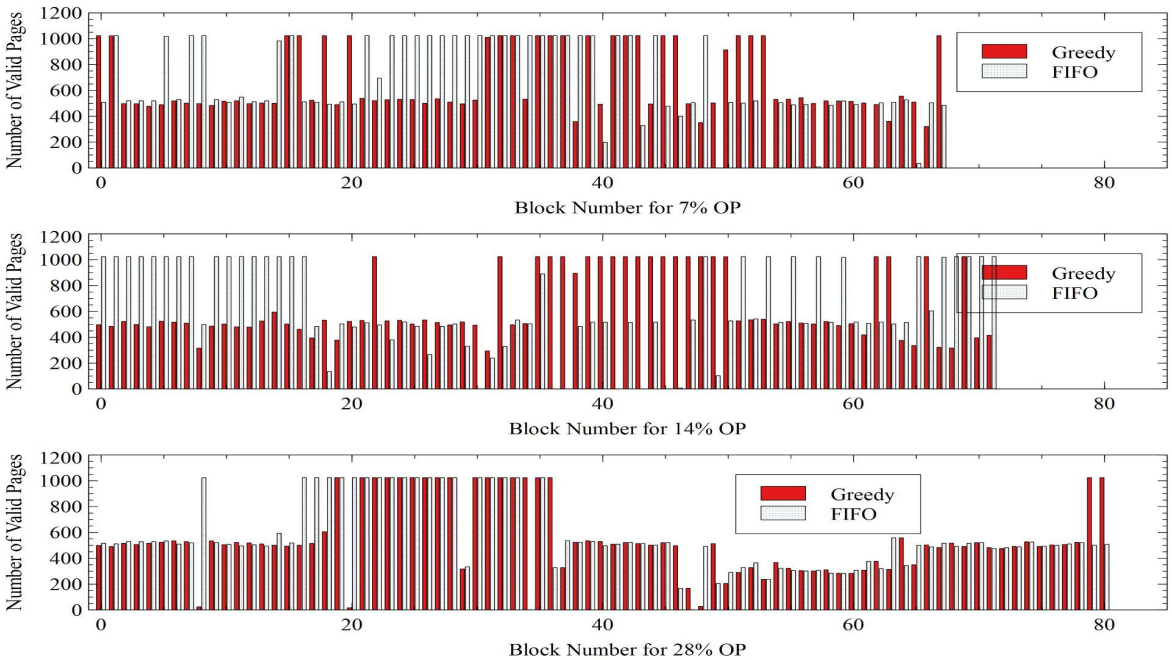
Figure 4

In figures 5 and 6, we have the distribution of valid pages across all physical blocks after each simulation with different GC algorithms and over-provisioning sizes. These histograms show the characteristic of the I/O trace data that was used to run the above simulations.



Impact of OP size and GC Algorithm on Valid Page Distribution for 128MB Drive

Figure 5



Impact of OP size and GC Algorithm on Valid Page Distribution for 256MB Drive

Figure 6

4. CONCLUSION

We developed a Page-Mapping Flash-Based SSD simulator in C++ that gets a trace block and SSD's configuration as input and outputs desired statistics such as R/W/D count, GC stats, copyback counts, WAF, etc. for each cpu and in total. We ran simulations of fio[7] generated I/O traces and configured different values for OP size, and applied various types of GC such as Greedy, FIFO, and Window to observe their impact on the performance of SSD and its lifespan. By comparing the results, we saw that with a higher OP region size, both Greedy and FIFO algorithms performed better. The Greedy algorithm yields a better result than the FIFO in general. And finally, the performance of the window GC algorithm improves with higher window size, however this size cannot be too small.

5. FUTURE WORK

One of the ideas that looks interesting to implement on top of this project, is to separate hot and cold logical blocks [2]. A hot logical block is one that gets updated frequently, and the rest are cold. If we put corresponding physical pages of cold logical blocks in the same physical blocks, we don't need to copyback cold ones whenever we update a hot one, so the load of work for GC would decrease significantly. To implement this idea we can keep track of the number of copybacks per logical block. Whenever a logical block's number of copybacks meets a certain threshold, we'll consider it as hot. We can multiply the threshold by a certain factor (e.g. a coefficient of the average rate of copybacks per second) every once in a while to avoid considering too many of them as hot. We would also have a specified WF for cold logical blocks per each CPU. Whenever we want to update a cold logical block, we copyback its data to the next available page of the cold WF of the CPU that is doing that operation.

The other possible extension we can develop is to make the project multi-threaded to have a/some specific thread(s) for GC and time the I/O operations to be able to show the real impact of various values for GC threshold.

REFERENCES

[1] W. Bux and I. Iliadis. *Performance of greedy garbage collection in flash-based solid-state drives*. Performance Evaluation, 67(11):1172–1186, 2010.

[2] B. Van Houdt. *Performance of garbage collection algorithms for flash-based solid-state drives with hot/cold data. Performance Evaluation*, 70(10):692–703, 2013.

[3] P. Desnoyers. *Analytic modeling of SSD write performance*. In Proceedings of International Systems and Storage Conference (SYSTOR 2012), 2012.

[4] L. Xiang and B. Kurkoski. *An improved analytical expression for write amplification in NAND flash*. In International Conference on Computing, Networking, and Communications (ICNC), pages 497–501, 2012.

[5] J.T. Robinson. *Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning.* SIGOPS Oper. Syst. Rev., 30(4):29–32, October 1996.

[6] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. *Write amplification analysis in flash-based solid-state drives*. In Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, SYSTOR ’09, pages 10:1–10:9, New York, NY, USA, 2009.

[7] Flexible I/O tester rev. 3.27. *Welcome to FIO's documentation! - fio 3.27 documentation.* (n.d.). Retrieved December 22, 2021, from https://fio.readthedocs.io/en/latest/index.html

[8] Park, C., Lee, S., Won, Y., & Ahn, S. (2017). Practical implication of analytical models for SSD write amplification. In ICPE 2017 - Proceedings of the 2017 ACM/SPEC International Conference on Performance Engineering (pp. 257-262)

[9] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. ACM Trans. Embed. Comput. Syst., 6(3), July 2007.