Write Amplification Analysis in Flash-Based Solid State Drives

Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, Roman Pletka
IBM Research
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{xhu,ele,rha,ili,rap}@zurich.ibm.com

ABSTRACT

Write amplification is a critical factor limiting the random write performance and write endurance in storage devices based on NAND-flash memories such as solid-state drives (SSD). The impact of garbage collection on write amplification is influenced by the level of over-provisioning and the choice of reclaiming policy. In this paper, we present a novel probabilistic model of write amplification for log-structured flash-based SSDs. Specifically, we quantify the impact of over-provisioning on write amplification analytically and by simulation assuming workloads of uniformly-distributed random short writes. Moreover, we propose modified versions of the greedy garbage-collection reclaiming policy and compare their performance. Finally, we analytically evaluate the benefits of separating static and dynamic data in reducing write amplification, and how to address endurance with proper wear leveling.

Categories and Subject Descriptors

B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*formal models, simulation*; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.4.2 [**Storage Management**]: Garbage collection

General Terms

Design, Performance, Algorithms

Keywords

Solid State Drives, Solid State Storage Systems, Write Amplification, Flash Memory

1. INTRODUCTION

The advent of solid-state drives (SSD) based on NAND-flash memories is currently revolutionizing the primary storage computer architecture, ranging from notebooks to enterprise storage systems. These devices provide random I/O performance and access latency that are orders of magnitude better than that of rotating hard-disk drives (HDD). Moreover, SSDs significantly reduce power consumption and dramatically improve robustness and shock resistance thanks to the absence of moving parts.

NAND-flash memories have unique characteristics that pose challenges to the SSD system design, especially the aspects of random write performance and write endurance. They are organized in terms of blocks, each block consisting of a fixed number of pages, typically 64 pages of 4 KiB each. A block is the elementary unit for erase operations, whereas reads and writes are processed in terms of pages. Before data can be written to a page (i.e., the page is programmed with that data), the page must have been erased. Moreover, NAND-flash memories have a limited program-erase cycle count. Typically, flash chips based on single-level cells (SLC) sustain 10^5 and those based on multi-level cells (MLC) 10^4 program-erase cycles.

Flash memory uses relocate-on-write – also called out-of-place write – mainly for performance reasons: If write-in-place is used instead, flash will exhibit high latency due to the necessary reading, erasing, and reprogramming (writing) of the entire block in which data is being updated.

However, relocate-on-write necessitates a garbage-collection process, which results in additional read and write operations. Whereas the reclaiming policy that selects the blocks to garbage-collect in Sprite LFS [13] was based only on the amount of free space to be gained, the policy defined in [10] also included the time elapsed since the last writing of the block with data. In general, the objective is to keep at a minimum the number of valid pages in the blocks selected for garbage collection. The efficiency of garbage collection could, for instance, be improved by delaying those blocks holding data being actively invalidated. The number of read and write operations resulting from garbage collection depends on the number of valid pages in the block.

In contrast to disks, flash memory blocks eventually wear out with progressing number of program-erase cycles until they can no longer be written. Wear-leveling techniques are therefore used to exhaust the program-erase cycles available (i.e., the cycle budget) of as many blocks as possible, in order to serve the largest number of user writes (or host writes), thereby maximizing endurance. Their performance is measured by the total unconsumed cycle budget left when garbage collection can no longer return a free block. Note

SYSTOR'09, May 4-6, Haifa, Israel

that retention is another issue that can be addressed by wear leveling as well.

Assuming independent and uniformly distributed random short writes, the optimal wear-leveling technique consists of wearing out all blocks over time as uniformly as possible. This can be achieved, for instance, by minimizing the delta between maximum wear and average wear over all blocks, with this delta corresponding to the wear-leveling inefficiency as described in [6].

In reality, host writes are not uniformly distributed. If a distinction can be made between blocks with static data (i.e., addresses to which the host only infrequently rewrites data) and blocks with dynamic data (with frequent rewrites), wear leveling can benefit from treating these two types of blocks differently instead of wearing them out uniformly. Hence, in this case, wear-leveling performance not only depends on the unconsumed cycle budget, but also on the number of cycles wasted by repeatedly moving unmodified static data.

In all cases, wear leveling causes additional read and write operations. Therefore, in flash, write amplification corresponds to the additional writes caused by garbage collection and by wear leveling. Hence, the total number of user writes that can be served depends on the total cycle budget available, write amplification, and the eventual unconsumed cycle budget due to wear-leveling inadequacy.

Finally, the management of out-of-place updates involves a mapping between logical block[1] addresses (LBA), i.e., the user (or host) address space, and physical block addresses (PBA). This mapping may be used to distinguish dynamic from static data.

In the remainder of this paper, we present a probabilistic analysis of write amplification in log-structured flash-based SSDs. The analysis assumes a windowed greedy reclaiming policy, which is a variation of the age-threshold-based policy described in [10]. Write amplification is derived assuming 4 KiB independently and uniformly distributed user write requests. The analytical results are confirmed by simulation results. Analytical results are then extended to the case in which static and dynamic data can be distinguished.

This paper is organized as follows. Section 2 reviews the relevant work in garbage collection and flash wear leveling. In Section 3, we introduce an analytical model based on a probabilistic approach and then continue with a description of our flash storage simulator in Section 4. Based on these two sections, we present essential analytical and simulation-based numerical results that allow us to quantify write amplification in Section 5 before we extend our analytical model to the more realistic scenario with static and dynamic data (Section 6). Finally, a discussion of our results concludes the paper.

2. RELATED WORK

To group I/O operations and improve performance, log-structured file systems, arrays, and disks [12, 13, 16, 9] all write new as well as updated data to new locations instead of writing it in place, and hence also depend on efficient garbage collection. Kawaguchi et al. [7] showed that in a flash-based log-structured file system garbage collection has a significant impact on performance when utilization is high. A common strategy for garbage collection is the greedy reclaiming policy [3], in which the block that has the largest number of invalid pages will be recycled. However, as garbage collection also contributes to using up the cycle budget of blocks, it is usually beneficial to combine it with wear leveling.

Various such combined algorithms have been proposed. The one described by Chang et al. [3] avoid unnecessary reclamations in garbage collection and combines this with wear leveling in the form of a periodical task that performs a linear search for blocks with a small erase count to identify blocks to be recycled. Agrawal et al. [1] describe another combined algorithm called modified greedy garbage-collection strategy. The algorithm generally selects the block with the most invalid pages for garbage collection, while avoiding a large spread in the remaining cycle budget among all blocks and limiting the frequent movement of static data. Such a strategy is referred to as static wear-leveling in [14, 4] and exhibits a four fold improvement in endurance over a strategy that does not relocate static data (assuming 75% of dynamic and 25% of static data). Ben-Aroya et al. [2] performed a worst-case competitive analysis with focus on endurance-based randomized algorithms. To achieve nearly ideal endurance, they suggest separating garbage collection from wear-leveling.

Initially, write amplification has been studied by Rosenblum et al. [13] for log-structured file systems as a function of disk utilization. Whereas the Sprite LSF analysis distinguishes between hot and cold data (including reads and writes), we instead distinguish between static and dynamic data as only writes contribute to write amplification. Furthermore, the Sprite LSF write-cost comparison includes time for seeks, rotational latency, and cleaning costs. Therefore, our results can only be qualitatively compared to those from Sprite LSF.

Although some of the flash memory papers briefly mention performance impacts from garbage collection and wear leveling, we did not find a detailed analysis of write amplification in flash-based storage systems and how it relates to parameters, such as the spare factor, in the literature.

3. WRITE AMPLIFICATION ANALYSIS

In this section, we first introduce a generic architecture for a log-structured SSD with a windowed greedy reclaiming policy for garbage collection and then propose a probabilistic approach for analyzing the write amplification factor in log-structured flash-based SSDs based on this policy.

3.1 Architecture of a Log-structured SSD

The generic architecture for a log-structured SSD consisting of a controller with some DRAM and a pool of flash memory chips can be described as follows: the entire flash memory space is organized in terms of blocks, with each block containing a fixed number n_p of pages, typically $n_p = 64$, and the size of each page s_p being 4 KiB. User data pages, addressed by LBAs, are written to the free space of flash memory, addressed by PBAs. When a flash memory page has been written, it is no longer available for writ-

[1]Note, that the term "block" used here does not correspond to a flash memory block. It is a term used in HDDs, referring to the smallest addressable data unit which typically has 512 Bytes. In the flash context, it refers to a fraction of a page.

ing until its block is erased. The controller maintains an LBA-PBA map, and when a page addressed by an LBA is updated, a free flash memory page will be allocated to store the new data. The corresponding LBA entry in the LBA-PBA map will be modified accordingly: the LBA is mapped to the new PBA and the old PBA is marked as invalid data page. Note that the LBA-PBA map can be maintained in DRAM or in flash memory. However, the latter causes additional read and write operations and hence contributes to write amplification. As this is implementation specific, these additional operations are not taken into account here. Also, the implementation of an efficient LBA-PBA map is out of scope of this paper.

If the write workload is strictly sequential in the sense that all data is updated in sequential order of LBAs, there is no need for complex garbage collection because flash blocks are being invalidated block by block as write requests proceed; one can simply erase the block containing no valid data and thus avoid the burden of relocating valid data pages.

In the case of a random write workload, and after processing a large number of page writes, the number of free pages in flash memory becomes low. Garbage collection then reclaims space blocked by invalid pages that are scattered over blocks. Once a block to be reclaimed has been selected, all valid pages in that block are relocated into a new block with free pages. The selected block can then be erased, and all n_p pages on that block become free space again to accommodate new writes. The efficiency of garbage collection is measured by write amplification defined as follows:

Definition 1 (Write Amplification).
In a log-structured system, write amplification, A, due to garbage collection is defined as the average of actual number of page writes per user page write.

Figure 1 illustrates the concept of write amplification. Suppose that I pages have been rewritten and hence have been invalidated in a block before this block is selected for garbage collection. The block still has V valid pages, where $V + I = n_p$, that have to be relocated to another block before the block can be erased and reclaimed. In other words, in order to (re)write I user pages, the number of physical pages that have to be written is $V + I$. Therefore the write amplification is

$$A = \frac{V + I}{I} = 1 + \frac{V}{I}. \quad (1)$$

Note that in (1) the term V/I is the extra write requests due to relocation of valid pages, which we define as the write amplification factor.

Definition 2 (Write Amplification Factor).
The write amplification factor, A_f, is defined as the ratio of the average number of writes used to relocate pages to the average number of free pages gained through the garbage collection procedure.

From this definition, the write amplification factor can be written as

$$A_f = \frac{V}{I}. \quad (2)$$

A non-zero write amplification factor means that each user page write causes extra writes to relocate pages, leading on

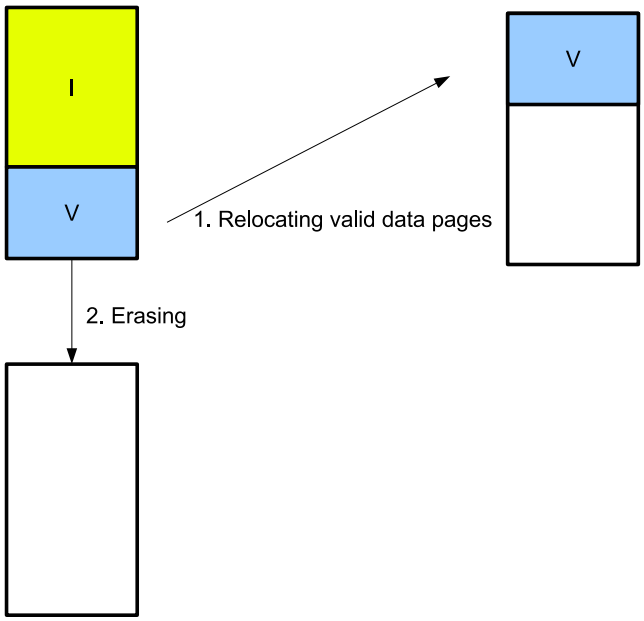

Figure 1: The concept of write amplification.

average to a total of $(1 + A_f)$ page writes. Write amplification deteriorates not only user random write performance, but also endurance. For a strictly sequential write workload, the write amplification factor is zero, i.e., there is no write amplification at all.

The reclaiming policy should attempt to minimize write amplification. Here we consider the popular greedy policy that waits until almost all free pages are exhausted and then selects the block with the least number of valid pages to be garbage-collected. The rationale behind this policy is that: The longer the reclaiming process can be delayed, the fewer valid pages will be in the block selected, hence minimizing the write amplification. The greedy reclaiming policy leads to the lowest contribution to write amplification with independently, randomly, uniformly distributed writes.

A critical factor that impacts the performance of garbage collection is over-provisioning. The idea of over-provisioning is to let the user use only a portion of the total capacity. Thus, increasing the amount of over-provisioning leads to an increase of the number of invalid pages, which improves the overall efficiency of garbage collection.

To measure the effect of over-provisioning, we introduce two terms, namely, the over-provisioning factor and the spare factor.

Definition 3 (Over-Provisioning/Spare Factors).
Suppose an SSD with a raw storage capacity of t blocks of which the user can only use a part, say, u blocks, and where $u \leq t$. Then the over-provisioning factor, O_f, is defined as $O_f = t/u$, and the spare factor, S_f, is defined as $S_f = (t - u)/t$.

Note that the over-provisioning factor and the spare factor can be used interchangeably. One can readily deduce that

$$S_f = 1 - \frac{1}{O_f}, \quad (3)$$

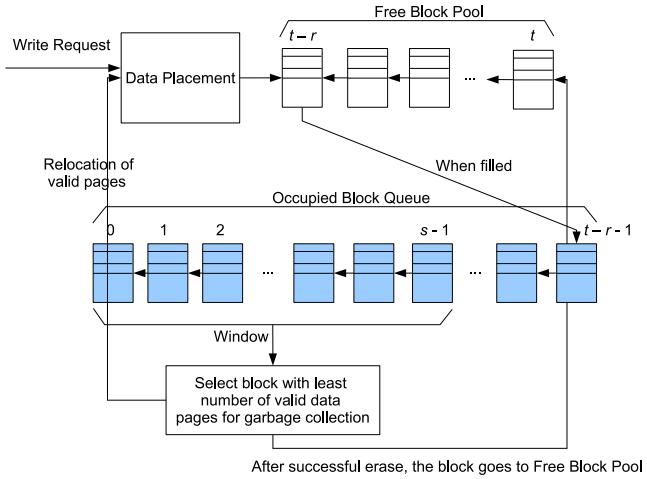

Figure 2: Schematic diagram of garbage collection with the windowed greedy reclaiming policy.

and the relationship between u and t can be represented through S_f:

$$u = t(1 - S_f). \tag{4}$$

Until all flash block have been written, the unused user LBA space can be exploited as a form of over-provisioning: as the LBA space is progressively used, the over-provisioning factor and performance decrease. However, this is not suitable for enterprise SSDs, as well as when flash is used as a cache.

3.2 The Windowed Greedy Reclaiming Policy

Figure 2 shows how flash blocks are utilized and how the free-space-reclaiming process proceeds. A pool of free blocks is used to serve both user write requests and relocate requests. Occupied blocks are maintained in a queue according to the order in which they have been written. An index is assigned to each position in the queue as shown in Figure 2, with the oldest block having index zero. Garbage collection using the greedy reclaiming policy only starts once $t - r$ blocks are occupied, with r blocks reserved so that garbage collection can operate in parallel. Normally r is very small compared to t.

The greedy reclaiming policy however would consume too many CPU cycles to select the block with the least number of valid pages from all $t - r$ blocks, because each host write may decrease the number of valid pages in already written blocks. As a result, this would require to constantly update the number of valid pages in all $t-r$ blocks. A more practical alternative is to restrict the selection process to the oldest $s(s < t - r)$ blocks only (i.e., the s blocks with the lowest indices), as they are more likely to contain the least number of valid pages. Here s is referred to as the window size for applying the reclaiming policy. This windowed greedy policy can be seen as a variation of the age-threshold-based policy described in [10], but, in contrast to the latter we consider the oldest s blocks instead of blocks older than a given age threshold. Our scheme can be seen as adaptive version of the age threshold scheme, albeit with a lower implementation complexity. Note that for $s = 1$ this scheme reduces to the circular buffer scheme.

3.3 Probabilistic Analysis

For analysis purposes, we assume that each write request has a fixed request size of a single page, with the address following a uniform random distribution in the space $[0, un_p - 1]$.

Without loss of generality, we assume that pages in the occupied blocks have been written sequentially within the block. Suppose that block b within the window from block 0 to block $(s - 1)$ has the least number of valid pages and is selected for garbage collection. According to definition 2, the write amplification factor can be calculated as the average of valid pages divided by the average of invalid pages in the selected block b. Denote by p_0^*, p_1^*, ..., $p_{n_p}^*$ the probabilities that the selected block b has 0, 1, ..., n_p valid pages, respectively. The write amplification factor, A_f, can then be computed by

$$A_f = \frac{\sum_{k=0}^{n_p} k p_k^*}{n_p - \sum_{k=0}^{n_p} k p_k^*}. \tag{5}$$

If all s blocks only have valid pages, the greedy reclaiming policy will anyway garbage collect one of those blocks in order to shift the window and eventually reach blocks with invalid pages to be reclaimed. Variations of the greedy policy that efficiently address this issue are under investigation.

Suppose $V^{(0)}$, $V^{(1)}$, ..., and $V^{(s-1)}$ are discrete random variables of the number of valid pages in blocks 0, 1, ..., $s - 1$. Denote by $p(\forall_j V^{(j)} > k)$ the probability that the number of valid pages in each block j, $0 \le j \le s - 1$, is larger than k, namely,

$$\begin{aligned} p(\forall_j V^{(j)} > k) &= p(V^{(0)} > k, V^{(1)} > k, \cdots, \\ &\qquad V^{(s-1)} > k), \end{aligned} \tag{6}$$

where $0 \le k \le n_p$. For $s << u$, which is typically the case in a practical system, we can assume that the joint probability from Equation (6) can be approximated by the marginal probabilities[2]

$$\begin{aligned} p(\forall_j V^{(j)} > k) &= p(V^{(0)} > k, V^{(1)} > k, \cdots, \\ &\qquad V^{(s-1)} > k) \\ &= p(V^{(0)} > k) p(V^{(1)} > k) \cdots \\ &\qquad p(V^{(s-1)} > k) \qquad (7) \\ &= \prod_{j=0}^{s-1} p(V^{(j)} > k). \qquad (8) \end{aligned}$$

Observe that $p(\forall_j V^{(j)} > k - 1)$ can be rewritten as

$$p(\forall_j V^{(j)} > k - 1) = p_k^* + p_{k+1}^* + \cdots + p_{n_p}^*, \tag{9}$$

[2]In general, the $V^{(j)}$ are not independent because the total number of valid pages cannot be more than un_p. Also, the cleaning process may tend to produce correlated values by cleaning pages with certain characteristics. Likewise, higher-level file system operations might cause similar dependencies.

so that one can compute p_k^* by

$$p_k^* = p(\forall_j V^{(j)} > k-1) - p(\forall_j V^{(j)} > k), \quad (10)$$

for k=1, ..., $n_p - 1$. For $k = 0$,

$$p_0^* = 1 - p(\forall_j V^{(j)} > 0),$$

and for $k = n_p$,

$$p_{n_p}^* = p(\forall_j V^{(j)} > n_p - 1),$$

as $p(\forall_j V^{(j)} > n_p) = 0$.

We now proceed to evaluate $p(V^{(j)} > k)$, for $k = 0, 1, \ldots, n_p - 1$. Denote by $p_j(m)$ the probability that the j-th block has m valid pages, then

$$p(V^{(j)} > k) = 1 - \sum_{m=0}^{k} p_j(m). \quad (11)$$

To evaluate $p_j(m)$, we first consider $p_{i,j}$, the probability of the i-th page on the j-th block being valid, where $0 \le i \le n_p - 1$ and $0 \le j \le s-1$. Suppose that $h(j)$, a function of j, is the number of pages being written after the j-th block up to the $(t-r-1)$-th block, at which point garbage collection is triggered, satisfying our initial assumption that the LBA of each page write is randomly, uniformly and independently distributed. In other words, the i-th data page on the j-th block has a probability $1/un_p$ of being invalidated independently by each of $h(j) + (n_p - i - 1)$ page writes. Thus $p_{i,j}$ can be computed as

$$\begin{aligned} p_{i,j} &= \big(1 - \frac{1}{un_p}\big)^{[h(j)+(n_p-i-1)]} \\ &\approx \big(1 - \frac{1}{un_p}\big)^{h(j)}, \end{aligned} \quad (12)$$

where the approximation has been made because $n_p - i - 1$ is much smaller than $h(j)$. As in its approximation $p_{i,j}$ is independent of i, we denote it by p_j, and rewrite (12) as

$$p_j = \big(1 - \frac{1}{un_p}\big)^{h(j)}. \quad (13)$$

As each page on the j-th block has the same probability p_j to be valid at the moment of garbage collection, the number of valid pages on the j-th block follows a binomial distribution of the parameter n_p and p_j

$$p_j(m) = \binom{n_p}{m} p_j^m (1-p_j)^{n_p - m}. \quad (14)$$

At this point we evaluate $h(j)$ in (13). One should be aware that $h(j)$ is closely related to how the first physical appearance of each individual LBA covering the entire user space is mapped onto flash memory pages on blocks 0, 1, ..., $(t-r-1)$. Note that any of the first-appearing LBA pages never invalidates any of the pages starting from the first page of block 0 up to itself. Consequently, as further discussed below, the first-appearing LBA pages written after block j do not invalidate it, therefore these pages must be excluded when calculating $h(j)$.

We consider two phases of operation: the initial and the steady-state. At the initial phase, it is assumed that before garbage collection first takes place, u blocks have been written with LBAs covering the entire user space. In other words, from block 0 to block $u-1$, no page invalidates other pages, as each one corresponds to a different LBA. After that, each page starts to invalidate the physical page corresponding to the same LBA. We refer this model as the "fixed" model, and $h(j)$ can be evaluated by

$$h(j) = \begin{cases} (t-r-u)n_p & \text{if } j \le u-1 \\ (t-r-j)n_p & \text{otherwise.} \end{cases} \quad (15)$$

Towards the end of the initial phase, the garbage collection process will be activated. Subsequently, the initial phase will gradually evolve to the steady-state phase in which the first physical appearance of LBAs is now approximated by the classic "coupon collector" model [5]. One can view each individual LBA (ranging from 0 to $un_p - 1$) as a different coupon, arriving independently and uniformly when a data page carrying an LBA "coupon" is written. Denote by c_{lba}^j the total number of LBA coupons collected between blocks 0 and j. Note that c_{lba}^j is a random variable, thus we evaluate its expectation $E(c_{\text{lba}}^j)$. Using a special case of the Generalized Birthday Problem [8] called the balls and bins model as explained in Appendix A, we obtain

$$E(c_{\text{lba}}^j) = un_p\Big(1 - \big(1 - \frac{1}{un_p}\big)^{[(j+1)n_p]}\Big). \quad (16)$$

Note that the balls and bins model only approximates the actual steady-state behavior. This can be observed by considering the case where j approaches t. It can be shown that $E(c_{\text{lba}}^j)$ does not approach un_p as one would expect.

Here, $h(j)$ can be evaluated by subtracting the number of remaining LBAs, $un_p - E(c_{\text{lba}}^j)$, that are yet to appear after the j-th block has been written, from the total number of pages subsequently written after j, to cover the entire user space.

$$\begin{aligned} h(j) = \max\Big(0, n_p(t-r-j-1) - \\ un_p\Big(\big(1 - \frac{1}{un_p}\big)^{[(j+1)n_p]}\Big)\Big) \end{aligned} \quad (17)$$

The above equation ensures that $h(j)$ does not take negative values when j approaches t.

To compute the write amplification factor, one can apply (5), (8), (10), (11), (13), (14), and (15) under random page-by-page write workload and the greedy reclaiming policy, using the fixed model, or apply (5), (8), (10), (11), (13), (14), and (17), using the coupon collector model.

4. DESCRIPTION OF THE SIMULATOR

Our flash simulator is an event-driven simulator written in Java that consists of roughly 10k lines of code. Figure 3 illustrates the architecture of the simulator. Different types of flash memories (SLC and MLC) are supported, and their read, program, and erase as well as page and block characteristics are configurable. Flash chips are modeled according to ONFi 1.0 [11]. Reads and writes are simulated without actual data being transferred. Multiple flash chips are grouped into channels, and multiple channels are attached to a flash controller.

The controller is configurable, and different types of controllers are available that support key functionalities such as wear leveling, garbage collection, bad block management,

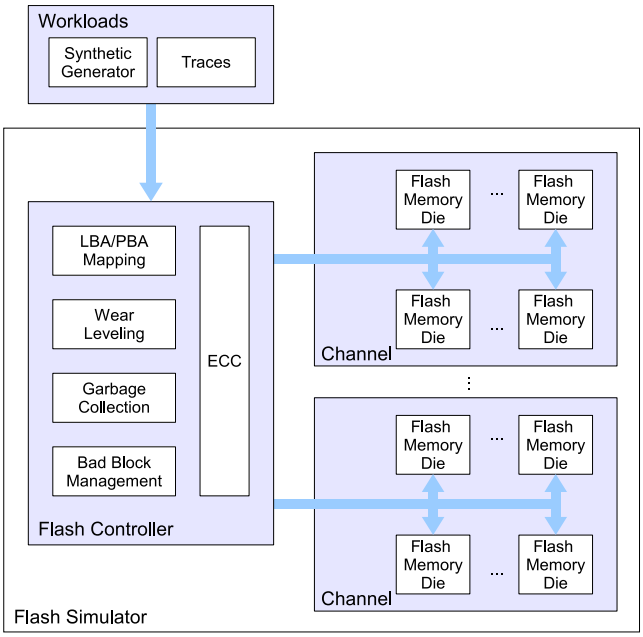

Figure 3: Overview of the simulator.

and error-correction codes (ECC). The controller uses a log-structured LBA-to-PBA map.

Reading and writing to the simulated flash storage device can be done using different types of workloads, such as uniform, exponential, and ZIPF distribution with configurable read/write ratios.

The simulator allows us to simulate system-level behavior of flash-based storage devices. More specifically, write amplification and endurance can be measured as shown in the next section. In the future, we plan to extend the simulator with further functionalities such as various types of reliability protection schemes complementing the ECC.

The simulator takes into account how chips are connected to the flash controller by the number of channels used, which has a significant impact on the I/O performance of the system, but does not affect the write amplification analysis.

Certain flash memories support a copy-back mode, a feature that allows data to be to moved in the chip using a simple command, without going through the controller and hence unnecessarily wasting bandwidth and processing cycles. The copy-back mode is mainly used for static wear leveling, which can be neglected with uniformly-distributed host writes used for the analysis in Section 3. Hence it is not supported by the simulator. In reality, when static and dynamic blocks can be distinguished, static wear leveling will have an impact on write amplification, and, the copy-back mode can, to some extent, reduce the impact of relocations on write amplification but it cannot be used to relocate data across chips.

5. ANALYTICAL AND SIMULATION RESULTS

This section presents numerical results obtained by applying the analytical results from Section 3 as well as simulation

Table 1: Parameters used to compute the write amplification factor under random page-by-page write workload and the greedy reclaiming policy.

Parameter	Notation	Value
Total number of blocks	t	400000
Reserved number of blocks	r	10
Number of pages per block	n_p	64
Window size for applying reclaiming policy	s	500

results of the steady state. As soon as all blocks have been used once and garbage collection starts, write amplification appears and rapidly reaches its steady state average. The steady state ends once the cycle budget is exhausted and the chips must be replaced.

Figure 4 shows how the write amplification factor evolves as a function of the spare factor under random page writes and the greedy reclaiming policy. According to the parameters in Table 1, three curves are shown: the simulation results, the analytical results under the fixed model, and the analytical results using the coupon-collector model. It can readily be seen that the analytical results using the fixed model are indistinguishable from those results using the coupon collector model. It can also be seen that the analytical results match the simulation results very well, except for the noticeable differences at low (< 0.2) spare factor. This suggests that for a sufficiently large spare factor both the fixed and the coupon collector model can be used to compute the write amplification factor with the greedy reclaiming policy.

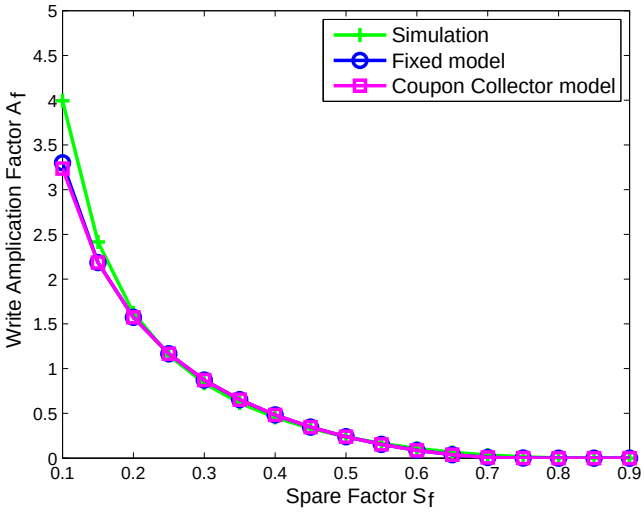

Figure 4: Write amplification factor versus spare factor under random page-by-page write workload and the greedy reclaiming policy.

Next, we examine the impact of the window size, s, on the write amplification factor. A small s is appealing in terms of reducing computational cost and latency, while potentially

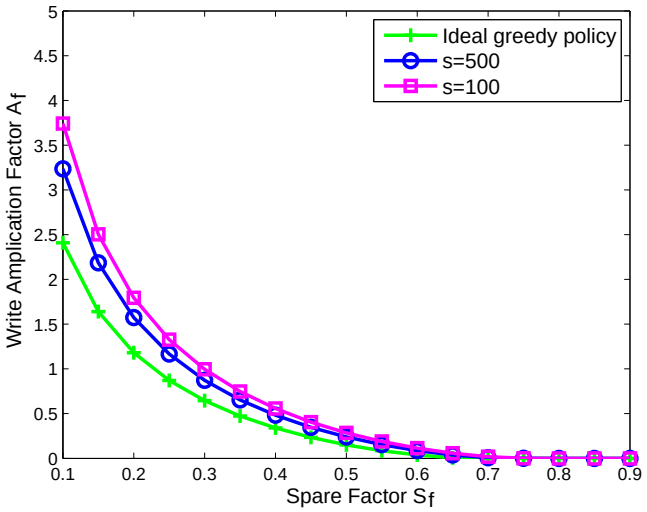

Figure 5: Analytical results of the impact of the window sized of garbage collection on the write amplification factor.

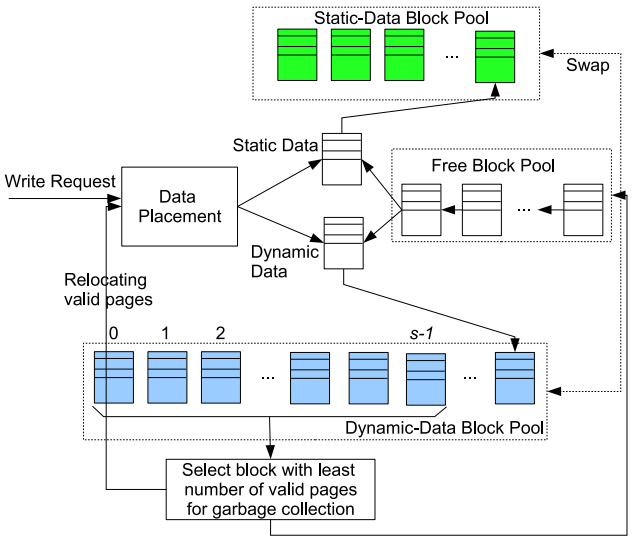

Figure 6: Schematic diagram of garbage collection with greedy reclaiming policy in the "separated" scenario.

worsening the write amplification factor. Therefore, s has to be chosen judiciously. Figure 5 shows the differences in the write amplification factor between the ideal case (with its selection window for garbage collection covering all occupied blocks) and two cases covering only the oldest s blocks, where $s = 500$ and $s = 100$. It is interesting to remark that a relatively small s is sufficient to achieve a write amplification factor close to the ideal case, particularly for a large spare factor. Finally, in the extreme case of the circular buffer, $s = 1$, we observed that the discrepancy between simulation and analytical results is larger.

6. SEPARATING DYNAMIC AND STATIC DATA

In this section we apply the analytic framework developed in Section 3 to the practical situation of static and dynamic data.

We consider the following two scenarios: In the first scenario, the flash controller ignores whether a page is static or dynamic, and thus dynamic and static pages are stored in a inter-mixed fashion into flash blocks. This scenario is called "mixed" scenario. In the second scenario, we assume the flash controller knows (either a priori or by learning) whether an LBA is dynamic or static, and then places static and dynamic pages into distinct pools of blocks, as shown in Figure 6. This scenario is thus called the "separated" scenario. The actual task of identifying static and dynamic data can be achieved by collecting statistics on writes for each LBA. We believe that an adapted version of the method introduced in [13] can achieve reasonable results. This is however out of scope in this paper.

Furthermore, we assume that whenever a block in the dynamic pool exhausts its cycle budget far more than the average of the other blocks, the block is swapped with a block in the static pool that holds static data pages only. As the swap process is done only a limited number of times for each block holding static data during its lifetime, we can neglect the additional write amplification caused by the swap process.

Denote by u_s the total number of blocks that would be used to hold static data pages. For the mixed scenario, (13) is rewritten as

$$p_j = \frac{u_s}{u} + (1 - \frac{u_s}{u}) \cdot \big(1 - \frac{1}{(u - u_s)n_p}\big)^{h(j)}, \qquad (18)$$

and then we can apply (5), (8), (10), (11), (14), (15), and (18) to compute the write amplification factor under random page-by-page write workload and the greedy reclaiming policy, using the fixed model. The reason that we use the fixed model in the mixed scenario is because of its simplicity.

Figure 7 shows the impact of the existence of static data pages on write amplification, with the parameters taken from Table 1. The write amplification factor worsens significantly if there is a considerable portion of static data pages and over-provisioning cannot effectively reduce the write amplification factor to zero. The reason is that static data pages are scattered among blocks randomly, leading to a significant amount of repeated relocations.

For the separated scenario, (13) should be modified as

$$p_j = \big(1 - \frac{1}{(u - u_s)n_p}\big)^{h(j)}, \qquad (19)$$

because static data pages are excluded from the pool on which garbage collection is running. Therefore, (17) should be changed to

$$\begin{aligned} h(j) \quad = \quad & n_p(t - u_s - r - j - 1) \\ & -(u - u_s)n_p\big(1 - \frac{1}{(u - u_s)n_p}\big)^{(j+1)n_p}. \end{aligned} \qquad (20)$$

Accordingly, we use (5), (8), (10), (11), (14), (19), and (20)

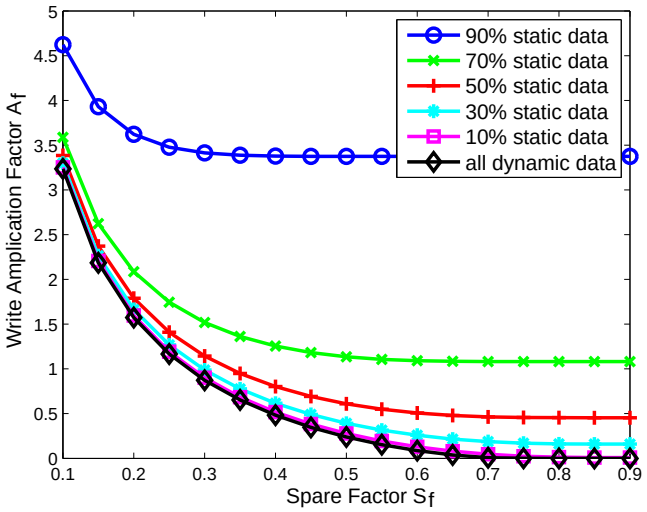

Figure 7: Impact of static data pages on the write amplification factor.

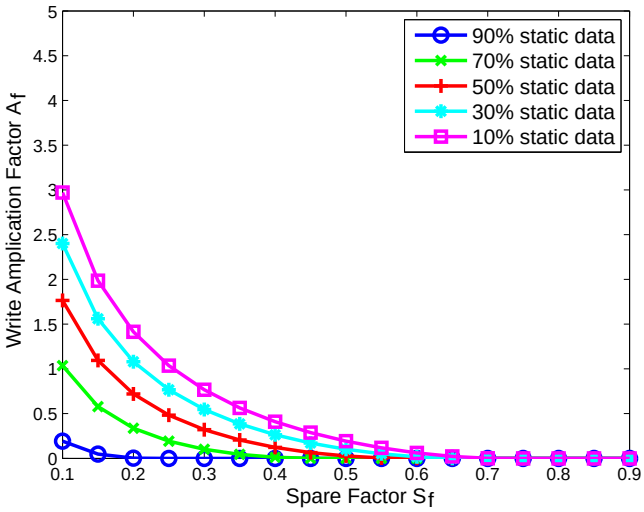

Figure 8: The effect of intelligent data placement on improving the write amplification factor in the separated scenario.

to compute write amplification factor for the separated scenario.

Figure 8 shows how the write amplification factor can be improved through intelligent data placement by separating static data pages from dynamic data pages. The curves are obtained with the parameters from Table 1. The separation of dynamic and static data pages turns out to be extremely effective in taking advantage of the existence of static data to improve the write amplification factor. The larger the portion is that the static data accounts for, the smaller is the write amplification factor, and the less over-provisioning the SSD requires. These results are qualitatively similar to those of Sprite LSF [13].

7. CONCLUSION

Write amplification is a critical factor that determines the performance of SSDs, especially with sustained workloads, as it accounts for the additional system writes that must be performed for each user write. Furthermore, write amplification affects the endurance of SSDs by accelerating the consumption of the program-erase cycle budget of flash blocks. For example, with an over-provisioning of about 10%, the write amplification factor can be as high as 4 when dealing with random short writes, hence accelerating the aging of an SSD and reducing its sustained performance by the same factor.

Therefore, this paper focused on write amplification, by proposing an analytical model of the contribution of garbage collection to write amplification, which has been validated by simulation. The important relationship between write amplification and over-provisioning has been shown, with analytical as well as simulation results. The ideal greedy reclaiming policy used for garbage collection, which is optimal in terms of write amplification, has been compared with the more practical windowed versions that perform acceptably well, especially with higher over-provisioning. In the region of typical spare factors – for instance, the 146 GiB ZEUS$^{\text{IOPS}}$ enterprise SSD from Stec [15] has a spare factor of 0.43 – and for sufficiently large window sizes, simulation results match well with our analytical model.

All the results are obtained from workloads consisting of uniformly-distributed random short writes, for which SSDs clearly demonstrate their performance advantage over HDDs. However, in reality, some portions of data are less frequently updated than others. We have shown that by distinguishing such static from dynamic data and storing it in separate blocks, the write amplification factor can be improved substantially: this is particularly evident with larger fractions of static data or low over-provisioning. Finally, although wear leveling alone contributes only marginally to write amplification as compared with garbage collection, it should be noted that wear leveling is key to achieving high endurance. Therefore, designs that efficiently address garbage collection and wear leveling are critical for SSDs, and will be addressed in future work.

8. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of the Usenix Annual Technical Conference*, June 2008.

[2] A. Ben-Aroya and S. Toledo. Competitive analysis of flash-memory algorithms. In *Proceedings of 14th Annual European Symposium on Algorithms (ESA)*, pages 100–111, Sept. 2006.

[3] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems*, 3(4):837–863, Nov. 2004.

[4] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proceedings of*

44th Design Automation Conference (DAC), pages 212–217, June 2007.

[5] P. Erdös and A. Rényi. On a classical problem of probability theory. *Magyar Tud. Akad. Mat. Kutato Int. Kozl*, 6:215–219, 1961.

[6] A. Fazio. The real story about NAND flash and solid-state drive reliability. Intel Developer Forum US, Aug. 2008.

[7] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX Technical Conference*, pages 155–164, Jan. 1995.

[8] E. H. McKinney. Generalized birthday problem. *American Mathematical Monthly*, (73):385–387, 1966.

[9] J. Menon. A performance comparison of RAID-5 and log-structured arrays. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 167–178, Aug. 1995.

[10] J. Menon and L. Stockmeyer. An age-threshold algorithm for garbage collection in log-structured arrays and file systems. In J. Schaeffler, editor, *High Performance Computing Systems and Applications*, pages 119–132. Kluwer Academic Publishers, 1998.

[11] ONFi. Open NAND Flash Interface Specification 1.0. Specification, Open NAND Flash Interface (ONFi), Dec. 2006.

[12] J. K. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, Jan. 1989.

[13] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[14] D. Shmidt. TrueFFS wear-leveling mechanism. Technical report, M-Systems, May 2002.

[15] Stec. The Zeus$^{\text{IOPS}}$ enterprise SSD.

[16] D. Woodhouse. JFFS: The journaling flash file system. In *Ottawa Linux Symposium*, July 2001.

APPENDIX

A. APPENDIX: BALLS AND BINS MODEL

Below, we use the balls and bins model to derive (16). The balls and bins model is briefly described as follows: Suppose we have m balls and n bins. We want to throw the balls into the bins, uniformly and independently. We are interested in the number of empty bins and the number of non-empty bins.

Consider the probability that a given bin is empty, when we throw m balls into n bins uniformly and independently. As each ball hits the first bin with probability $1/n$, we have

$$p(\text{first bin is empty}) = (1 - \frac{1}{n})^m.$$

Let I_j be an indicator such that

$$\begin{array}{ll} I_j = 1 & \text{if bin } j \text{ is empty} \\ I_j = 0 & \text{otherwise} \end{array}$$

Moreover, let $I{=}I_0 + I_1 + \ldots + I_{n-1}$ be the total number of empty bins, then we have

$$\begin{aligned} E[I] &= E[I_0 + I_1 + \cdots + I_{n-1}] \\ &= nE[I_0] \\ &= n(1 - \frac{1}{n})^m \end{aligned}$$

which corresponds to the expectation of the number of non-empty bins.

If we think of individual LBAs of the space used as bins, namely, $n = un_p$, and that each page write associated with an LBA is like throwing a ball into a bin identified by the LBA with a probability of $1/un_p$. Then the expected total number $E(c^j_{\text{lba}})$ of LBAs that first appeared with blocks from 0 to j is the expected number of non-empty bins, yielding (16).