RZ 3771 (# 99781) 03/31/2010
Computer Science 15 pages

Research Report

The Fundamental Limit of Flash Random Write Performance: Understanding, Analysis and Performance Modelling

X.-Y. Hu, R. Haas

IBM Research – Zurich
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date.
Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Almaden • Austin • Beijing • Delhi • Haifa • T.J. Watson • Tokyo • Zurich

The Fundamental Limit of Flash Random Write Performance:

Understanding, Analysis and Performance Modelling

Xiao-Yu Hu Robert Haas

IBM Research, Zurich

{xhu,rha}@zurich.ibm.com

Abstract

The understanding, analysis and modelling of the fundamental limit of the sustained random write performance and endurance of Flash solid state drives (SSDs) are critical for Flash SSD vendors and storage system designers and practitioners. This not only helps design high-performance Flash SSDs, but also dictates how Flash can be integrated into today's memory and storage hierarchy.

This paper analyzes the fundamental limit of the sustained random write performance of Flash SSDs. An empirical model is developed to compute the write amplification and performance slowdown factor for the greedy garbage collection policy under a pure random write workload. Potential causes of the commonly observed performance slowdown are investigated, and remedies suggested. In particular, we quantitatively demonstrate the potential for enhancing the random write performance and endurance by separating long-lived data from short-lived data inside Flash SSDs. Moreover, our theoretical results suggest a tiered storage system in which cold (i.e., infrequently used) data blocks are moved to a hard disk drive (HDD) to improve cost effectiveness and to reduce Flash memory utilization, which improves garbage collection performance.

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 (SSDs), Solid State Storage Systems, Flash Memory, Write Amplification, Garbage Collection

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

NAND Flash memory holds the promise of revolutionizing the memory and storage hierarchy in computer systems. Flash memory has been considered to augment DRAM, to extend persistent storage such as hard-disk drives (HDDs), and to form a new layer filling the gap between traditional DRAM memory and HDDs. Whether Flash memory is used to replace or to complement existing DRAM and HDDs, design challenges resulting from the unique Flash memory characteristics have to be addressed.

The read/write/erase behavior of Flash memory is radically different from that of HDD or DRAM owing to its unique erase-before-write and wear-out characteristics. Flash memory that contains data must be erased before it can store new data, and it can only endure a limited number of erase-program[1] cycles, usually between 100,000 for single-level cells (SLC) and 10,000 for multiple-level cells (MLC). Flash memory is organized in units of pages and blocks. Typically, a Flash page is 4 KiB in size and a Flash block has 64 Flash pages (thus 256 KiB).

Reads and writes are performed on a page basis, whereas erases operate on a block basis. Reading a page from Flash cells to a data buffer inside a Flash die takes 25 μs, writing a page to Flash cells takes about 200 μs, and erasing a Flash block normally takes 2 ms. Although the read and write operations take on the order of tens of microseconds, the erase operations take milliseconds. Therefore, using Flash naively, such as for write-in-place, would not yield a high performance.

To address the erase-before-write characteristics, Flash memory together with a microprocessor (controller), collectively called a Flash solid-state drive (SSD), requires a set of Flash management functions, such as the Flash translation layer (FTL), garbage collection, wear levelling and bad-block management (BBM), to perform out-of-place update, analogous to the write process in the log-structured file systems.

In particular, random writes in SSDs cause write amplification in which a single user write can cause more than one actual write, owing to background activities in SSDs. Write

[1] Writing data onto Flash is conventionally called programming.

amplification occurs mainly because SSDs write data in an appending mode, which requires garbage collection similar to a log-structured file system.

It has been reported [3, 6, 8, 14, 41, 50] that existing Flash SSDs exhibit good sequential/random read performance and sequential write performance, but suffer from poor random write performance. Moreover, it is claimed that almost all SSDs slow down with use over time, incurring a significant performance degradation. Although it might be tempting to investigate the root cause of the poor random write performance for a particular Flash SSD, it is of more importance to understand whether this common behavior is a fundamental limit or merely an implementation artifact.

Contributions: In this paper, we analyze the fundamental limit of the sustained random write performance of Flash SSDs. We focus on key contributing factors limiting the random write performance, and quantify their impact. We stress practical implications of the analytical results, reflecting on the causes of and possible remedies for the performance slowdown widely observed in existing SSDs. We extend our results to quantitatively evaluate the potential performance and endurance improvement by separating inactive from active data inside Flash SSDs or by migrating inactive data out of Flash SSDs. More specifically, this paper makes the following main contributions:

- An empirical model to analyze the random write performance as a function of utilization is developed and evaluated against simulation results. The performance model accurately approximates the write amplification of a given garbage collection strategy and helps determine the sustained random write performance and thus the slowdown from the peak random write performance. Although the model is limited to the pure random write workload, it is, to our knowledge, the first of its kind to model random write performance of Flash SSDs, and its relevance/usefulness is demonstrated by using it to quantify the potential of data placement and data migration schemes.
- A widely-known greedy garbage collection model is defined in detail and proved to be the optimal algorithm for the random write workload. It is further shown that no static wear levelling is needed for this workload. This is the first theoretical work on the analysis of Flash garbage collection policies.
- Using the proposed empirical model and the Flash SSD simulator, the potential for enhancing the random write performance and endurance by separating long-lived from short-lived data inside Flash SSDs is investigated in a qualitative manner. The idea of separation long-lived from short-lived data initially appeared in the treatment of log-structured file systems, but its impact and advantage in terms of write amplification are quantified here for the first time.
- Using the proposed empirical model, the advantage of integrating Flash into a tiered storage system is quantitatively demonstrated, wherein cold (i.e., infrequently used) data blocks are moved to an HDD to improve cost effectiveness and to reduce Flash memory utilization, which improves garbage collection performance. The concept has recently received much attention, and our work provides scientific evidence in support of the concept.

Methodology: The key obstacle to analyzing the fundamental limit of random write performance of Flash SSDs is the lack of a sensible model for hardware/firmware architecture as well as Flash management functions. There are a variety of Flash SSD designs from various vendors, featuring a broad range of hardware architectures, and each Flash SSD vendor keep its core Flash management functions, such as FTL, garbage collection and wear levelling, proprietary, making a generic performance analysis extremely difficult.

Therefore we take the following methodology: Instead of focusing on a particular Flash SSD design, we identify a common Flash SSD architecture that is sufficiently typical, and embodies the state-of-the-art of Flash SSD designs, from which we built an event-driven, Java-based Flash SSD simulator [23]. The simulator has two functional parts: a low-level and a high-level part. The low-level part is devoted to the emulation of physical Flash channels and the high-level part is responsible for firmware, such as data structures and algorithms for managing Flash memory.

The low-level part emulates the basic Flash commands, such as reading/writing a page and erasing a block, on multiple parallel Flash channels, each of which has multiple dies attached. As the rudimentary read/write/erase behavior of Flash commands is well defined by Flash memory manufactures, the low-level part turns out to be straightforward and relatively easy to develop; it captures every phase of Flash commands to a precision level of the clock. In a sister project, we developed a hardware prototype of Flash controller using the Xilinx FPGA evaluation board ML405, and by a cross-check, found that our simulator faithfully matches the read/write/erase behavior of Flash commands on the real Flash memory.

The high-level part is responsible for Flash management functions, such as FTL, garbage collection, wear levelling and bad block management. The simulator is made in a flexible way (using Java interface facility) such that various algorithms can easily be switched on and off for the purpose of testing. The simulator has been used to develop and test new algorithms for advanced Flash management functions, as well as to simulate a storage system integrating Flash memory as cache. In this work, it is used to validate the performance model we describe later.

Our key goal in this paper is not to develop a sophisticated garbage collection algorithm, but to identify and analyze the fundamental limit of Flash SSDs due to Flash's erase-

before-write characteristic. To this purpose we describe a version of greedy garbage collection algorithms in detail, and our analysis will largely be based on it for tractability reason. Although this algorithm looks simplistic and lacks features of a realistic garbage collection algorithm, almost all existing garbage collection algorithms in Flash SSDs and log-structured file systems root deeply in it.

The rest of the paper is organized as follows. Section 2 discusses related work and Section 3 gives an overview of Flash SSDs from both the architecture, hardware and firmware points of view. Section 4 presents a qualitative and quantitative analysis of the random write performance, focusing on the fundamental limit resulting from write amplification due to Flash's erase-before-write characteristic. Section 5 discusses potential causes of the commonly observed performance slowdown, and suggested remedies. Section 6 concludes the paper.

2. Related work

The performance and cost-per-GiB gaps between DRAM and HDDs are constantly growing over the past two decades, with no sign that this trend is abating. Flash memory fills the gap in the middle, which can potentially form a new tier in the current memory and storage hierarchy. Moreover, Flash memory has the potential to augment DRAM as memory expansion or HDDs as storage extension, thanks to its low power consumption. It is widely anticipated that within a few years Flash memory will likely be ubiquitously used in notebooks, servers, database systems, and storage systems.

The research on the optimal memory and storage architecture to integrate Flash memory is still in its infancy. Leventhal [37] explored the options of "Flash as a log device" and "Flash as a cache" based on Sun's ZFS file system. Graefe [20] investigated two software architectures for exploiting Flash memory: "extended buffer pool" and "extended disk". Narayanan et al. [39] analyzed a number of workload traces of data-center servers to decide whether and how SSDs should be used from storage-provisioning point of view. Wu and Zwaenepoel [52] developed eNVy as an architecture of a large non-volatile main memory system built primarily with Flash memory. Gordon [9] has recently been proposed as a system architecture for data-centric applications that combines low-power processors, Flash memory, and data centric programming systems to improve performance while reducing power consumption.

Flash-Aware File Systems: One way to use Flash memory is to put it under the control of Flash-aware file systems. Most Flash-aware files systems are inspired by the log-structured file system, which performs out-of-place write on HDDs [44]. Examples are FFS [26], JFFS for Linux [51], TFFS [19], and YAFFS [38]. A survey on Flash-aware file systems and various sophisticated data structures and algorithms to manage Flash memory can be found in [18].

Flash As Cache: In the context of hybrid HDDs, Flash memory is used as a nonvolatile cache to improve I/O performance and reduce power consumption [7, 13, 22]. Flash as cache for servers has been investigated in [27, 28], suggesting that the overall performance and reliability can be improved by splitting Flash-based disk caches into read and write regions.

Flash in Database Applications: The performance of Flash SSDs for database applications has been studied in [35, 36]. To cope with the poor random write performance of Flash SSDs, an IPL (in-page logging) scheme is proposed: changes made to a data page are not written directly, but records associated with the page are logged [34]. Koltsidas and Viglas [31] proposed a technique to dynamically place pages with read-intensive workloads on the Flash disk and pages with write-intensive workloads on a HDD. A Flash-aware data layout – append and pack – has been developed to stabilize and repair Flash SSD's performance by eliminating random writes [50]. USB Flash drives are considered to replace HDDs to perform synchronous transactional logging [15]. TxFlash, Flash memory as a SSD exporting a transactional interface to the higher-level software has been proposed [42]. Recently, FAWN, a new cluster architecture for low-power data-intensive computing centered around log-structured key-value data store using Flash memory, was proposed in [4].

Flash SSDs: Flash SSDs emulate a block-device interface similar to HDDs. Although the design and implementation of Flash SSDs are often kept as proprietary by SSD vendors, there are published research works, for example [2, 6, 17, 41] on SSD organization and hardware architecture, [21, 25, 30, 32, 33] on FTL, [2, 5, 10–12, 18, 45, 47, 49] on garbage collection and wear-levelling. Experimental results on testing the performance of Flash SSDs have been extensively studied [3, 6, 8, 14, 41, 50], highlighting the observed poor random write performance and the performance slowdown. FAB [24], CFLRU [40], BPLRU [29], and LB-CLOCK [16] are examples for using DRAM buffers to improve the random write performance of Flash SSDs.

3. SSD: A Primer

Figure 1 shows a schematic diagram of the generic architecture of Flash SSDs, which consist of multiple Flash packages connected to a controller that uses an embedded microprocessor with some DRAM for maintaining internal data structures and possibly buffering I/O requests.

Hardware: A NAND Flash memory die/chip generally uses a compact Flash interface with fewer that 30 pins, e.g. Micron products have only 24 pins, featuring a multiplexed command, address, and data bus, which typically operates at 40 MHz. The multiplexed interface is called channel when shared by multiple dies in the form of packages. This channel over which Flash packages or dies receive commands and transmit data from/to the controller is the main bottle-

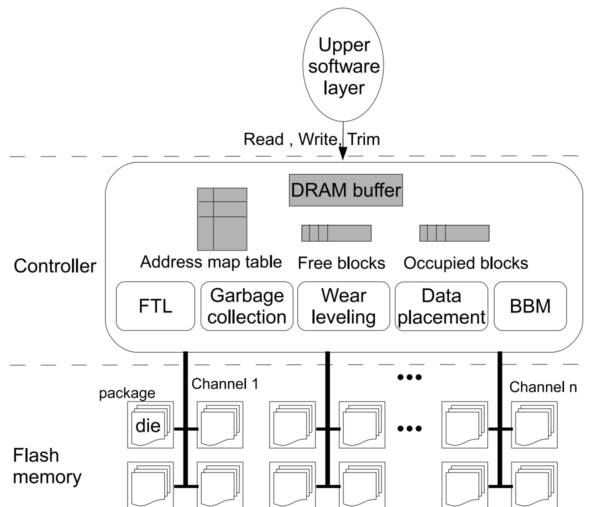

Figure 1. Schematic of the Flash SSD architecture.

neck of I/O performance per channel. For reads, it takes roughly 100 μs to transfer a 4 KiB page from the Flash data buffer to the controller in addition to the 25 μs Flash needs to upload data from Flash cells to the data buffer. For writes, it requires the same 100 μs serial transfer time per page as reads, and about 200 μs programming time. Therefore the performance suffers from the following two drawbacks: 1) I/O throughput per channel is low and 2) writes are much slower than reads.

From a hardware perspective, parallelism is a main theme of modern Flash SSDs to alleviate the bottleneck of such multiplexed Flash interfaces [17]. Multiple levels of parallelism can be used to improve the bandwidth and to reduce latency. Inside a Flash package, multiple reads and writes may proceed in parallel within different planes [2] and dies, using their own data buffers, thus allowing reads and writes to progress in parallel. Multiple group of Flash packages are channelled to the controller in parallel, with each group having its own dedicated Flash interface to receive commands and transmit data in and out. The bandwidth is proportional to the number of channels between the controller and Flash packages.

Firmware: To hide the erase-before-write characteristics of Flash memory and the excessive latency of block erases, modern Flash SSDs implement a software layer, called FTL, that performs logical-to-physical address translation, i.e., translating a logical block address (LBA) in the upper software layer to a physical address in Flash memory.

To perform out-of-place writes, the controller of the SSD maintains a data structure for the pool of free Flash blocks in which data can be written. How the data pages of free Flash blocks are allocated to service user write requests is dictated by the data placement function. The controller maintains a data structure for the pool of occupied Flash blocks in which valid and invalid pages may co-exist in the same Flash block. A complete sequence of an out-of-place write is as follows: (i) choose a free Flash page, (ii) write new data to it, (iii) invalidate the previous version (if any) of the page involved, and (iv) update the logical-to-physical address map to reflect the address change.

The out-of-place write necessitates a background routine called garbage collection (GC) to reclaim invalid pages dispersed in Flash blocks. GC selects an occupied Flash block as its victim, copies valid pages out of the block, erases it, and if successful, adds it to the pool of free blocks for subsequent use.

In addition, wear levelling may be triggered to balance the wear among blocks, and bad block management keeps track of bad Flash blocks and prevents their reuse. The controller may optionally use some DRAM as a write buffer to absorb repeated write requests and possibly to increase sequentiality of the workload.

Modern Flash SSDs exhibit impressive performance for random/sequential read and sequential write workloads. Their performance, under these three types of workloads, is essentially proportional to the number of parallel channels and also depends on the parallelism level within each channel. There is no GC need for a sequential write workload as an entire Flash block is made invalid during the write process, and it can be erased and reclaimed without any data movement. Although the erase operation takes about 2 ms and "locks" its associated plane for that time, the overall performance impact is negligible because all other planes attached to the same channel can still proceed with operations.

The interface between the upper layer software, such as operating, file and database systems, and the firmware of SSDs consists of three operations: read, write, and trim. This interface captures a broad spectrum of potential Flash usages, ranging from Flash as complementary main memory or virtual memory, to Flash as a cache, Flash as write buffer or write-ahead logging, and Flash as a disk. Although the third interface operation is not yet widely supported, our analysis shows that the use of trim can dramatically improve the efficiency of garbage collection and endurance lifetime of Flash SSDs.

4. The Fundamental Limit of Random Write Performance

Compared with their impressive sequential/random read and sequential write performance, Flash SSDs suffer from poor random write performance [3, 6, 14, 41]. Moreover, it is observed that a variety of SSDs slow down over time, and

[2] One die may contain multiple planes that can perform read, write and erase operations independently.

some even with significant performance degradation [8, 14, 50] under random-write dominated workloads.

In this section we first qualitatively analyze factors contributing to the poor random write performance, and argue that write amplification due to garbage collection is the fundamental culprit, which cannot be completely eliminated by advances in controller architecture and algorithm design. We prove that the greedy garbage collection policy is the optimal one under a specific workload model. An empirical performance model to evaluate the write amplification and performance slowdown factor is presented. We give both analytical and simulation results that illustrate the exponential-like performance slowdown as the utilization increases.

4.1 Qualitative Analysis

At the initial stage of SSD usage, a single user write results in only a single actual write, as there are still a lot of free Flash space for serving write requests without triggering GC. The peak random write performance of Flash SSDs is defined as the performance during which there are plenty of free Flash blocks ready to be written and without need for garbage collection. The peak random write performance depends on the number of parallel channels, the parallelism level within each channel, and the speed with which a Flash page can be written.

After depleting most free Flash space, GC has to be triggered to reclaim free Flash blocks to accommodate new writes. To sustain the write process, the reclaimed free Flash blocks has to match the Flash blocks being written on average. The sustained random write performance of Flash SSDs is the performance during which the number of free pages drops to the threshold that triggers garbage collection, so that on average the number of user page writes is equivalent to the number of reclaimed free pages by background garbage collection. The garbage collection process involves copying valid data pages and write these pages to another location. The phenomenon that a single user write can cause more than one actual write in Flash SSDs is called write amplification, which measures the efficiency of garbage collection. Obviously write amplification negatively affects the sustained random write performance.

The performance gap between the peak and the sustained random write performance is due to write amplification. Write amplification is a critical factor limiting the random write performance and endurance of Flash SSDs. Write amplification is commonly defined as the average of the actual number of page writes per user page write in the long term, capturing the impact of the relocation of valid pages due to the controller's housekeeping activities.

For a sequential write workload, write amplification is equal to 1, i.e., there is no write amplification. For random write workloads, there are three potential causes for write amplification: the granularity of FTL, garbage collection, and wear levelling.

4.1.1 Granularity of FTL

The granularity of logical-to-physical address mapping of an FTL scheme may adversely impact write amplification. FTL schemes can be categorized as block-level, hybrid, or page-level according to the granularity of the address map.

In the block-level FTL scheme, a group of contiguous LBAs is translated into a physical Flash block with their offset within the block being fixed, to minimize the memory requirement for storing the address map. The block-level FTL scheme, mostly seen in Smart Media cards [1], is not efficient for small random writes because a page update may require several page reads and a whole block update (termed as full merge in hybrid FTL schemes), leading to severe write amplification.

Hybrid FTL schemes, a hybrid between page-level and block-level schemes, logically partition blocks into groups: data blocks and log/update blocks, such as BAST [30], FAST [32], SuperBlock FTL [25], and LAST [33]. Data blocks form the majority and are mapped using the block-level mapping; log/update blocks are mapped using a page-level mapping. A hybrid FTL scheme has to merge log blocks with data blocks, which invokes extra pages reads and page writes, whenever no free log blocks are available. The merge operations can be classified into switch merge, partial merge, and full merge, in increasing order of write amplification overhead. Random writes inherently cause expensive full merges for hybrid FTL, inducing excessive write amplification [21].

The third type of FTL schemes is the page-level mapping scheme which eliminates the need for merge operations of the hybrid FTL, thus does not cause any write amplification in managing the address map. The page-level FTL scheme requires managing an in-memory mapping table, the size of which is essentially proportional to the raw Flash capacity. A recent improvement, the demand-based page-level FTL scheme [21], can reduce the memory requirement significantly by selectively caching page-level address mappings.

As the merge overhead is not universal to all FTL schemes, its impact on write amplification can be considered as a design and implementation artifact rather than a fundamental limit to random write performance. Although the poor random write performance of some of existing SSDs is partially attributed to the FTL merge overhead [3], we expect this situation will change as the design and implementation of SSDs mature. Hereafter, we assume that a page-level FTL scheme is used and thus there is no FTL merge overhead.

4.1.2 Garbage Collection

Garbage collection is universal to all Flash SSDs that perform out-of-place writes, wherein a user data page is always written to a free page instead of updating its previous version in place. Upon writing, the logical-physical address mapping of the FTL is updated to reflect the new location, and its previous location is marked as invalid. Under an indepen-

dently, uniformly-addressed workload with aligned 4 KiB random write requests (hereafter simply called random write workload), invalid pages will inevitably be spread through Flash blocks. Prior to free pages complete depletion, a background process, called garbage collection, has to reclaim these invalid pages by selecting a victim block, relocating its valid pages into a different block, and then erasing the victim block. The overhead of relocating valid pages leads to write amplification, a fundamental phenomenon in Flash SSDs performing out-of-place writes.

The impact of garbage collection on write amplification is influenced by the following factors: the level of over-provisioning, the choice of reclaiming policy, and the types of workloads. For convenience of analysis, we assume a pathological workload, i.e., the random write workload with aligned 4 KiB write requests, for which an optimal garbage collection policy can easily be defined.

Over-provisioning refers to a common practice that the user address space can only take a fraction of the raw Flash memory capacity. Because of out-of-place writes, over-provisioning exists in practically all Flash SSDs, either in an explicit or an implicit way. For instance, Texas Memory Systems and STEC explicitly state that their SSDs have more raw Flash memory than the maximum logical address space that the user can use, whereas other SSD vendors, such as Intel and Micron, let the user choose the size of user address space up to the maximum of the raw Flash memory capacity. In this case, over-provisioning exists implicitly and generally decreases with usage.

For both explicit and implicit over-provisioning, the user may adjust the Flash memory utilization (μ), the ratio of the number of current in-use LBA addresses (i.e. for which the SSD holds valid data) over the total physical Flash memory capacity. With implicit over-provisioning, utilization can reach up to 1.0, whereas with explicit over-provisioning, the utilization is upper-bounded by the vendor's over-provisioning specification.

The utilization is closely tied to write amplification due to garbage collection. The larger the utilization, the more likely a victim selected to be reclaimed has many valid pages that have to be relocated, and the worse the write amplification.

Given a fixed utilization and workload, write amplification depends solely on the efficiency of the garbage collection policy. In general, a garbage collection policy involves issues such as when to trigger garbage collection, which block to select as victim, and where to write the relocated valid data.

4.1.3 Wear levelling

Flash blocks can sustain a limited number of program-erase cycles, thus it is desirable to write and erase Flash blocks evenly. Wear levelling, in a broad sense, refers to an algorithm by which the controller in a Flash SSD moves data around by re-mapping logical addresses to different physical addresses of Flash memory. The frequency of this data movement, the algorithm to find the "the least worn" area to which to write, and any data swapping capabilities are generally proprietary techniques of Flash SSD vendors.

There are two types of wear levelling: dynamic and static. Dynamic wear levelling relies on the out-of-place writes to even out the wear of Flash blocks. If a Flash block holds inactive data that are less frequently, or never, updated, the block tends to be less worn than others. Static wear levelling refers to the activities that identify and move the inactive data out of less worn blocks and reclaim them. It therefore induces extra write amplification and is only beneficial for workloads that have a biased update frequency within the user address space. Under the random write workload, the out-of-place write, governed by a page-level FTL and the greedy garbage collection, will eventually wear out all Flash blocks evenly, so there is no need for static wear levelling.

4.2 Greedy Garbage Collection Policy

We now describe a greedy garbage collection policy and prove that it is the optimal garbage collection policy in the sense of minimizing write amplification under the random write workload.

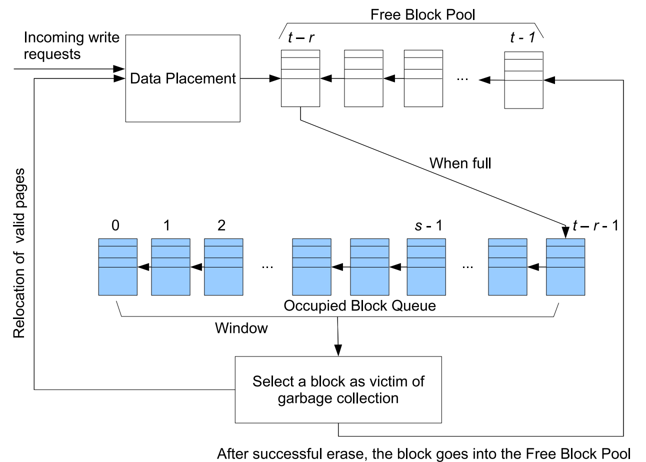

Figure 2. Block diagram of garbage collection.

Figure 2 shows a schematic diagram of greedy garbage collection. Incoming write requests and relocation write requests are both serviced by writing to free pages/blocks supervised by the data placement function. Once a free block has been filled up, it is removed from the free block pool and moves to the end of the occupied block queue. Each time garbage collection is triggered, a single occupied block is selected as victim, and all its valid data pages are read and written to another location by issuing relocation write requests. Upon completion of relocation, the victim block is erased and joins the free block pool again. Denoting the number of free blocks by r and the total number of physical

Flash blocks by t, there are $t - r$ occupied blocks that can potentially be victims of garbage collection.

Consider the following greedy garbage collection policy:

- Delay garbage collection as long as possible until the number of free blocks drops below a pre-defined threshold.
- Select the block with the lowest number of valid pages among all occupied blocks.

Now we prove that the greedy garbage collection strategy is optimal in the sense of minimizing write amplification under the random write workload with aligned page writes.

The optimality of delaying garbage collection as long as possible is straightforward. In principle, garbage collection can be triggered even if there are still many free blocks, and there are practical arguments for doing that, such as, to lengthen the duration of sporadic write workloads during which garbage collection can be turned off for performance reasons. This, however, compares unfavorably with the greedy garbage collection policy in terms of garbage collection efficiency. The more incoming write requests are serviced before garbage collection occurs, the less likely it will be that a given page in an occupied block is valid, and the fewer the valid pages in the victim block to be relocated, which ultimately leads to less write amplification. In other words, the greedy usage of free blocks through delaying garbage collection as long as possible maximizes the benefit of over-provisioning.

The greedy policy to select a block from the occupied blocks that has the lowest number of valid pages as the victim is a natural choice, and its optimality is tied to the assumption of the random write workload. We prove the optimality by contradiction. Suppose that at time t_1 block a is selected as victim according to the greedy garbage collection policy because it has the least valid pages on it, say v_a. Suppose there is an imaginary optimal garbage collection policy, other than the greedy one, that would instead select another block b as the victim, which has v_b valid pages at t_1, where $v_b > v_a$. Now suppose this imaginary garbage collection would select block a as victim at a later time t_2. The necessary (but not sufficient) condition for the imaginary optimal garbage collection policy to outperform the greedy one is that the number of pages turning from valid to invalid in block a, in the time period t_1 to t_2, should outpace that in block b. Under the random write workload with aligned page writes, every valid page has the same probability of turning invalid, and thus the condition contradicts the probability law as block a has fewer valid pages than block b at time t_1, meaning that an imaginary optimal garbage collection policy outperforming the greedy one for the random write workload does not exist.

The greedy garbage collection policy requires finding an occupied block with the least valid pages among a possibly large number of Flash blocks, incurring potentially significant computational overhead. A window-based garbage collection policy can reduce the computational overhead by restricting the search for the block with the least valid pages to a window of occupied Flash blocks [23]. Once that block has been erased, the next block in the queue enters the window. A special case of the window-based garbage collection is the FIFO (or cyclic) garbage collection policy, which sets the window size to 1, i.e. it always picks the oldest Flash block written as the victim, or in other words, it reclaims Flash blocks according to their physical address sequentially and cyclically. If the window covers all occupied blocks, it corresponds to the greedy garbage collection policy.

Remarks: Note that the greedy garbage collection policy described above is just a simple abstraction for performance analysis and should not considered as a practical algorithm. A realistic SSD might trigger garbage collection process proactively, for instance, whenever a sufficient idle time is found, instead of relying on free block threshold alone, for the benefit of maximizing burst write performance. The real-time issue of garbage collection should also be carefully considered in practice to avoid unpleasant side-effects, for example, if garbage collection is triggered too rarely or too late, but takes longer to complete, the write could suffer overly large delays freezing a process or even the SSD.

4.3 Empirical Performance Model

Given the random write workload with aligned page writes, the qualitative analysis above indicates that the fundamental cause of the slowdown of the sustained random write performance is the increased utilization. Now we develop an empirical performance model to capture the performance impact of the utilization. The greedy garbage collection policy, shown to be optimal under the random write workload, is assumed, to preclude possible artifacts arising from a suboptimal garbage collection policy.

Let n_p, denote the number of pages in a Flash block, $n_p = 64$, and p_0^*, p_1^*, ..., $p_{n_p}^*$ be the probabilities that the victim selected by the greedy garbage collection policy has 0, 1, ..., n_p valid pages, respectively. The write amplification w_a, by definition, is computed by

$$w_a = \frac{n_p}{n_p - \sum_{k=1}^{n_p} k p_k^*}, \tag{1}$$

and

$$p_0^* = 1 - p(\forall_j V^j > 0) \tag{2a}$$

$$p_k^* = p(\forall_j V^j > k-1) - p(\forall_j V^j > k) \quad \text{for } k = 1, \cdots, n_p - 1 \tag{2b}$$

$$p_{n_p}^* = p(\forall_j V^j > n_p - 1), \tag{2c}$$

in which $p(\forall V^j > k)$ is defined as the probability that the number of valid pages in *every* block of the garbage collection window is larger than k, which can be approximated by

the product of s (the window size) and the marginal probabilities $p(V^j > k)$ that the j-th block has more than k valid pages, namely,

$$p(\forall_j V^j > k) \approx \prod_{j=0}^{s-1} p(V^j > k). \quad (3)$$

Denote the probability that the j-th block has m valid pages by $p_j(m)$, then

$$p(V^j > k) = 1 - \sum_{m=0}^{k} p_j(m), \quad (4)$$

and by assuming that each page on the j-th block has the same probability p_j to be valid at the moment of garbage collection, $p_j(m)$ is computed by

$$p_j(m) = \binom{n_p}{m} p_j^m (1 - p_j)^{n_p - m}. \quad (5)$$

Two models, the fixed model and the coupon collector model, have been proposed to approximate p_j, which yielded analytical write amplification results in good agreement with simulation results, except at high utilization and for very small window sizes [23]. However, for the FIFO garbage collection, both models failed to predict the write amplification accurately, particularly in the case of high utilization.

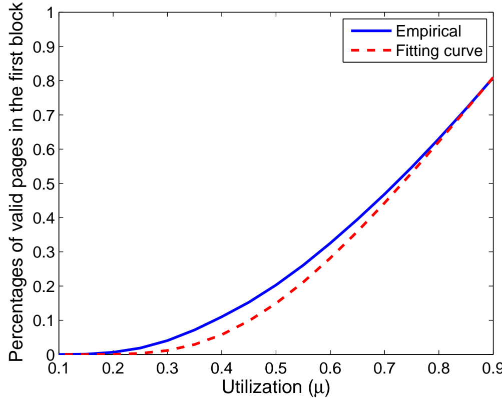

Figure 3. Empirical and fitting percentages of valid pages in the first block as a function of utilization.

To remedy this drawback, we present an empirical model to evaluate p_j. Figure 3 shows empirical percentages of valid pages of the first occupied block as a function of utilization obtained by our Flash simulator [23], and the curve can be well fitted by

$$p_0 = e^{-\alpha(\frac{1}{\mu} - 1)}, \quad (6)$$

where α is a constant of value 1.9 and μ is the utilization. Empirically we also find that p_j can be recursively evaluated by

$$p_j = \beta p_{j-1} / (1 - \frac{1}{u_p})^{n_p}, \quad (7)$$

where β is a constant of value 1.1, u_p is the number of pages of the user space, and n_p is the number of pages per Flash block.

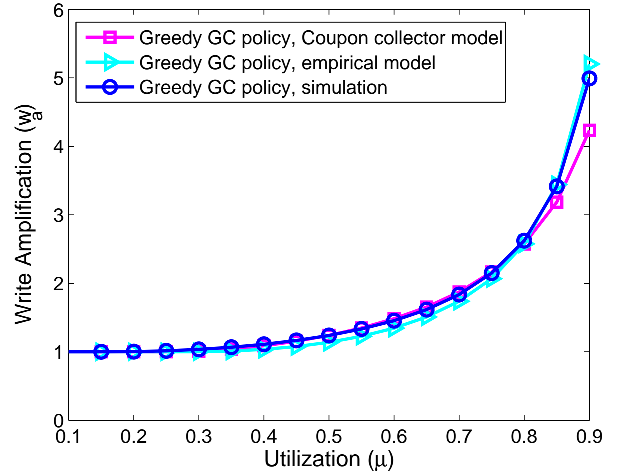

Figure 4. Write amplification as a function of utilization using the greedy GC policy.

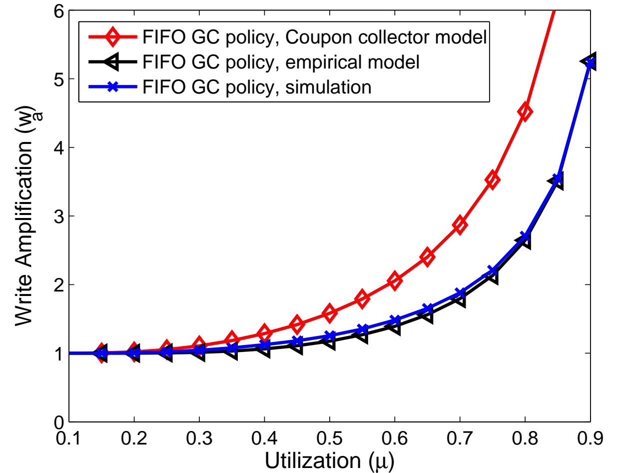

Figure 5. Write amplification as a function of utilization using the FIFO GC policy.

Figures 4 and 5 compare the write amplification under the random write workload with the greedy and the FIFO garbage collection policy, using the empirical model as well as the coupon collector model, with simulation results. Simulation parameters are given in [23]. We expect actual parameters values not to affect write amplification substantially given $t \gg r$. It can be seen that, for the greedy GC policy, both the coupon collector model and the empirical model yield fairly accurate evaluation of the write amplification for the entire utilization range, as compared with the simulation results. The coupon collector model, however, fails to yield a good approximation for the FIFO garbage collection policy, whereas the empirical model faithfully predicts the write

amplification also in this scenario. Another interesting observation is that, under the random write workload, the write amplification of the FIFO garbage collection policy is just slightly worse than that of the greedy one for the entire utilization range, suggesting that the window size does not play a significant role for the random write workload. This phenomenon can be attributed to the speciality of the random write workload.

The fundamental performance slowdown of the sustained random write performance relative to the peak random write performance is caused by the write amplification due to garbage collection. With the help of the empirical model for computing the write amplification, we now measure the slowdown factor s_f, defined as the ratio of the peak random write performance in terms of IOPS to the sustained random write performance,

$$s_f \triangleq \frac{IOPS_{PeakRandomWrite}}{IOPS_{SustainedRandomWrite}}. \tag{8}$$

After depleting free blocks, each page write translates into $w_a - 1$ page reads and w_a page writes on average to sustain continuous writes. Recall that each page read or write requires the same time, approximately 100 μs, for data transfer to/from the controller, whereas each page read takes about 25 μs and each page write takes about 200 μs. Thus the equivalent performance overhead of a page read can be cast into a fraction γ of a page write,

$$\gamma \triangleq \frac{100 + 25}{100 + 200} = \frac{5}{12}. \tag{9}$$

This is to say, each page write from the user in the sustainable mode has an overhead of $(w_a - 1)\gamma + w_a$ of equivalent page writes, therefore

$$IOPS_{SustainedRandomWrite} = \frac{12}{17w_a - 5} IOPS_{PeakRandomWrite}, \tag{10}$$

and s_f can be computed by

$$s_f = \frac{17w_a - 5}{12}. \tag{11}$$

Figure 6 shows the performance slowdown behavior of the sustained random write performance relative to the peak random write performance with increasing utilization, using simulation results on write amplification. It is shown that the performance deteriorates rather quickly (almost exponentially) as the utilization increases.

4.4 Fundamental Limit on Endurance

Write amplification is the fundamental cause not only to the performance slowdown, but also for the endurance lifetime. Endurance is a critical reliability issue in Flash SSDs.

Given a fixed amount of raw Flash memory, a Flash SSD can sustain a total number of page writes (product of the number of raw Flash pages and the nominal program-erase cycle count of Flash pages), of which only a fraction is available for user writes. To capture this fact, long-term data endurance (LDE) is defined as the total number of actual user page writes (alternatively measured as GiB) to measure an SSD's nominal endurance lifetime [45]. In general, LDE depends on the types of workload, efficiency of garbage collection and wear levelling, and utilization. In particular, under the random write workload and the optimal garbage collection policy, LDE is inversely proportional to the write amplification, which grows exponentially with utilization. It is thus crucial to minimize write amplification as much as possible in a Flash SSD in order to increase its performance and reliability.

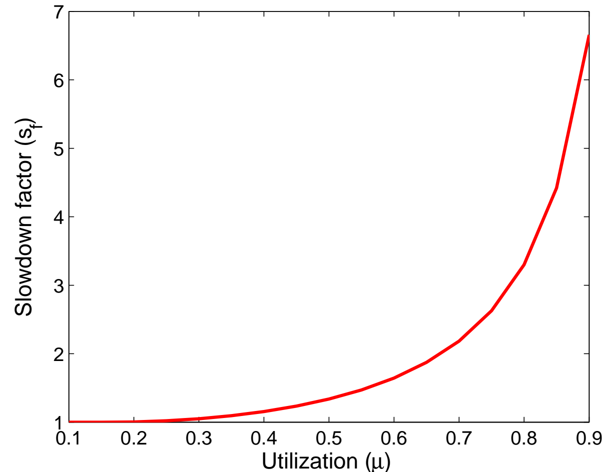

Figure 6. Slowdown factor as a function of utilization assuming the random write workload and the greedy GC policy.

5. Taming the Write Amplification

5.1 Utilization Control

The most straightforward way to improve the performance slowdown and endurance of Flash SSDs is to limit the utilization. There are two ways to do this: Explicit or implicit over-provisioning.

Explicit over-provisioning refers to the practice that only a fraction of the raw (physical) Flash memory space is exposed to the user as the logical address space, although the entire physical Flash memory space is actually used. This practice is often seen in enterprise-class Flash SSDs. In this way, the utilization is upper-bounded by a pre-specified threshold and thus the sustained random write performance and endurance are guaranteed at a certain level over the lifetime thanks to the tamed write amplification. The main disadvantage is the extra cost due to over-provisioning that causes the loss of space available to users.

Implicit over-provisioning leaves the responsibility of controlling the utilization over the lifetime to the user. This gives the user more flexibility for adjusting the utilization according to access patterns, and is popular with SSD vendors targeting personal computing applications. The rationale behind this practice is that if a workload with mostly read requests and only a light random write workload is expected, the user may use the Flash SSD more cost-effective at a higher utilization.

One may expect that, with implicit over-provisioning, the sustained random write performance would stabilize without slowdown over usage, once the utilization remains fixed. Contrary to conventional wisdom and current practice, experiments in [41] reported that even with only 10% utilization of the address space, after having written the entire address space once, there was still a substantial performance slowdown with two modern Flash SSDs when implicit over-provisioning is used.

The reason is that current file systems are designed assuming a simple abstraction of a linear block-level, update-in-place interface to the underlying storage and optimized under "unwritten contract" for HDDs [43, 46]. For various reasons, file systems maintain a table map between the meta data or data blocks of files/directories and the logical address space of the underlying storage to support file creation, modification and deletion. Because the default storage is assumed to be HDDs, which support in-place updates, current file systems do not inform the storage device when deleting files/directories.

This practice is particularly problematic when the underlying storage is Flash SSDs. Because Flash SSDs are not aware of those data blocks that have been deleted by the file systems, Flash SSDs have to handle an inflated, larger logical space than file systems actually use. A lot of data pages containing no useful data have to be relocated during garbage collection because of the lack of deletion information, leading to excessive write amplification and a severe performance slowdown.

This problem is particularly pronounced for implicit over-provisioning, as frequent file/directory deletions and creations by file systems quickly eat up the implicit over-provisioning by inflating the logical address space seen by the controller with useless data. Referring to Fig 6, the sustained write performance can be slowed down more than six times if the inflated utilization amounts to 0.9. In the worst case, the SSD may even die much earlier because of excessive write amplification.

To overcome this problem, the best way is to let file systems inform Flash SSDs by means of a trim command when certain data blocks have been deleted [41, 43, 45]. Notifying Flash SSDs of data blocks that are no longer valid saves the SSDs from having to relocate those data during garbage collection, reducing the write amplification and thus also improving the sustained random write performance and endurance.

The actual use of trim commands is virtually nonexistent so far, but is expected to appear with newer releases of operating systems in the near future. For example, the trim command has been proposed, although not yet finalized, to add block delete notifications to the ATA interface [48], and both Windows© 7 and Linux© ext4 announced to support it. Today there are only few SSD vendors that have trim support, but we expect that trim support will become universal in the future.

Remarks: It can be concluded that the current practice of some SSD vendors to pass the responsibility of controlling the utilization of the SSD device to the end user(s), namely implicit over-provisioning, can potentially cause substantial performance slowdown or even incur the risk of losing data. Explicit over-provisioning is shown to be the only effective way to tame the utilization currently (at the time of writing) before the trim command is supported by both file systems and SSD vendors, but comes at additional cost due to the loss of space available to users. One intermediate remedy for implicit over-provisioning is to create multiple partitions on a Flash SSD, and leave one partition unused to enable explicit over-provisioning.

5.2 Sequentiality Shaping

Arguably the random write workload is an unusual or even pathological workload. Most random write workloads exhibit a certain level of sequentiality, which can be approximately modelled by a random write workload with an average size larger than one single page. It is expected that sequentiality has a positive impact on the random write performance by reducing the write amplification.

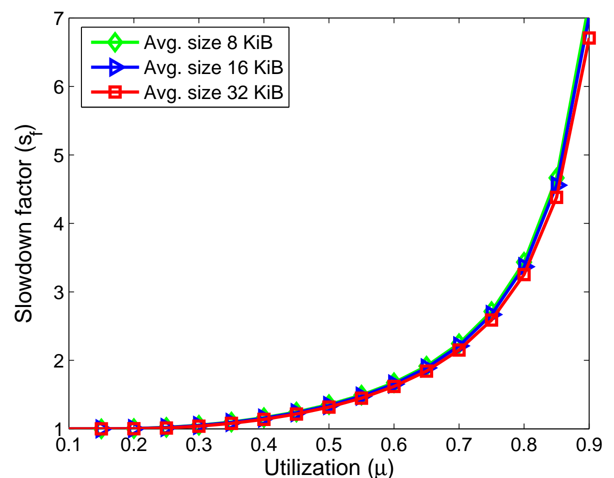

Figure 7. Slowdown factor with varying average request sizes.

One form of sequentiality shaping is to use a DRAM-based buffer cache to increase sequentiality of the under-

lying workload [29, 50]. To characterize the effect of the increased sequentiality on performance, we investigate the slowdown factors for a range of random write workloads of average sizes via simulation, shown in Figure 7. To avoid the complexity of handling unaligned pages, the workloads are assumed to be aligned on the page boundary, and with sizes of multiple pages. Compared with long sequential workloads, the sequentiality modelled in these workloads is local and sporadic. It can be seen from Figure 7 that the local and sporadic sequentiality has a rather small positive impact on reducing the slowdown factor. For the average request sizes of 8, 16, and 32 KiB, there is no significant advantage in term of performance slowdown compared with the random write workload with aligned 4 KiB write requests.

This result seems counterintuitive to the general belief that sequentiality can effectively decrease the write amplification. It reveals that there is a significant difference between long sequentiality and local sporadic sequentiality. The reason is that, under long sequential workloads, the data layout on Flash is most likely sequential, and when updating sequentially, there are large chunk of invalid pages, leading to reduced write amplification and slowdown factor; In contrast, under local sporadic sequential workload, the data layout on Flash is much less likely sequential, and then there is less possibility for the existence of large chunks of invalid pages. This suggests that workload shaping schemes that exploit local sporadic sequentiality of the workload may not be effective in terms of reducing write amplification.

Remarks: Note that the above results are obtained based on a page-level FTL that incurs no FTL garbage collection overhead. When a hybrid or block FTL is used, it is likely that increasing the level sequentiality of the workloads, e.g., by using buffer management schemes, can have a positive impact on reducing write amplification due to FTL garbage collection.

Some file systems use a large page size such as 8, 16 or even 32 KiB. It is conjectured that large page sizes would be beneficial for reducing write amplification because of the inherent sequentiality of a single access. Figure 8 shows the simulated slowdown factors as a function of utilization for page sizes of 8, 16 and 32 KiB, in which every write is assumed to be aligned on the boundary of pages, the workload is again IUP and the greedy GC is used. It is confirmed that increasing page size has a positive impact on combatting performance slowdown at high utilization — the larger the page size, the better improvement in terms of performance slowdown.

5.3 Data Placement

It is unlikely that an application writes randomly to the entire logical address space of the Flash SSD. Instead, most practical write workloads exhibit spatial skewness, namely, some data is updated more frequently, whereas other data is updated less frequently. In terms of the frequency of being updated, data can be classified as *active* (or short-lived) and *inactive* (or long-lived). The read-only data naturally belongs to inactive data.

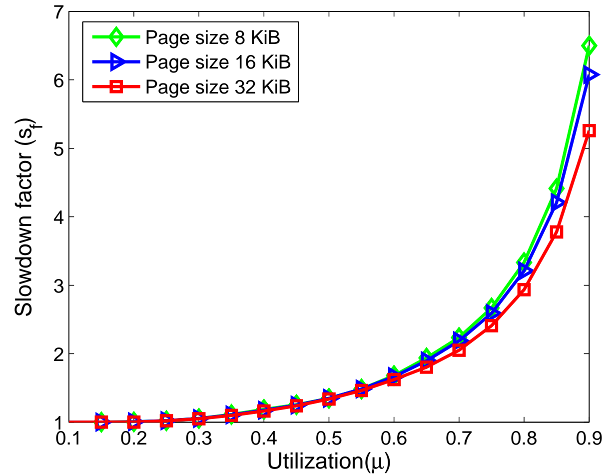

Figure 8. Slowdown factors under varying page sizes.

The issue of data placement is trivial under the random write workload, because each page has the same probability of being updated, and thus page writes can simply be packed into free Flash blocks sequentially. Care must be taken, however, when active and inactive data co-exist. If active and inactive data are not distinguished and placed into the same Flash block in a mixed way, active data tends to be updated quickly (therefore becoming invalid), whereas other data are inactive and likely remain valid for a relatively long time. The result is that the garbage collection has to relocate the inactive data. In contrast, if a block only contains active data and becomes selected as victim for garbage collection, it is most likely that all pages in this block will be invalid. Hence it is desirable to distinguish active data from inactive data and to place active data and inactive in separate blocks.

To investigate the impact of data placement, we assume a special workload with active data being updated independently and with equal probability and inactive data never being updated, i.e., read-only data, and we compare two data placement schemes: mixed and separated. The mixed data placement scheme assumes no information on data activeness, and simply packs data together into free Flash blocks without awareness of the activeness of a data page. The separated data placement scheme is assumed to have perfect knowledge of the activeness of each data page and thus places active data and inactive data separately in different blocks.

Figures 9 and 10 show write amplification of the two data placement schemes as a function of utilization for workloads with varying portions of read-only data. The write amplification for the mixed data placement is obtained via simulation, using the FIFO garbage collection policy. It can be seen that the FIFO garbage collection itself can not exploit

the existence of a large percentage of inactive data, although it performs nearly optimal for the random write workload. The write amplification for the separated data placement is computed using the empirical model under the FIFO garbage collection. It can be seen that, once active data and inactive data are separated, the write amplification, given a fixed utilization, can be significantly improved.

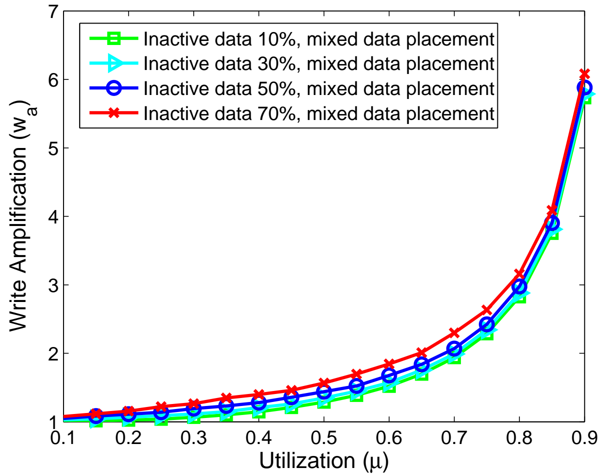

Figure 9. Write amplification as a function of utilization under the mixed data placement scheme.

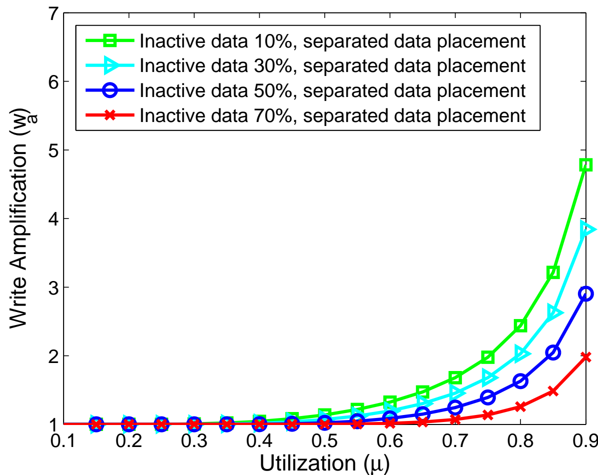

Figure 10. Write amplification as a function of utilization under the separated data placement scheme.

It is evident that the separated data placement scheme offers an advantage over the mixed one, and as the portion of read-only data increases, the improvement becomes more pronounced. This result provides quantitative/theorectical evidence that active data and inactive data should not be placed on the same Flash block, and strongly suggests that an ideal separation data placement scheme can significantly reduce write amplification when incorporated into garbage collection policy for realistic workloads with a lot inactive data.

The concept and its related techniques of separating active data from inactive data have been well studied in the log-structured file systems in the context of garbage collection and cleaning policies [44], which is a closely related research area to Flash SSDs. However there is no prior work, to our knowledge, to address the performance modelling of write amplification with respect to utilization, and to study the potential of an ideal data placement scheme.

5.4 Data Migration

Spatial skewness is found ubiquitously also in read workloads. In terms of read frequency, data can be classified as hot, i.e., being frequently read, or cold. As the price of Flash SSDs per GiB is still at least one order of magnitude higher that that of HDDs, it would be cost-effective to migrate data that is cold and inactive out of Flash SSDs and into lower-cost storage media, such as HDDs. It is expected that such migration would not degrade the overall performance because such data is rarely accessed.

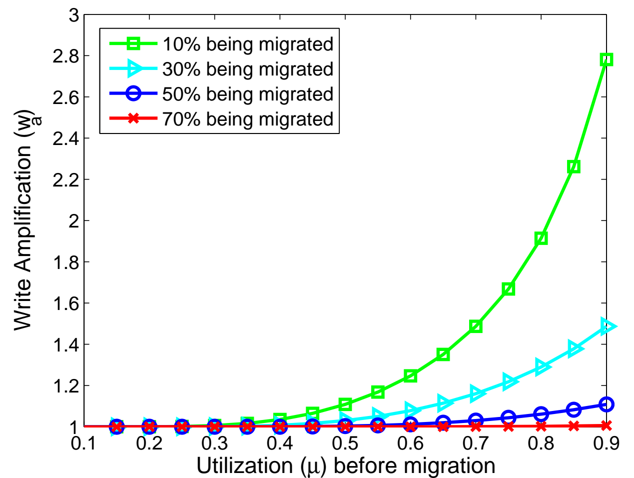

Figure 11. Write amplification as a function of utilization (original) with data migration.

We argue that, if the portion of cold and inactive data is relatively large and is perfectly identified, migrating this portion of data would actually boost the sustained random write performance of Flash SSDs by effectively reducing Flash utilization and thus reducing write amplification. Figure 11 shows the decreasing behavior of write amplification as larger portions of data are identified as cold and inactive and are migrated. The underlying reason is that the effective utilization is reduced by migrating part of the data out of Flash SSDs, leading to a lower write amplification. This observation has a profound implication: Cold and inactive data should not be stored forever on Flash memory, not

merely for cost efficiency reasons, but also for improving Flash SSD performance and endurance. Our findings provide compelling evidence and strongly suggest that the optimal way to integrate Flash into current memory and storage hierarchy should be either to use Flash as a cache layer or to use a tiered storage architecture with Flash dedicated to hot and active data only.

6. Conclusions

The poor random write performance of Flash SSDs and their performance slowdown can be caused by either design/implementation artifacts, which can be eliminated as technology matures, or by fundamental limits due to unique Flash characteristics. Identifying and understanding the fundamental limit of Flash SSDs are beneficial not only for building advanced Flash SSDs, but also for integrating Flash memory into the current memory and storage hierarchy in an optimal way.

We analyze the fundamental limit of the sustained random write performance for Flash SSDs, assuming an independently, uniformly-addressed random write workload. We prove that the greedy garbage collection policy is the optimal garbage collection policy for such a workload and that no static wear levelling is needed. Furthermore, an empirical model is developed to compute the write amplification and performance slowdown factor as a function of utilization. Potential causes of the commonly observed performance slowdown are investigated, and remedies suggested. Moreover, we demonstrate that data placement and data migration may lead to a significant improvement of the random write performance and endurance by exploiting the spatial skewness of workloads. Our results provide compelling evidence that the optimal way to integrate Flash into the current memory and storage hierarchy would be either to use Flash as a cache layer or to use a tiered storage architecture with Flash dedicated to hot and active data only.

The paper leaves several questions to the future research. First, as the performance analysis model developed in this work is limited to the greedy garbage collection policy, it would be interesting (and also challenging) to analyze the random write performance for a more realistic garbage collection policy under realistic workloads. Second, the paper highlights the benefits of data placement and data migration through a theoretical analysis, but provides no concrete solution, and we hope this work could trigger further research, although there are some preliminary attempts, on how to efficiently combine data placement/migration with garbage collection. Last but not least, recent work has shown that a smart write buffer management algorithm together with block-level FTL may perform as well as or even better than page-level mapping schemes, thus, an analysis of garbage collection for block-level FTL is desirable.

References

[1] SmartMedia^{TM} specification. SSFDC Forum, 1999.

[2] 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.

[3] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *Proceedings of the Workshop in Experimental Algorithms*, pages 208–219, 2008.

[4] D. G. Andersen, J. Franklin, and M. Kaminsky. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM Symposium on Operating Systems Principles SOSP*, 2009.

[5] 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.

[6] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *ACM SIGOPS Operating Systems Review*, 41:88–93, 2007.

[7] T. Bisson and S. A. Brandt. Reducing hybrid disk write latency with flash-backed I/O requests. In *Proceedings of the 15th IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2007.

[8] L. Bouganim, B. Jonsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2009.

[9] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 217–228, Feb. 2009.

[10] L.-P. Chang. On efficient wear leveling for large-scale flash-meory storage systems. In *Proceedings of 22nd ACM Symposium on Applied Computing*, May 2007.

[11] 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.

[12] 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.

[13] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proceedings of the 11th International Symposium on Low Power Electronics and Design (ISLPED)*, 2006.

[14] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the ACM SIGMETRICS/Performance*, 2009.

[15] S. Chen. FlashLogging: Exploting flash devices for synchronous logging performance. In *Proceedings of the ACM*

SIGMOD, 2009.

[16] B. Debnath, S. Subramanya, D. Du, and D. J. Lilja. Large block CLOCK (LB-CLOCK): A write caching algorithm for solid state disks. In *Proceedings of the 17th IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009.

[17] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSD) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th IEEE International Symposium on Computer Architecture (ISCA)*, 2009.

[18] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, June 2005.

[19] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the Usenix Annual Technical Conference*, 2005.

[20] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proceedings of the Third International Workshop on Data Management and New Hardware (DaMoN)*, 2007.

[21] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, Feb. 2009.

[22] J.-W. Hsieh, T.-W. Kuo, P.-L. Wu, and Y.-C. Huang. Energy-efficient and performance-enhanced disks using flash-memory cache. In *Proceedings of the 11th International Symposium on Low Poweer Electronics and Design (ISLPED)*, 2007.

[23] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid-state drives. In *Proceedings of the ACM SysStor: The Israeli Experimental Systems Conference*, May 2009.

[24] H. Jo, J. K. S. Park, J. Kim, and J. Lee. FAB: Flash-aware buffer management policy for portable media players. *IEEE Trans. Consumer Electronics*, 22(2), 2006.

[25] J. Kang, H. Jo, J. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 161–170, 2006.

[26] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the Usenix Annual Technical Conference*, Jan. 1995.

[27] T. Kgil and T. Mudge. Flashcache: A NAND flash memory file cache for low power web servers. In *Proceedings of the CASES'06*, 2006.

[28] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *IEEE Proceedings of the 35th IEEE International Symposium on Computer Architecture (ISCA)*, 2008.

[29] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies*, pages 1–14, Feb. 2008.

[30] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compact flash systems. *IEEE Trans. Consumer Electronics*, 48:366–375, May 2002.

[31] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. In *Proceedings of the VLDB*, 2008.

[32] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A log buffer based flash translation layer using fully associative sector translation. *ACM Trans. Embedded Computing Systems*, 6(3), 2007.

[33] S. Lee, D. Shin, Y. Kim, and J. Kim. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. In *Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED)*, Feb. 2008.

[34] S.-W. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of the ACM SIGMOD*, 2007.

[35] S.-W. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In *Proceedings of the ACM SIGMOD*, 2009.

[36] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the ACM SIGMOD*, 2008.

[37] A. Leventhal. Flash storage memory. *Communications of the ACM*, 52:47–51, 2008.

[38] C. Manning. YAFFS: Yet another flash file system, 2004.

[39] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proceedings of the ACM EuroSys*, 2009.

[40] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the CASES'06*, 2006.

[41] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *Proceedings of the First Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH), held in conjunction with ASPLOS*, 2009.

[42] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the USENIX Operating Systems Design and Implementation OSDI*, 2008.

[43] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *Proceedings of the USENIX Annual Technical Conference*, June 2009.

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

[45] SanDisk. Long-term data endurance (LDE) for client SSD, Oct. 2008. white paper, No 80-11-01654.

[46] S. W. Schlosser and G. R. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, pages 87–100, 2004.

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

[48] F. Shu. Notification of deleted data proposal for ATA8-ACS2, 2007.

[49] Silicon Systems Whitepaper. Preventing storage system wear-out, 2008.

[50] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *Proceedings of the 5th International Workshop on Data Management on New Hardware*, 2009.

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

[52] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb. 1994.