Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes

Gala Yadgar, Eitan Yaakobi, and Assaf Schuster, ***Technion—Israel Institute of Technology***

https://www.usenix.org/conference/fast15/technical-sessions/presentation/yadgar

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX

Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes

Gala Yadgar, Eitan Yaakobi, and Assaf Schuster
Computer Science Department, Technion
{*gala,yaakobi,assaf*}*@cs.technion.ac.il*

Abstract

NAND flash, used in modern SSDs, is a *write-once* medium, where each memory cell must be erased prior to writing. The lifetime of an SSD is limited by the number of erasures allowed on each cell. Thus, minimizing erasures is a key objective in SSD design.

A promising approach to eliminate erasures and extend SSD lifetime is to use *write-once memory (WOM)* codes, designed to accommodate additional writes on write-once media. However, these codes inflate the physically stored data by at least 29%, and require an extra read operation before each additional write. This reduces the available capacity and I/O performance of the storage device, so far preventing the adoption of these codes in SSD design.

We present *Reusable SSD*, in which invalid pages are reused for additional writes, without modifying the drive's exported storage capacity or page size. Only data written as a second write is inflated, and the required additional storage is provided by the SSD's inherent overprovisioning space. By prefetching invalid data and parallelizing second writes between planes, our design achieves latency equivalent to a regular write. We reduce the number of erasures by 33% in most cases, resulting in a 15% lifetime extension and an overall reduction of up to 35% in I/O response time, on a wide range of synthetic and production workloads and flash chip architectures.

1 Introduction

The use of flash based solid state drives (SSD) has increased in recent years, thanks to their short read and write latencies and increasing throughput. However, once flash cells are written upon, they must be erased before they can be rewritten. These comparatively slow erasures, along with the additional overheads they incur, significantly slow down pending read and write operations. In addition, flash cells have a limited *lifetime*, measured as the number of erasures a block can endure before its reliability deteriorates below an acceptable level [1, 12].

Erasures are the major contributors to cell wear [24]. Thus, much effort has been invested in attempts to reduce them and extend SSD lifetime. Suggested methods include minimizing write traffic [16, 18, 29, 38, 43, 46, 53] and distributing erase costs evenly across the drive's blocks [1, 20, 27, 28]. While most of these methods improve performance due to the reduction in erasures, others extend device lifetime at the cost of degrading performance [11, 24, 30]. Another approach is to improve current error correction methods in order to compensate for the decreasing reliability of blocks late in their lifetime [7, 44, 60].

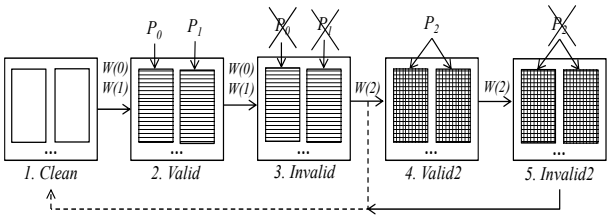


Figure 1: A simplified depiction of our second write approach. Each block holds one page, and begins in a clean state (1). Logical pages P_0 and P_1 are written by the application and stored on the first two blocks of the drive (2). When they are written again their copies are invalidated (3) and written elsewhere (not shown). Normally, the blocks would now be erased and returned to the clean state. Instead, in our design, they are *reused* to write logical page P_2 as a *second write* (4). When page P_2 is written again by the application, its copy is invalidated (5) and the blocks are erased, returning to the clean state.

A promising technique for reducing block erasures is to use write-once memory (WOM) codes. WOM codes alter the logical data before it is physically written, thus allowing the reuse of cells for multiple writes. They ensure that, on every consecutive write, zeroes may be overwritten with ones, but not vice versa. WOM codes were originally proposed for write-once storage media such as punch cards and optical disks [47]. However, they can be applied to flash memories, which impose similar constraints: the bit value of each cell can only increase, not decrease, unless the entire block is erased[1]. Indeed, several recent studies proposed the use of WOM codes to reduce SSD block erasures [4, 14, 21, 23, 35, 42, 57].

Unfortunately, the additional writes come at a price. The old data must be read before the new data can be encoded. More importantly, WOM codes 'inflate' the data: the physical capacity required for storing the encoded data is larger than the original, logical, data by at least 29% —

[1] We adopt the conventions of coding literature, and refer to the initial, low voltage state of flash cells as zero.

a theoretical lower bound [17, 47]. Furthermore, WOM code design involves three conflicting objectives: minimizing physical capacity, minimizing encoding complexity, and minimizing the probability of encoding failure. Any two objectives can be optimized at the cost of compromising the third.

Existing studies have focused on minimizing the physical capacity overhead and the probability of encoding failure, greatly increasing complexity. The resulting designs, in which additional writes are performed on the same 'used' page, incur impractically high overheads. Thus, the industry has been unable to exploit recent theoretical advances effectively.

Our goal is to bridge the gap between theory and practice. To achieve a practical, applicable design, we are willing to tolerate a certain probability of (albeit rare) encoding failure and mitigate its penalties with the negligible overhead of an additional calculation.

We present *Reusable SSD* — a design that uses WOM codes to perform second writes on flash, thus reducing erasures and extending SSD lifetime. This is, to the best of our knowledge, the first design that addresses all the practical constraints of WOM codes. A simplified depiction of the design of Reusable SSD appears in Figure 1.

In order to preserve the SSD's logical capacity, we perform first writes with unmodified logical data, with no additional overheads, and utilize existing spare space within the SSD to perform 'inflated' second writes. For efficient storage utilization, we use second writes only for "hot" data that is invalidated quickly.

In order to preserve the SSD's I/O performance, we use WOM codes with encoding complexity equivalent to that of error correction codes used by current SSDs. Second writes are written on two used physical pages on different blocks, so that they are read and written in parallel, avoiding additional latency. We prefetch the old, invalid, data in advance to avoid additional delays.

We evaluate Reusable SSD using the SSD [1] extension of the DiskSim simulator [5], and a well-studied set of workloads [40, 41]. Second writes in our design are indeed shown to be "free": they reduce the number of erasures by an almost steady 33%, resulting in a 15% lifetime extension. By eliminating so many erasures while preserving the read and write latency of individual operations, our design also notably reduces I/O response time: up to 15% in enterprise architectures and up to 35% in consumer class SSD architectures. Furthermore, our design is orthogonal to most existing techniques for extending SSD lifetime. These techniques can be combined with Reusable SSD to provide additional lifetime extension.

The rest of this paper is organized as follows. Section 2 contains the preliminaries for our design. We present our implementation of second writes in Section 3, and give an overview of our design in Section 4. The details are described in Section 5, with our experimental setup and evaluation in Section 6. We survey related work in Section 7, and conclude in Section 8.

2 Preliminaries

2.1 Use of NAND Flash for SSD

A flash memory chip is built from floating-gate cells that can be *programmed* to store a single bit, two bits, and three bits in SLC, MLC and TLC flash, respectively. Cells are organized in blocks, which are the unit of erasure. Blocks are further divided into pages, which are the read and program units. Each block typically contains 64-384 pages, ranging in size from 2KB to 16KB [12, 14]. Within the chip, blocks are divided into two or more *planes*, which are managed and accessed independently. Planes within a chip can operate concurrently, performing independent operations such as read, program, and erase, possibly with some minor restrictions [14, 19, 49, 55].

Each page is augmented with a *page spare area*, used mainly for storing redundancy bytes of *error correction codes* (ECC) [12, 14]. The size of the spare area ranges between 5% and 12.5% of the page's logical size [19, 49, 55, 60]. The larger sizes are more common in recent architectures, because scaling in technology degrades the cell's reliability [12, 45]. Furthermore, the *bit error rate (BER)* increases as a function of the block's lifetime, requiring stronger ECC as the block grows older [8, 12, 37].

Write requests cannot update the data in the same place it is stored, because the pages must first be erased. Thus, writes are performed out-of-place: the previous data location is marked as invalid, and the data is written again on a clean page. To accommodate out-of-place writes, some physical storage capacity is not included in the drive's exported logical capacity. Thus, the drive's *overprovisioning* is defined as $\frac{T-U}{U}$, where T and U represent the number of physical and logical blocks, respectively [12]. Typical values of overprovisioning are 7% and 28% for consumer and enterprise class SSDs, respectively [52]. The *Flash Translation Layer (FTL)* is responsible for mapping logical addresses to physical pages.

Whenever the number of clean blocks drops below a certain threshold, the *garbage collection* process is invoked. Garbage collection is typically performed greedily, picking the block with the minimum *valid count* – number of valid pages, as the victim for *cleaning*. The valid pages are *moved* – read and copied to another available block, and then the block is erased. The addition of internal writes incurred by garbage collection is referred to as *write amplification* [12]. It delays the cleaning process, and requires, eventually, additional erasures. Write amplification can be reduced by increasing overprovisioning, sacrificing logical capacity for performance and block lifetime [12, 43, 52].

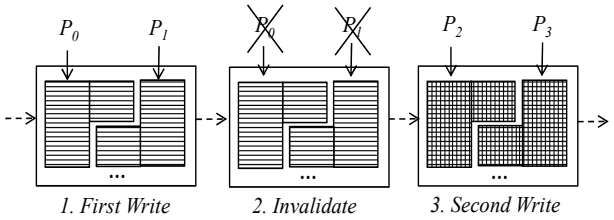


Figure 2: Naive implementation of second writes with the code in Table 1. Every write, first or second, must program and occupy storage capacity equivalent to 150% of the logical page size.

Data on the drive is usually not updated uniformly. Thus, some blocks may reach their lifetime limit, rendering the drive inoperable, while many blocks are still 'young'. Several techniques have been proposed for *wear leveling* – distributing erasures uniformly across the drive's blocks [1, 27].

2.2 Write-Once Memory Codes

Write-once memory (WOM) codes were first introduced in 1982 by Rivest and Shamir, for recording information multiple times on a write-once storage medium [47]. They give a simple WOM code example, presented in Table 1. This code enables the recording of two bits of information in three cells twice, ensuring that in both writes the cells change their value only from 0 to 1. For example, if the first message to be stored is 11, then 001 is written, programming only the last cell. If the second message is 01, then 101 is written, programming the first cell as well. Note that without special encoding, 11 cannot be overwritten by 01 without prior erasure. If the first and second messages are identical, then the cells do not change their value between the first and second writes. Thus, before performing a second write, the cell values must be *read* in order to determine the correct encoding.

Data bits	1st write	2nd write
00	000	111
10	100	011
01	010	101
11	001	110

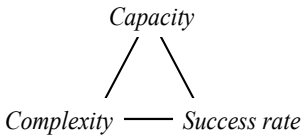
Table 1: WOM Code Example

In the example from Table 1, a total of four bits of information are written into three cells: two in each write. Note that on a write-once medium, a basic scheme, without encoding, would require a total of four cells, one for every data bit written. In general, the main goal in the design of WOM codes is to maximize the total number of bits that can be written to memory for a given number of cells and number of writes. The number of bits written in each write does not have to be the same.

A WOM code design is a called a *construction*. It specifies, for a given number of cells, the number of achievable writes, the amount of information that can be written in each write, and how each successive write is encoded. Numerous methods have been suggested for improving WOM code constructions [4, 6, 21, 25, 47, 51, 56].

To see how WOM codes can be used to reduce erasures, consider a naive application of the WOM code in Table 1 to SSD. Every page of data would be encoded by the SSD controller into 1.5 physical pages according to the WOM code construction from Table 1. Thus, each page could be written by a first write, invalidated, and written by a second write before being erased, as depicted in Figure 2. Such an application has two major drawbacks: (1) Although additional writes can be performed before erasure, at any given moment the SSD can utilize only 2/3 of its available storage capacity. (2) *Every* I/O operation must access physical bits equivalent to 50% more than its logical size, slowing down read and write response times.

Figure 3: WOM code design space trades off storage capacity, encoding complexity (efficiency) and the rate of successful encoding.



Moreover, to accommodate such an application, the SSD manufacturer would have to modify its unit of internal operations to be larger than the logical page size. Alternatively, if unmodified hardware is used, each I/O operation would have to access two physical pages, increasing its response time overhead to 100%.

The limitations of practical WOM codes complicate things even further. WOM code constructions that achieve a capacity overhead close to the theoretical lower bound ("capacity achieving") entail encoding and decoding complexities that are far from practical [51, 56]. Alternatively, more efficient constructions achieve similar capacity overhead but do not necessarily succeed in successive writes for all data combinations [6]. In other words, each such code is characterized by a small (nonnegligible) probability of failure in writing.

Figure 3 depicts the inherent tradeoff of WOM code design space. Of the three objectives: capacity, complexity, and high encoding success rate, any two can be optimized at the cost of compromising the third.

3 Implementing Second Writes

Our design is based on choosing WOM code constructions suitable for real systems. We narrow our choice of WOM code by means of two initial requirements:

1. First writes must not be modified. Their encoding and data size must remain unchanged.
2. The complexity of the chosen code must not exceed that of commonly used error correction codes.

The first requirement ensures that the latency and storage utilization of most of the I/O operations performed on the SSD will remain unaffected. The second requirement enables us to parallelize or even combine WOM and ECC encoding and decoding within the SSD controller, without incurring additional delays [25].

Thus, we limit our choice to codes that satisfy the above constraints and vary in the tradeoff between storage capac-

	a	b	c	d	e	f
Req. storage	200%	206%	208%	210%	212%	214%
Success rate	0	5%	58%	95%	99%	100%

Table 2: Sample WOM codes and their characteristics. Success rates were verified in simulations on random data, as described in [6], assuming a 4KB page size. The required storage is relative to the logical page size.

ity and success rate. One such example are *polar WOM codes* [6], based on a family of error-correcting codes recently proposed by Arikan [2]. Their encoding and decoding complexities are the same as those of LDPC error correction codes [3, 6, 50]. Polar WOM codes can be constructed for all achievable capacity overheads, but with nonnegligible failure probability [6]. Table 2 summarizes several known instances which match our requirements.

The tradeoff between storage efficiency and success rate is evident. While choosing a code that always succeeds (Table 2(f)) is appealing, it requires programming three physical pages for writing a single logical page, which, on most flash architectures, cannot be done concurrently. However, a code that requires exactly two physical pages for a second write (Table 2(a)) always fails in practice.

We compromise these two conflicting objectives by utilizing the page spare area. Recall that the spare area is mainly used for error correction. However, since the bit error rates increase with block lifetime, weaker ECC can sometimes be used, utilizing only a portion of the page spare area and improving encoding performance [7, 28, 33, 44, 48, 58, 60]. We take advantage of the page spare area and divide it into two sections, as depicted in Figure 4. In the first, smaller section, we store the ECC of the first write. In the second, larger section, we store the ECC of the second write *combined* with some of the encoded data of the second write. A way to produce this combined output has been suggested in [25].

By limiting the size of the ECC of the first write, we limit its strength. Consequently, when the blocks' BER increases beyond the error-correcting capabilities of the new ECC, we must disable second writes, and the SSD operates normally with first writes only. Bit errors are continuously monitored during both reading and writing, to identify bad pages and blocks [27]. The same information can be used to determine the time for disabling second writes on each block. Another consequence of our choice of code is that WOM computations will fail with a small probability. When that happens, we simply retry the encoding; if that fails, we write the logical data as a first write. We explain this process in detail in Section 5.6, and evaluate its effect on performance in Section 6.6.

In our implementation, we use the code from Table 2(d), which requires 210% storage capacity and succeeds with a probability of 95%. We assume each page is augmented with a spare area 9% of its size [60], and allocate 2.5% for storing the ECC of the first write (Figure 4(a)). The remaining 6.5% stores 5% of the WOM code output combined with the ECC equivalent to an ECC of 4% of the page size (Figure 4(b)). Altogether, the two physical pages along with their spare areas provide a total capacity equivalent to 210% of the logical page size. An ECC of 2.5% of the page size is sufficient for roughly the first 30% of the block's lifetime [7, 33, 48], after which we disable second writes.

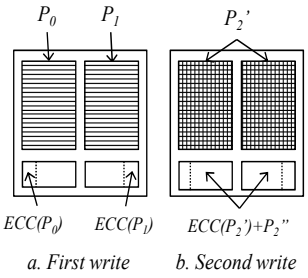


Figure 4: Use of page spare area for second writes. A small section is used for the ECC of the first write (a). The remaining area is used for the combined WOM code and ECC of the second write (b).

The utilization of the spare area can change in future implementations, in order to trade off storage capacity and success rate, according to the size of the page spare area and the available codes. According to current manufacturing trends, BERs increase, requiring stronger ECCs, and, respectively, larger page spare area. However, the size of the spare area is set to accommodate error correction for the highest expected BER, observed as flash cells reach their lifetime limit. The Retention Aware FTL [33] combines two types of ECC, using the weaker code to improve write performance when lower BERs are expected. The same method can be used to utilize the redundant spare area for second writes.

WOM codes require that there be enough ($\geq 50\%$) zero bits on the physical page in order to apply a second write. Thus, we ensure that no more than half the cells are programmed in the first write. If a first write data page has too many one bits, we program its complement on the physical page, and use one extra bit to flag this modification. The overhead of this process is negligible [10]. The application of WOM encoding to SLC flash, where each cell represents one bit, is straightforward. In MLC and TLC flash, the cell voltage levels are mapped to four and eight possible 2-bit and 3-bit values, respectively. WOM encoding ensures that the cell level can only increase in the second write, assuming the number of one bits in each level is greater than or equal to the number of ones in all lower levels. Due to inter-cell interference, the failure probability may be higher with MLC and TLC [26].

Expected benefit. To estimate the expected reduction in the number of erasures, we perform the following best case analysis. We refer to an SSD that performs only first writes as a *standard SSD*. Assume that each block contains N pages, and that there are M page write requests. The expected number of erasures in a standard SSD is $E = \frac{M}{N}$. In Reusable SSD, $N + \frac{N}{2}$ pages can be written on each block before it is erased. Thus, the expected number

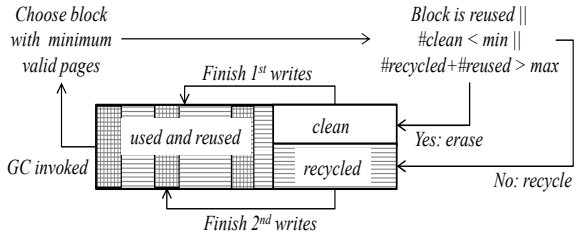


Figure 5: Garbage collection and block lifecycle

of erasures is $E' = \frac{M}{N+N/2} = \frac{2}{3}E$, a reduction of 33% compared to a standard SSD.

To calculate the effect on the SSD's lifetime, consider a standard SSD that can accommodate W page writes. 30% of them ($0.3W$) are written in the first 30% of the SSD's lifetime. With second writes, 50% more pages can be written ($0.15W$), resulting in a total of $1.15W$, equivalent to an increase of 15% in the SSD's lifetime.

4 Reusable SSD: Overview

In our design, blocks transition between four states, as depicted in Figure 5. In the initial, *clean* state, a block has never been written to, or has been erased and not written to since. A block moves to the *used* state after all of its pages have been written by a first write. When a used block is chosen as victim by the garbage collection process, it is usually recycled, moving to the *recycled* state, which means its invalid pages can be used for second writes. When all of the invalid pages on a recycled block have been written, it moves to the *reused* state. When a reused block is chosen by the garbage collection process, it is erased, moving back to the clean state.

Note that a used block can, alternatively, be erased, in which case it skips the recycled and reused states, and moves directly to the clean state. The garbage collection process determines, according to the number of blocks in each state, whether to erase or recycle the victim block. Figure 5 provides a high level depiction of this process, explained in detail in Section 5.4.

Figures 5 and 6 show how blocks are organized within the two planes in a flash chip. We divide the physical blocks in each plane into three logical partitions: one for clean blocks, one for recycled ones, and one for used and reused blocks. One clean block and one recycled block in each plane are designated as $CleanActive$ and $RecycledActive$, respectively. First writes of pages that are mapped to a specific flash chip are performed, independently, on any of the two $CleanActive$ blocks in that chip. Second writes are performed in parallel on both $RecycledActive$ blocks in the chip.

A logical page is written as a second write if (1) recycled blocks are available for second writes in both planes, and (2) the data written has been classified as hot. Pages written as first writes are divided between planes to balance the number of valid pages between them. Figure 6

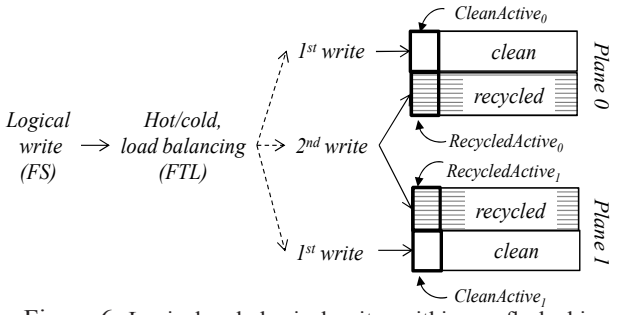


Figure 6: Logical and physical writes within one flash chip

provides a high level description of this process. A detailed description of our design is given in the next section.

5 Design Details

5.1 Page Allocation

Within an SSD, logical data is striped between several chips. The *page allocation scheme* determines, within each flash chip, to which plane to direct a logical write request. The target plane need not be the one on which the previous copy of the page was written. The standard scheme, which we modify for second writes, balances the number of clean pages in each plane [1]. Thus, a write request is directed to the plane that currently has fewer clean pages than the other.

We adapt the standard scheme to second writes as follows. When a page is classified as hot, it is written in parallel to a pair of $RecycledActive$ blocks, one in each plane, as depicted in Figure 6. To minimize the size of additional metadata, we require that a second write be performed on a pair of pages with identical offset within their blocks. Thus, we maintain an offset counter, advanced after each second write, that points to the minimal page offset that corresponds to invalid data in both $RecycledActive$ blocks. The two pages are written in parallel, utilizing the specific architecture's set of parallel or multiplane commands.

The modification to the page allocation scheme is minimal. First writes are divided between planes as before. Read requests of pages written in first writes are served as before. Read requests of pages written in second writes are served in parallel, using the respective parallel read command.

Our requirement that second write pages have identical offset affects performance only slightly. Although invalid pages may be dispersed differently in each $RecycledActive$ block, this limitation is negligible in practice. Most blocks are recycled with a very small valid count (up to 7% of the block size in our experiments), so most invalid pages can be easily reused.

5.2 Page Mapping

Our design is based on a page mapping FTL, which maintains a full map of logical pages to physical ones in the *page map* table. Since every logical page may be mapped

to two physical pages, the page map in a naive implementation would have to double in size. However, the size of the page map is typically already too large to fit entirely in memory. Thus, we wish to limit the size of the additional mapping information required for second writes.

To do so, we require that in a second write, the logical page be written on two physical pages with identical offset within the two $RecycledActive$ blocks. We maintain a separate mapping of blocks and their pairs, in a table called the *block map*, while the page map remains unmodified. Each block map entry corresponds to a block in $plane0$, and points to the pair of this block in $plane1$ on the same chip. Entries corresponding to clean and used blocks are null.

For a page written in a first write, the page map points to the physical location of this page. For a page written in a second write, the map points to the physical location of the first half of this page, in $plane0$. Thus, if the page map points to a page in $plane0$ and the corresponding block map entry is non-null, the page is stored on two physical pages, whose addresses we now know.

For a rough estimate of the block map size, assume that 2 byte block addresses are used — enough to address 64K blocks. The block map maintains entries only for blocks in $plane0$, so for a drive with 64K blocks we would need 64KB for the block map. Such a drive corresponds to a logical capacity of 16GB to 128GB, with blocks of size 256KB [55] to 2MB [19], respectively. 64KB is very small compared to the size of page mapping FTLs, and to the available RAM of modern SSDs [12]. Thus, the overhead required for mapping of second writes is negligible. The block map can be further compacted, if, instead of storing full block addresses, it would store only the relative block address within $plane1$.

The state-of-the-art page mapping FTLs, such as DFTL [15], add a level of indirection to selectively cache the hottest portion of the page map. Within such FTLs, the block map necessary for second writes can be used in a similar manner. However, due to its small size, we can reasonably assume that the block map will be stored entirely in memory, without incurring any additional lookup and update overhead.

Hybrid or block mapping FTLs typically use page-mapped *log blocks* on which data is initially written, before it becomes cold and is transferred for long term storage on *data blocks* [9]. These FTLs can be modified as described above, to apply second writes to pairs of log blocks. Although data blocks can also be paired for second writes, it may be advisable to restrict second writes only to the hot data in the log.

5.3 Prefetching Invalid Pages

Recall that a WOM encoding requires the invalidated data currently present on the two physical pages. Thus, one page must be read from each $RecycledActive$ block in a chip, before a second write can be performed. In our design, second writes are always directed to the next pair of invalid pages available on the current pair of $RecycledActive$ blocks. Thus, these pages can be *prefetched* — read and stored in the SSD's RAM, as soon as the previous second write completes.

One page must be prefetched for each plane. Thus, for a typical architecture of 8KB pages, 1MB of RAM can accommodate prefetching for 128 planes, equivalent to 64 flash chips. In the best case, prefetching can completely eliminate the read overhead of second writes. In the worst case, however, it may not complete before the write request arrives, or, even worse, delay application reads or other writes. Our experiments, described in Section 6.4, show that the latter is rare in practice, and that prefetching significantly reduces the overall I/O response time.

5.4 Garbage Collection and Recycling

We modify the standard garbage collection process to handle block recycles. Recall that greedy garbage collection always picks the block with the minimum valid count as victim for cleaning and erasure. Ideally, using second writes, every used block would first be recycled and reused before it is erased. However, our goal of preserving the exported storage capacity and performance of the SSD imposes two restrictions on recycling.

Minimum number of clean blocks. When a victim block is cleaned before erasure, its valid pages are *moved*: they are invalidated and copied to the active block. We require that valid pages move to $CleanActive$, and not to $RecycledActive$, for two reasons. First, to avoid dependency in the cleaning process, so that cleaning in both planes can carry on concurrently, and second, so that remaining valid pages that are likely cold will be stored efficiently by first writes. Thus, at least two clean blocks must be available in each plane for efficient garbage collection.

Maximum number of reused and recycled blocks. To preserve the drive's exported logical size, we utilize its overprovisioned space for second writes as follows. Consider a drive with T physical blocks and U logical blocks, resulting in an overprovisioning ratio of $\frac{T-U}{U}$. Then physical pages with capacity equivalent to $R = T - U$ blocks are either clean, or hold invalid data. For second writes, we require that the number of blocks in the recycled or reused states not exceed $2R$. Since second writes occupy twice the capacity of first writes, this number of blocks can store a logical capacity equivalent to R. Thus, the drive's physical capacity is divided between $T - 2R$ blocks holding first writes, and $2R$ blocks holding data of size R in second writes, with a total logical capacity of $T - 2R + R$, equivalent to the original logical capacity, U.

Garbage collection is invoked when the number of available clean blocks reaches a given $threshold$. We

modify the standard greedy garbage collector to count recycled blocks towards that threshold. When the threshold is reached in a plane, the block with the minimum number of valid pages in this plane is chosen as victim. The block is erased if (1) it is reused, (2) there are fewer than 2 clean blocks in the plane, or (3) the total of reused and recycled blocks is greater than $2R$. Otherwise, the block is recycled (see Figure 5).

While wear leveling is not an explicit objective of our design, blocks effectively alternate between the 'cold partition' of first writes, and the 'hot partition' of second writes. Further wear leveling optimizations, such as retirement, migrations [1] and fine grained partitioning [54], are orthogonal to our design, and can be applied at the time of block erasure and allocation.

Cleaning reused blocks. Special consideration must be given to second write pages that are still valid when a reused block is recycled. Usually, each plane is an independent *block allocation pool*, meaning garbage collection invocation, operation, and limits apply separately to each plane. This allows page movement during cleaning to be performed by *copyback*, transferring data between blocks in the same plane, avoiding inter-plane bus overheads.

Each reused block chosen as victim in one plane has a pair in the second plane of that chip. However, since each block may also have valid pages of first writes, a block may be chosen as victim while its pair does not have the minimum valid count in its plane. Thus, we clean only the victim block, as follows. All valid pages of first writes are moved as usual. Valid pages of second writes are read from both blocks, and must be transferred all the way to the controller for WOM decoding[2]. Then they are written as first writes in the plane on which garbage collection was invoked. The overhead of this extra transfer and decoding is usually small, since most reused blocks are cleaned with a low valid count (usually below 7% of the block size).

5.5 Separating Hot and Cold Data

The advantages of separating hot and cold data have been evaluated in several studies [9, 13, 20, 27, 38, 42, 54]. In our design, this separation is also motivated by the need to maintain the drive's original capacity and performance. The largest benefit from second writes is achieved if they are used to write hot data, as we explain below.

When a reused block is cleaned before erasure, all remaining valid pages must be copied elsewhere. Second write pages that are moved to the active block are not "free," in the sense that they end up generating first writes. If second writes are used only for hot data, we can expect it to be invalidated by the time the block is chosen for cleaning.

[2]Recent SSDs require similar transfer for valid pages of first writes, for recalculation of the ECC due to accumulated bit errors.

In addition, in order to maximize the potential of second writes, we wish to avoid as much as possible cases in which used blocks are erased without being reused. This may happen if too many reused blocks have a high valid page count, and the number of reused and recycled blocks reaches $2R$. Then, the garbage collector must choose used blocks as victims and erase them.

The use of a specific hot/cold data classification scheme is orthogonal to the design of Reusable SSD. As a proof of concept, we identify hot/cold data according to the size of its file system I/O request. It has been suggested [9, 20] that large request sizes indicate cold data. We classify a logical page as cold if its original request size is 64KB or more. We also assume, as in previous work [20, 42], that pages that are still valid on a block chosen by the garbage collector are cold. Thus, pages moved from a block before it is erased are also classified as cold. Cold data is written as first writes, and hot data as second writes, if recycled blocks are available (see Figure 6).

5.6 Handling Second Write Failures

Our design uses WOM codes that fail with a nonnegligible probability (the success rate is $P = 95\%$ in our implementation). A failed encoding means that the output contains 0 bits in places corresponding to cells in the invalidated pages that have already been programmed to 1.

The simplest approach to handling such failures is to simply abort the second write, and write the data on a clean block as a first write. The first write requires additional latency for computing the ECC, typically 8us [60], but is guaranteed to succeed. Within our design, choosing this approach would imply that 5% of the hot pages destined for second writes end up occupying pages in cold blocks.

A different approach is to handle the problematic bits in the same manner as bit errors in standard first writes. The output is programmed on the physical pages as is, and the ECC 'fixes' the erroneous bits. However, recall that we already 'sacrificed' some ECC strength for implementing second writes, and the number of erroneous bits may exceed the remaining error-correction capability.

Our approach is to *retry* the encoding, i.e., recompute the WOM code output. In the general case, this can be done by encoding the logical data for writing on an alternative pair of invalid pages. The two attempts are independent in terms of success probability, because they are applied to different data combinations. Thus, the probability of success in the first encoding *or* the retry is $P' = 1 - (1 - P)^2$, or 99.75% in our case. This value is sufficient for all practical purposes, as supported by our evaluation in Section 6.6.

The overhead incurred by each retry is that of the additional WOM computation, plus that of reading another pair of physical pages. However when using Polar WOM codes, the extra read overhead can be avoided. Due to

the probabilistic nature of these codes, the retry can be performed using the *same* physical invalidated data, while only altering an internal encoding parameter [6]. Successive retries are independent, yielding a similar overall success probability as with the general retry method described above. Thus, in our design, a WOM code failure triggers one retry, without incurring an additional read. If the retry fails, the page is written as a first write. We evaluate the overhead incurred by retries in both the special and general cases in Section 6.6.

6 Evaluation

We performed a series of trace driven simulations to verify that second writes in Reusable SSD are indeed 'free.' We answer the following questions. (1) How many erasures can be eliminated by second writes? (2) What is the effect of second writes on read and write latency? (3) Do second writes incur additional overheads? (4) How sensitive are the answers to design and system parameters? In addition, we establish the importance of our main design components.

6.1 Experimental Setup

We implemented second writes within the MSR SSD extension [1] of DiskSim [5]. We configured the simulator with two planes in each flash chip, so that each plane is an independent allocation pool, as described in Section 5.4. We allow for parallel execution of commands in separate planes within each chip. Second writes are simulated by mapping a logical page to two physical pages that have to be read and written. We use a random number generator to simulate encoding failures, and disable second writes on blocks that reach 30% of their lifetime.

The SSD extension of DiskSim implements a greedy garbage collector with wear leveling and migrations, copying cold data into blocks with remaining lifetime lower than a threshold. We modify this process as described in Section 5.4, so that it applies only to victim blocks that are going to be erased. Garbage collection is invoked when the total of clean *and* recycled blocks in the plane drops below a threshold of 1%. DiskSim initializes the SSD as full. Thus, every write request in the trace generates an invalidation and an out-of-place write. We use two common overprovisioning values, 7% and 28%, which represent consumer and enterprise products, respectively [52]. We refer to the unmodified version of DiskSim, with first writes only, as the *standard SSD*.

We evaluate our design using real world traces from two sources. The first is the MSR Cambridge workload [40], which contains traces from 36 volumes on 13 servers. The second is the Microsoft Exchange workload [41], from one of the servers responsible for Microsoft employee e-mail. The volumes are configured as RAID-1 or RAID-5 arrays, so some of them are too big to fit on a single SSD.

Volume	Requests (M)	Drive size (GB)	Requests/sec	Peak writes/sec	Write ratio	Average write size (KB)	Total writes (GB)
zipf(1,2)	3	4	200	200	1	4	12
src1_2	2	16	3.15	95.69	0.75	33	45
stg_0	2		3.36	10.23	0.85	10	16
hm_0	4	32	6.6	48.62	0.64	9	23
rsrch_0	1.5		2.37	6.16	0.91	9	11
src2_0	1.5		2.58	19.95	0.89	8	10
ts_0	2		2.98	14.4	0.82	8	12
usr_0	2.5		3.7	12.54	0.6	11	14
wdev_0	1		1.89	7.41	0.8	8	7
prxy_0	12.5	64	20.7	42.57	0.97	7	83
mds_0	1		2	5.61	0.88	8	8
proj_0	4		6.98	132.71	0.88	41	145
web_0	2		3.36	18.52	0.7	13	17
prn_0	5.5	128	9.24	145.1	0.89	11	54
exch_0	4		45.28	90.56	0.92	27	94
src2_2	1	256	1.91	271.41	0.7	55	42
prxy_1	24		278.83	120.81	0.35	13	106

Table 3: Trace characteristics. The duration of all production traces is one week, except prxy_1 and exch_0, which are one day.

Manufacturer	Type	Pages/ Block	Read (us)	Write (ms)	Erase (ms)	Size (Gb)
Toshiba [55]	SLC	64	30	0.3	3	32
Samsung [49]	MLC	128	200	1.3	1.5	16
Hynix [19]	MLC	256	80	1.5	5	32

Table 4: NAND flash characteristics used in our experiments.

We used the 16 traces whose address space could fit in an SSD size of 256GB or less, and that included enough write requests to invoke the garbage collector on that drive. These traces vary in a wide range of parameters, summarized in Table 3. We also used two synthetic workloads with Zipf distribution, with exponential parameter $\alpha = 1$ and 2. Note that a perfectly uniform workload is unsuitable for the evaluation of second writes, because all the data is essentially cold.

We use parameters from 3 different NAND flash manufacturers, corresponding to a wide range of block sizes and latencies, specified in Table 4. While the flash packages are distributed in different sizes, we assume they can be used to construct SSDs with the various sizes required by our workloads. Due to alignment constraints of DiskSim, we set the page size to 4KB for all drives. To maintain the same degree of parallelism for all equal sized drives, we assume each chip contains 1GB, divided into two planes. The number of blocks in each plane varies from 512 to 2048, according to the block size. The MSR SSD extension implements one channel for the entire SSD, and one data path (way) for each chip. We vary the number of chips to obtain the drive sizes specified in Table 3.

6.2 The Benefit of Second Writes

Write amplification is commonly used to evaluate FTL performance, but is not applicable to our design. Second writes incur twice as many physical writes as first writes but these writes are performed after the block's capacity

has been exhausted by first writes, and do not incur additional erasures. Thus, to evaluate the performance of Reusable SSD, we measure the relative number of erasures and relative response time of our design, compared to the standard SSD. Note that in a standard SSD, the number of erasures is an equivalent measure to write amplification. In all our figures, the traces are ordered by the amount of data written compared to the physical drive size, i.e., in mds_0 the least data was written, and in zipf (1) and (2) the most.

Erasures. Figure 7 shows the relative number of erasures of Reusable SSD compared to the standard SSD. Recall that according to the best case analysis, Reusable SSD can write up to 50% more pages on a block before its erasure, corresponding to a reduction of 33% in the number of block erasures. In most traces, the reduction is even slightly better, around 40%. This is due to the finite nature of our simulation – some of the recycled blocks were not erased within the duration of the simulation. Since Reusable SSD can apply second writes in the first 30% of the drive's lifetime, it performs additional writes equivalent to a lifetime extension of 15%.

In several traces the reduction was less than 33%, because a large portion of the data was written in I/O requests larger than the 64KB threshold. The corresponding pages were classified, mostly correctly, as cold and written as first writes, so the full potential of second writes was not realized. In traces src2_2, prxy_1, exch_0, prxy_0, proj_0 and src1_2, the percentage of such cold pages was 95, 44, 83, 32, 86 and 78, respectively (compared to 1%-15% in the rest of the traces). We further investigate the effect of the hot/cold threshold in Section 6.5.

The different block sizes of the drives we used affect the *absolute* number of erasures, both for standard and for Reusable SSD. However, the *relative* number of erasures was almost identical for all block sizes.

The synthetic Zipf workloads have no temporal locality, so more pages remain valid when blocks are erased or recycled, especially with zipf(1) which has less access skew than zipf(2). With low overprovisioning, Reusable SSD is less efficient in reducing erasures for this workload because there are fewer invalid pages for use in second writes.

The reduction in the number of erasures usually depends on the drive's overprovisioning (OP). Higher overprovisioning means the maximum number of blocks that can be in the reused or recycled states ($2R$) is higher, thus allowing more pages to be written in second writes. In the extreme case of trace src2_2 with OP=28% , all writes could be accommodated by the overprovisioned and recycled blocks, thus reducing the number of erasures to 0. In the other traces with a high percentage of cold data, the number of erasures did not decrease further with overprovisioning because fewer blocks were required to accommodate the "hot" portion of the data. We examine a wider range of overprovisioning values in Section 6.6.

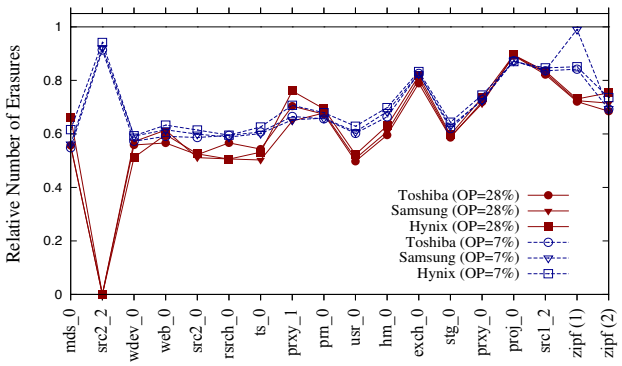


Figure 7: Relative number of erasures of Reusable SSD compared to the standard SSD. The reduction in erasures is close to the expected 33% in most cases.

Performance. Overprovisioning notably affects the performance of first as well as second writes. When overprovisioning is low, garbage collection is less efficient because blocks are chosen for cleaning while they still have many valid pages that must be moved. This degrades performance in two ways. First, more block erasures are required because erasures do not generate full clean blocks. Second, cleaning before erasure takes longer, because the valid pages must be written first. This is the notorious delay caused by write amplification, which is known to increase as overprovisioning decreases.

Indeed, the reduction in erasures achieved by Reusable SSD further speeds up I/O response time when overprovisioning is low. Figure 8 shows the reduction in average I/O response time achieved by Reusable SSD compared to the standard SSD. I/O response time decreased by as much as 15% and 35%, with OP=28% and OP=7%, respectively.

The delay caused by garbage collection strongly depends on write and erase latencies, as well as on the block size. When overprovisioning is low (7%) and writes cause a major delay before erasures, the benefit from second writes is greater for drives with longer write latencies – the benefit in the Hynix setup is up to 60% greater than in the Toshiba setup. When overprovisioning is high (28%) and the cost of cleaning is only that of erasures, the benefit from second writes is greater for drives with small blocks whose absolute number of erasures is greater – the benefit in the Toshiba setup is up to 350% greater than the benefit in the Hynix setup.

6.3 The Benefit of Parallel Execution

To establish the importance of parallelizing second writes, we implemented a "sequential" version of our design, where second writes are performed on a pair of contiguous invalid pages on the same block. The two planes in each chip can still be accessed concurrently – they each have an independently written $RecycledActive$ block.

The reduction in the number of cleans is almost iden-

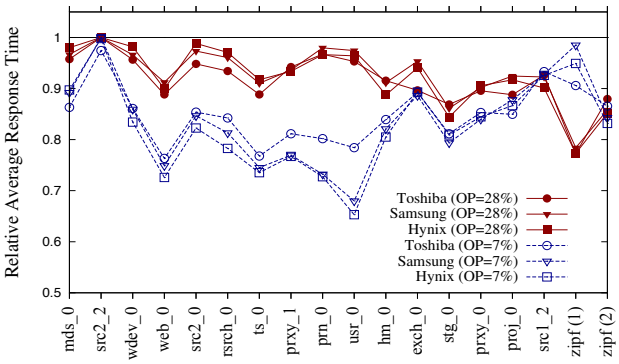


Figure 8: Relative I/O response time of Reusable SSD compared to the standard SSD. The reduction in erasures reduces I/O response time, more so with lower overprovisioning.

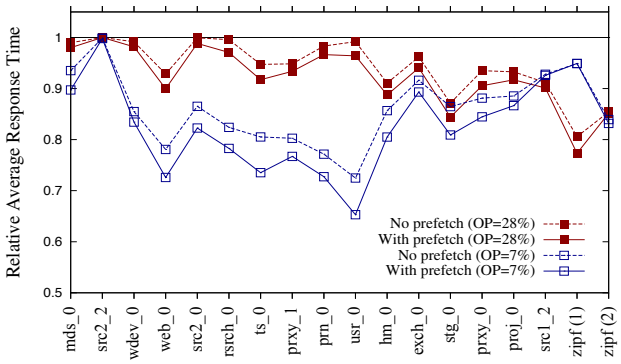


Figure 9: Relative I/O response time of Reusable SSD with and without prefetching, in the Hynix setup. Prefetching always reduces I/O response time.

tical to that of the parallel implementation, but the I/O response time increases substantially. In the majority of traces and setups, second writes increased the I/O response time, by an average of 15% and as much as 31%. We expected such an increase. Data written by a second write requires twice as many pages to be accessed, both for reading and for writing, and roughly 33% of the data in each trace is written as second writes.

As a matter of fact, the I/O response time increased less than we expected, and sometimes *decreased* even with the sequential implementation. The major reason is the reduction in erasures – the time saved masks some of the extra latency of second writes. Another reason is that although roughly 33% of the data was written in second writes, only 1%-19% of the reads (2%-6% in most traces) accessed pages written in second writes. This corresponds to a well-known characteristic of secondary storage, where hot data is often overwritten without first being read [53].

Nevertheless, an increase of 15%-30% in average response time is an unacceptable performance penalty. Our parallel design complements the reduction in erasures with a significant reduction in I/O response time.

6.4 The Benefits of Prefetching Invalid Pages

To evaluate the contribution of prefetching invalid pages, we disabled prefetching and repeated our experiments. Figure 9 shows the results for the Hynix setup with OP=28% and OP=7%. These are the two setups where second writes achieved the least and most reduction in I/O response time, respectively. These are also the setups where the contribution of prefetching was the highest and lowest, respectively.

With OP=7%, and specifically the Hynix setup, the reduction in erasures was so great that the extra reads before second writes had little effect on overall performance. Prefetching reduced I/O response time by an additional 68% at most. With OP=28%, where the reduction in I/O response time was less substantial, prefetching played a more important role, reducing I/O response time by as much as $\times 21$ more than second writes without it.

The results for the rest of the setups were within this range; the average I/O response time for second writes with prefetching was 120% shorter than without it. Prefetching never delayed reads or first writes to the point of degrading performance.

6.5 The Benefits of Hot/Cold Classification

When an I/O request size is equal to or larger than the hot/cold threshold, its data is classified as cold and written in first writes. We examine the importance of separating hot data from cold, and evaluate the sensitivity of our design to the value of the threshold. We varied the threshold from 16KB to 256KB. Figure 10 shows the results for the Toshiba setup – the results were similar for all drives. We present only the traces for which varying the threshold changed the I/O response time or the number of erasures. The results for 256KB were the same as for 128KB.

Figure 10(a) shows that as the threshold increases, more data is written in second writes, and the reduction in the number of erasures approaches the expected 33%. However, increasing the threshold too much sometimes incurs additional cleans. For example, in prn_0, data written in requests of 64KB or larger nearly doubled the valid count of victim blocks chosen for cleaning, incurring additional delays as well as additional erase operations. Figure 10(c) shows that a reduction [increase] in the number of erasures due to higher thresholds entails a reduction [increase] in the relative I/O response time.

Figures 10(b) and 10(d) show the results for the same experiments with OP=28%. The additional overprovisioned capacity extends the time between cleans, to the point where even the cold data is already invalid by the time its block is erased. Both the number of erasures and the I/O response time decrease as more data can be written in second writes. Specifically, Figure 10(b) shows that the full "50% free" writes can be achieved in enterprise class setups. Still, the hot/cold classification guarantees better performance, possibly at the price of limiting the reduction in erasures.

An adaptive scheme can set the threshold according to

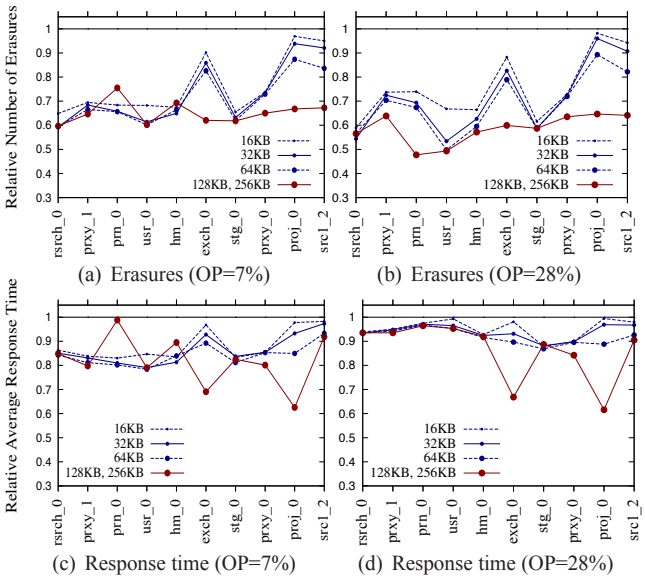


Figure 10: Relative I/O response time and erasures when varying the hot/cold classification threshold. Increasing the threshold too much might increase I/O response time.

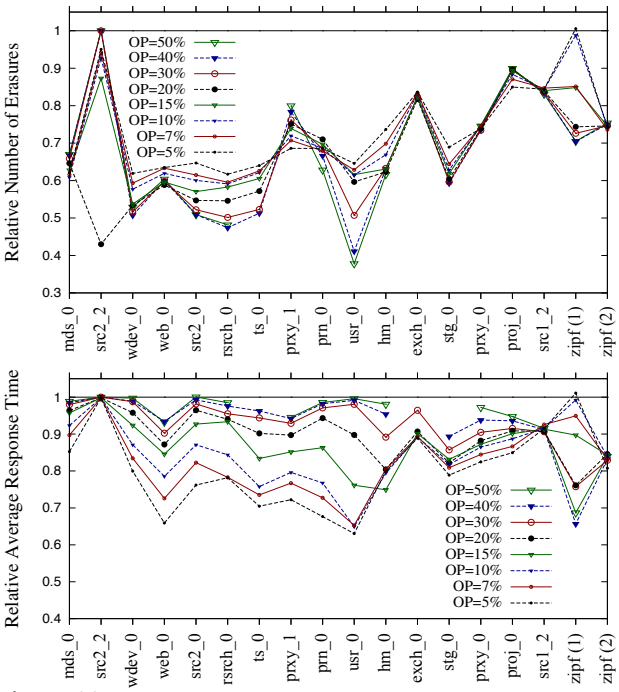


Figure 11: Relative number of erasures (top) and average I/O response time (bottom) with the Hynix setup, with varying overprovisioning ratios.

the observed workload to optimize both objectives, but is outside the scope of this work. Alternatively, classification can be done using recent optimized schemes (see Section 7) for more accurate results. Note that regardless of the classification scheme, Reusable SSD also separates application writes from garbage collection writes. This separation is expected to reduce the number of erasures compared to the standard SSD, even without second writes.

6.6 Sensitivity Analysis

Overprovisioning. For a comprehensive analysis we repeated our experiments, varying the overprovisioning value from 5% to 50%[3]. For all the drives and traces, the number of erasures and the I/O response time decreased as overprovisioning increased, both in the standard SSD and in Reusable SSD. Figure 11 shows the relative number of erasures and relative I/O response time of Reusable SSD compared to the standard SSD. We show results for the Hynix setup, where varying the overprovisioning value had the largest effect on these two measures.

These results support our observation in Section 6.2, that the reduction in erasures is larger when overprovisioning is higher, except in traces that have a high portion of cold data written as first writes. Reusable SSD reduces I/O response time more with lower overprovisioning, where erasures cause longer delays. The maximal variation in relative average response time was 24%, 32%, and 35% in the Toshiba, Samsung and Hynix setups, respectively.

WOM encoding failures. Reusable SSD is designed to work with any WOM code construction that satisfies the requirements specified in Section 3. To evaluate the sensitivity of our design to WOM code characteristics, we repeated our experiments, varying the encoding success rate from 75% to 100%.

In the first set of experiments we disable retries completely, so they serve as a *worst case* analysis. On a WOM encoding failure we default to a first write on the $CleanActive$ block. Every such failure incurs the overhead of an additional ECC computation, because ECC must be computed for the logical data. The ECC for a 4KB page can usually be computed in less than $10\mu s$ [60]. To account for changes in page size, ECC and WOM code, and as a worst case analysis, we set the overhead to half the read access time in each drive.

Figure 12(a) shows the relative I/O response time of Reusable SSD without retries, compared to the standard SSD. Surprisingly, varying the success rate resulted in a difference in relative I/O response time of less than 1% for all traces with OP=7%, and for most traces with OP=28%. The reduction in erase operations was not affected at all. We show here only the traces for which the difference was larger than 1%. We show the results with the Toshiba setup because the differences with the other setups were even smaller. The reason for such small differences is that in most traces, the maximum allowed number of reused and recycled blocks does not accommodate all the hot data, and some hot data is written as first writes when no recycled block is available. Thus, WOM encoding failures

[3]The address space of ts_0, exch_0 and stg_0 was too large to fit in the respective drive sizes from Table 3 with OP=50% (and OP=40% for exch_0). Thus, the data points corresponding to those traces and OP values are missing.

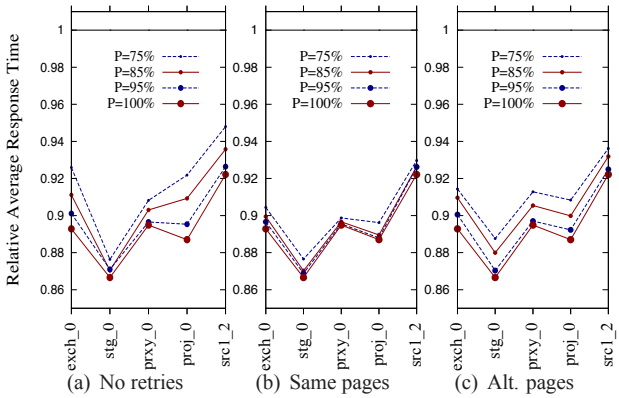


Figure 12: The sensitivity of Reusable SSD to varying WOM encoding success rates with no retries (a), retries on the same physical pages (b) and retries on alternative physical pages (c), with the Toshiba setup and OP=28%.

simply distribute the hot data differently, incurring only the additional ECC computation delay.

Figure 12(b) shows the relative I/O response time of Reusable SSD with retries, as described in Section 5.6. A retry incurs the same overhead as the first WOM encoding. If that retry fails, extra overhead is incurred by the ECC computation of the first write. Note that with one retry, the overall probability of successful encoding with $P = 75\%$ is $P' = 1 - (1 - 0.75)^2 = 93.75\%$. Indeed, the performance of Reusable SSD with $P = 75\%$ is almost equal to that of Reusable SSD without retries and $P = 95\%$. Similarly, the relative I/O response time with $P = 95\%$ and one retry is almost equal to that of using a WOM code with no encoding failures ($P = 100\%$).

We also examine the applicability of our design to WOM codes that do not guarantee independent success probability on the same invalid data. Thus, we ran one more set of experiments where, upon a WOM encoding failure, an additional pair of invalid pages is read, and the encoding is retried on these pages. In this variation, the overhead of retrying is the same as in our design, plus an additional read latency. Our results, presented in Figure 12(c), show that the additional overhead is negligible when P is high (95%), but nonnegligible for smaller values of P ($\leq 85\%$). In addition, unlike the first two approaches, this overhead appeared in the rest of the traces as well, and also with OP=7%. Still, even in those cases, the I/O response times were reduced substantially compared to the standard SSD, and the difference between $P = 100\%$ and $P = 75\%$ was always below 4%.

Energy consumption. According to a recent study [39], the energy consumption of an erase operation is one order of magnitude larger than that of a write operation, but the energy it consumes per page is the smallest of all operations. Of all measured operations, writes are the major contributor to energy consumption in flash chips. In Reusable SSD, roughly 33% of the pages are written in second writes, programming two pages instead of one. Thus, one might expect its energy consumption to increase in proportion to the increase in physical writes.

However, this same study also showed the energy consumed by writes to depend on the number of programmed cells. But not only do second writes not require programming twice as many cells, their overall number of programmed bits is expected to equal that of first writes [6]. We thus expect the energy consumption to decrease, thanks to the reduction in erasures that comes with second writes. Measurements on a naive implementation of second writes showed such a reduction [14], and we believe these results will hold for Reusable SSD. A more accurate evaluation remains part of our future work.

7 Related Work

The Flash Translation Layer is a good candidate for manipulating flash traffic to extend SSD lifetime. Most FTLs implement some notion of wear leveling, where cold data is migrated to retired or about-to-be-retired blocks, and blocks are allocated for writing according to their erase count or wear [1, 20, 24, 27, 28, 38, 42]. Buffering [29] and even deduplication [16] are used by some FTLs to reduce the number of flash writes.

Another approach reduces write traffic to the SSD by eliminating write operations at higher levels of the storage hierarchy. Such methods include a hard disk based write cache [53], specialized file systems and data bases [11, 31, 34, 36, 38], and admission control in flash based caches [18, 43, 46].

A recent analytic study showed that separating hot and cold data can minimize write amplification so that it approaches 1 [13]. Indeed, many FTLs write hot and cold data into separate partitions [9, 20, 27, 28, 38, 42, 54]. They classify hot data according to I/O request size [9, 20], time and frequency of write [38, 54], and whether the write was generated by the garbage collector [42].

Reusable SSD separates hot and cold data, and applies wear leveling and migration to blocks that are about to be erased. However, the specific classification or wear leveling technique is orthogonal to our design, and can be replaced with any of the state-of-the-art methods to combine their advantages with those of second writes. Similarly, when some of the write traffic is eliminated from the SSD, Reusable SSD can apply roughly 33% of the remaining writes to reused blocks, eliminating additional erasures.

More intrusive methods for extending SSD lifetime include modifying the voltage of write and erase operations [24], and even explicitly delaying requests to allow cell recovery [30]. They incur an overhead that limits the SSD's performance. Reusable SSD extends SSD lifetime without requiring any changes in flash hardware. More importantly, it improves — rather than degrades — performance. Still, Reusable SSD can also be combined with

such methods, to further extend device lifetime.

While the number of erasures is the most commonly used measure of device lifetime, recent studies show that cell programming has a substantial impact on their wear. They show that programming MLC cells as SLC [26], or occasionally 'relieving' them from programming [27] can significantly slow down cell degradation, regardless of the number of erasures. Second writes result in a higher average voltage level of flash cells, possibly increasing their wear. Thus, with second writes, 50% more writes can be performed before each erasure, but the number of 'allowed' erasures might decrease. However, as long as the increase in cell wear is smaller than 50%, second writes extend device lifetime. Since cell degradation is not linear with average voltage level, the magnitude of this effect cannot be derived from previous studies, and remains to be verified in future work. Our analysis of the benefits of Reusable SSD is conservative, disabling second writes on all pages after 30% of the block's lifetime. A more accurate model of cell wear can facilitate additional second writes on SLC flash or on the LSB pages in MLC flash.

Several studies suggested using WOM codes to extend SSD lifetime. Their designs are all based on an increased page size, as in Figure 2, resulting in greatly reduced capacity [4, 14, 21, 35, 57]. In [42], the capacity loss is bound by limiting second writes to several blocks. The authors of [23] assume the logical data has been compressed by the upper level, to allow for the overhead of WOM encoding. None of these studies address the additional latencies of reading invalid data before encoding and of reading and writing larger pages. In addition, most of them rely on capacity achieving codes, ignoring their high complexity or their nonnegligible failure rate [35, 42]. The design of Reusable SSD addresses all the practical aspects of second writes with off-the-shelf flash products and efficient coding techniques, achieving both performance improvements and a lifetime extension of up to 15%.

The above studies use write amplification to evaluate their designs, but it is not the correct figure of merit for multiple writes. Consider a best case example where a code with minimal 29% space overhead achieves a write amplification of 1. Still, the amount of physical data written is 29% more than the logical data written by the application. Thus, for correct evaluation, the number of erasures incurred in various designs should be compared, on SSDs with the same block size and overprovisioning.

The use of WOM codes has also been suggested for extending PCM lifetime [22, 32, 59]. The corresponding studies show a reduction in energy consumption and cell wear, but sacrifice either capacity, performance, or both.

8 Conclusions and Future Directions

We presented Reusable SSD, a practical design for applying second writes to extend SSD lifetime while significantly improving performance. Our design is general and is applicable to current flash architectures, requiring only minor adjustments within the FTL, without additional hardware or interface modifications.

Nevertheless, more can be gained from Reusable SSD as technology advances in several expected directions. Flash architectures that allow for higher degrees of parallelism can accommodate third and maybe fourth writes, combining 4 and 8 physical pages per logical page, respectively [6]. As the hardware implementation of Polar WOM codes matures, its encoding overheads will decrease [3, 50], enabling faster retries, and possibly use of constructions with higher success probability. Similarly, stronger ECCs can compensate for increasing BERs, increasing the percentage of a block's lifetime in which it can be recycled before erasure.

Finally, most previously suggested schemes for extending SSD lifetime are orthogonal to the design of Reusable SSD, and can be combined with second writes. The performance improvement achieved by Reusable SSD can mask some of the overheads of those schemes that incur additional latencies.

Acknowledgments

We thank our shepherd Peter Desnoyers and the anonymous reviewers for their insightful comments that helped improve this paper. We thank Yue Li for his help with the Polar WOM code experiments, Ran Koretzki for his help with the simulation setup, and Abdalla Amaria and Muhammad Musa for their help with the traces.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conf. (ATC)*, 2008.

[2] E. Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Trans. on Inform. Theory*, 55(7):3051–3073, 2009.

[3] A. Balatsoukas-Stimming, A. Raymond, W. J. Gross, and A. Burg. Hardware architecture for list successive cancellation decoding of polar codes. *IEEE Trans. on Circuits and Systems II: Express Briefs*, 61(8):609–613, Aug 2014.

[4] A. Berman and Y. Birk. Retired-page utilization in write-once memory – a coding perspective. In *IEEE Intl. Symp. on Inform. Theory (ISIT)*, 2013.

[5] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim simulation environment version 4.0 reference manual, May 2008.

[6] D. Burshtein and A. Strugatski. Polar write once memory codes. *IEEE Trans. on Inform. Theory*, 59(8):5088–5101, 2013.

[7] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Conf. on Design, Automation and Test in Europe (DATE)*, 2012.

[8] P. Cappelletti, R. Bez, D. Cantarelli, and L. Fratin. Failure mechanisms of flash cell in program/erase cycling. In *Intl. Electron Devices Meeting (IEDM) Technical Digest*, 1994.

[9] M.-L. Chiao and D.-W. Chang. ROSE: A novel flash translation layer for NAND flash memory based on hybrid address translation. *IEEE Trans. on Computers*, 60(6):753–766, 2011.

[10] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2009.

[11] B. Debnath, M. Mokbel, D. Lilja, and D. Du. Deferred updates for flash-based storage. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2010.

[12] P. Desnoyers. What systems researchers need to know about NAND flash. In *USENIX Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[13] P. Desnoyers. Analytic models of SSD write performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2009.

[15] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Intl. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[16] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2011.

[17] C. Heegard. On the capacity of permanent memory. *IEEE Trans. on Inform. Theory*, 31(1):34–41, 1985.

[18] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2013.

[19] Hynix. 32Gbit (4096M x 8bit) legacy NAND flash memory. Datasheet, 2012.

[20] S. Im and D. Shin. ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer. *J. Syst. Archit.*, 56(12):641–653, Dec. 2010.

[21] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Writing cosets of a convolutional code to increase the lifetime of flash memory. In *Allerton Conf. on Communication, Control, and Computing (Allerton)*, 2012.

[22] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Coset coding to extend the lifetime of memory. In *IEEE Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2013.

[23] A. Jagmohan, M. Franceschini, and L. Lastras. Write amplification reduction in NAND flash through multi-write coding. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2010.

[24] J. Jeong, S. S. Hahn, S. Lee, and J. Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2014.

[25] A. Jiang, Y. Li, E. Gad, M. Langberg, and J. Bruck. Joint rewriting and error correction in write-once memories. In *IEEE Intl. Symp. on Inform. Theory (ISIT)*, 2013.

[26] X. Jimenez, D. Novo, and P. Ienne. Libra: Software controlled cell bit-density to balance wear in NAND flash. *ACM Trans. Embed. Comput. Syst. (TECS)*, 14(2).

[27] X. Jimenez, D. Novo, and P. Ienne. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2014.

[28] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *Intl. Symp. on Computer Architecture (ISCA)*, 2008.

[29] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2008.

[30] S. Lee, T. Kim, K. Kim, and J. Kim. Lifetime management of flash-based SSDs using recovery-aware dynamic throttling. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[31] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *USENIX Annual Technical Conf. (ATC)*, 2014.

[32] J. Li and K. Mohanram. Write-once-memory-code phase change memory. In *Design, Automation and Test in Europe Conf. and Exhibition (DATE)*, 2014.

[33] R.-S. Liu, C.-L. Yang, and W. Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[34] Y. Lu, J. Shu, and W. Wang. ReconFS: A recon-

structable file system on flash storage. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2014.

[35] X. Luojie, B. M. Kurkoski, and E. Yaakobi. WOM codes reduce write amplification in NAND flash memory. In *IEEE Global Communications Conf. (GLOBECOM)*, 2012.

[36] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan. NVMKV: A scalable and lightweight flash aware key-value store. In *USENIX Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[37] N. Mielke, H. Belgal, I. Kalastirsky, P. Kalavade, A. Kurtz, Q. Meng, N. Righos, and J. Wu. Flash EEPROM threshold instabilities due to charge trapping during program/erase cycling. *IEEE Trans. on Device and Materials Reliability*, 4(3):335–344, Sept 2004.

[38] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[39] V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M. Stan, and S. Swanson. Modeling power consumption of NAND flash memories using FlashPower. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 32(7):1031–1044, July 2013.

[40] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, Nov. 2008.

[41] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *ACM European Conf. on Computer Systems (EuroSys)*, 2009.

[42] S. Odeh and Y. Cassuto. NAND flash architectures reducing write amplification through multi-write codes. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2014.

[43] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[44] Y. Pan, G. Dong, and T. Zhang. Exploiting memory device wear-out dynamics to improve NAND flash memory system performance. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2011.

[45] K. Prall. Scaling non-volatile memory below 30nm. In *IEEE Non-Volatile Semiconductor Memory Workshop*, 2007.

[46] T. Pritchett and M. Thottethodi. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Intl. Symp. on Computer Architecture (ISCA)*, 2010.

[47] R. L. Rivest and A. Shamir. How to Reuse a Write-Once Memory. *Inform. and Contr.*, 55(1-3):1–19, Dec. 1982.

[48] D. Roberts, T. Kgil, and T. Mudge. Integrating NAND flash devices onto servers. *Commun. ACM*, 52(4):98–103, Apr. 2009.

[49] Samsung. 16Gb F-die NAND flash, multi-level-cell (2bit/cell). Datasheet, 2011.

[50] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross. Fast polar decoders: Algorithm and implementation. *IEEE Journal on Selected Areas in Communications*, 32(5):946–957, May 2014.

[51] A. Shpilka. New constructions of WOM codes using the Wozencraft Ensemble. *IEEE Trans. on Inform. Theory*, 59(7):4520–4529, 2013.

[52] K. Smith. Understanding SSD over-provisioning. *EDN Network*, January 2013.

[53] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2010.

[54] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[55] Toshiba. 32 GBIT (4G X 8 BIT) CMOS NAND E2PROM. Datasheet, 2011.

[56] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf. Codes for write-once memories. *IEEE Trans. on Inform. Theory*, 58(9):5985–5999, 2012.

[57] E. Yaakobi, J. Ma, L. Grupp, P. H. Siegel, S. Swanson, and J. K. Wolf. Error characterization and coding schemes for flash memories. In *IEEE GLOBECOM Workshops (GC Wkshps)*, 2010.

[58] C. Zambelli, M. Indaco, M. Fabiano, S. Di Carlo, P. Prinetto, P. Olivo, and D. Bertozzi. A cross-layer approach for new reliability-performance trade-offs in MLC NAND flash memories. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2012.

[59] X. Zhang, L. Jang, Y. Zhang, C. Zhang, and J. Yang. WoM-SET: Low power proactive-SET-based PCM write using WoM code. In *IEEE Intl. Symp. on Low Power Electronics and Design (ISLPED)*, 2013.

[60] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2013.