Space and Write Overhead are Inversely Proportional in Flash Memory

Philipp Brandes Roger Wattenhofer
ETH Zurich
{pbrandes,wattenhofer}@ethz.ch

Abstract

In this paper we consider the trade-off between space and write overhead of flash memory. Every flash memory has additional space to compensate for wear leveling; we denote the space overhead with σ. Furthermore, every flash memory is forced to rewrite valid data when a block is erased; we denote the write overhead with ω. We show that space and write overhead are inversely proportional with $\sigma\omega \geq 1$. We also present an algorithm that proves that our analysis is tight, as it achieves $\sigma\omega = 1$ in a worst case. Moreover, we analyze a setting with the data being updated uniformly at random, or not at all.

Categories and Subject Descriptors D.4.2 [*Storage management*]: Garbage collection

Keywords wear leveling, write amplification, flash memory, solid state disks, FTL

1. Introduction

Flash memory is omnipresent in smartphones and cameras. Solid state disks (SSDs) based on NAND flash increasingly become the default choice for computers of any kind, from laptops to servers. Unfortunately, NAND flash is also known for having a limited lifetime, as the multi-level cells used to physically store data can only be written and erased about 10,000 times until they are worn out (Agarwal and Marrow 2010). Afterwards, cells can no longer reliably store data. As a result, despite not having moving mechanical parts, flash memory is perceived as more failure prone than traditional hard disk drives.

This problem is aggravated due to another technical property of flash memory. The smallest storage unit, a page, cannot be erased on its own, but only with the rest of the pages of a block. A block usually consists of 64 pages, with each page being able to store up to 2048 bytes (Micron 2008, 2010, 2011). When a block is erased to create free pages, the valid data stored in the block needs to be preserved, i.e., written to a page in another block to avoid data loss. Such writes are not issued by the user but by the system, and as such not necessary from the user's point of view. We call them *write overhead*. Since write overhead wears out our flash memory (and consequently decreases its lifetime), it should be minimized.

If some data gets updated often and it is always stored in the same block, this block will be worn out quickly. Such an access pattern is common in a real world environment (Chang and Du 2009; Chang and Huang 2011). Instead, a logical data address is mapped to a physical page address using a flash translation layer (FTL). FTL improves write overhead significantly.

In Section 2, we will discuss various advanced heuristic algorithms that manage to extend the lifetime of flash memory by moving data in a sophisticated way. In particular, the internal storage capacity of the hardware is generally higher than announced to the consumer, and this *space overhead* is used to move data less often. It is known that write overhead and space overhead are related; in general this relation depends on the application that is using the flash memory.

We study the trade-off between *write overhead* ω and *space overhead* σ in Section 5. We show that there is an inversely proportional law connecting the two. If we assume a worst possible application (an application that accesses data in a worst-case way), any FTL algorithm must obey

$$\sigma\omega \geq 1.$$

Moreover, in Section 4 we show that this inequality is tight: Our FTL algorithm, the cycling algorithm, can achieve equality, and hence is worst-case optimal.

In Section 6 we analyze the cycling algorithm with an average-case analysis. More concretely, we assume that some data is static and never updated. The remaining data is dynamic and updated uniformly at random. In this setting the cycling algorithm achieves $\sigma\omega < 1$, more precisely the

SYSTOR '15, May 26–28, 2015, Haifa, Israel.

http://dx.doi.org/10.1145/2757667.2757682

write overhead is at most $\frac{\mu \cdot T + \delta \cdot T \cdot e^{\frac{1-\alpha}{\alpha\delta}}}{T - (\mu \cdot T + \delta \cdot T \cdot e^{\frac{1-\alpha}{\alpha\delta}})}$ with μ, δ, and φ being the fraction of the flash memory that is filled with static data, dynamic data, and no data, respectively.

2. Related Work

As mentioned in the introduction, FTLs have been around as long as flash memory. Thus, we can only provide an overview in this section.

The work closest to ours, as in that it also considers worst cases analysis, is by Ben-Aroya and Toledo (Ben-Aroy and Toledo 2011). In their model the flash memory has T blocks out of which only V are visible to the user, and the remaining ones solely exist for the sake of wear leveling. They show that there exists for every algorithm an input sequence such that the algorithm can fulfill at most $(T-V+1) \cdot H$ requests, with H being the maximal number of writes a block can endure. They assume that every block has exactly one page. Since this guarantees that there is always a block that has either only free pages or only invalid pages, this greatly simplifies the problem. It is therefore not necessary preserve the valid data stored in the block, i.e., write it to a page in another block to avoid data loss. Hence, write overhead does not exist.

Some papers assume a uniform distribution of write accesses and then try to find a closed form expression for write amplification depending on the over-provisioning (Agarwal and Marrow 2010; Xiang and Kurkoski 2011). They use powerful (and therefore in practice rather slow) FTL algorithms that can erase the block with the most invalid pages if no block with a free page exists. Some work analyses specific strategies, e.g., choosing the "best" block from a sliding window, again using a uniform distribution to model the write accesses (Hu et al. 2009). In our Section 6, some data is static and never updated and only the remaining data is accessed uniformly at random. We consider the trade-off between space and write overhead and give a general bound. The trim command, i.e., marking pages as invalid, and its effects on write amplification, are also studied under a uniform write distribution (Frankie et al. 2012). Non-uniform write distributions are analyzed in (Desnoyers 2014). They employ the least recently written strategy, which is similar to our cycling algorithm.

FTL heuristics have been around right from the start. In addition to the mostly secret implementations by hardware manufacturers, FTL algorithms have been published as well. We can only list a small subset of these algorithms. The so called dual-pool algorithm tries to store cold data, i.e., data that is not accessed much, in blocks which have been erased often (Chang 2007). This work has been refined later (Chang and Du 2009; Chang and Huang 2011; Chang et al. 2013). Another approach is to use a log buffer based FTL. This is a hybrid between page-level mapping and block-level mapping. Requests are written to log blocks. Once such a block is filled, the data is merged with other data from the same LBA and written into a new block, thus allowing block-level mapping. A popular FTL algorithm is commonly referred to as FAST (Lee et al. 2007). A survey about FTL algorithms can be found in (Chang et al. 2006).

With the emergence of PCM-based memory, wear leveling remains an important topic even beyond current NAND flash (Qureshi et al. 2009). Low level technical details about flash memory can be found in (Micron 2008, 2010, 2011). There exists a journaling file system that is developed exclusively with flash drives in mind (Woodhouse 2011).

Since details about flash memory and SSDs in particular were initially trade secrets and thus not accessible to research, simulators were written to compensate for the secrecy of the manufacturers. One of them, CPS-SIM (Lee et al. 2009), also takes the number of buses and low level clocks into account. DiskSim, a disk simulator framework, was extended to include energy usage by (Kim et al. 2009) and parallelism and write ordering by (Agarwal 2008).

3. Model

The flash memory we consider consists of n blocks $b_0, \ldots, b_{n-1}$. Each block b_i in turn consists of m pages $p_0^i, \ldots, p_m^i$. A page is the smallest accessible unit, i.e., a page can only be accessed (read or written) as a whole. The physical pages in flash memory store data.

A physical page continuously cycles through three distinct states: *free*, *valid*, and *invalid*. First the page is free, and data can be written into it. After data was written into the page, the page is valid as it stores useful data. Later the data might be updated, and stored in a free flash memory page. The old page storing the old version of that data becomes invalid. However, the old page cannot be written again, first it needs to be erased. Because of technical limitations of NAND flash, all pages in a block must be erased together. Before a block can be erased, all valid pages must first be moved (written to other pages). After a block is erased, all pages in the block are free again, and a life cycle of the page is complete.

Let T be the total number of pages of our flash memory, that is, $T = m \cdot n$. Let V denote the maximum number of valid pages that can be stored, the capacity available to the user. The ratio $\alpha = \frac{T}{V} > 1$ is referred to as *over-provisioning*. We denote the *space overhead* with $\sigma = \frac{T-V}{V} = \alpha - 1$. This σ is the relative amount of storage that is hidden from the user, and used to improve the lifetime of our flash memory.

Since blocks can be erased only about 10,000 times, unbalanced data access will wear out some blocks earlier than others. The flash translation layer (FTL) maps logical addresses to physical pages.[1] This mapping is not static but

[1] Some FTLs instead use a block based mapping scheme, where flash memory is addressed on the level of blocks. In this paper we address memory on the page level.

evolves over the life time of the flash memory, and is determined by the FTL algorithm. Note that the FTL resides inside the flash memory and is invisible to the user.

To write data on flash memory, a *write request* must to be issued. A write request contains the data and a logical address. The task of an FTL algorithm is to accept write requests and choose for each write request a physical page, where the data will be stored. We distinguish between two types of write requests. To write new data or update existing data, an *external* write request will be issued. Apart from Section 6, we assume that an adversary issues these external write requests. The adversary knows the FTL algorithm and the current state of the flash memory at any moment. Note that whenever a block b_i that still contains valid pages will be erased, these valid pages first need to be written to a free page to avoid data loss. Thus, any FTL algorithm also needs to issue *internal* write requests to avoid data loss.

Let E_k^i and I_k^i be the number of external and internal write requests written into a physical page p_k^i of block b_i, respectively. Thus, the total number of external write requests is $E = \sum_{i=0}^{n-1} \sum_{k=1}^{m} E_k^i$ and the total number of internal write requests is $I = \sum_{i=0}^{n-1} \sum_{k=1}^{m} I_k^i$. This allows to define the average number of external write requests per physical page as E/T. The *local write overhead* ω_k^i of page p_k^i is $\frac{E_k^i+I_k^i}{\frac{E}{T}} - 1$, i.e., the total number of write requests over the average number of write requests minus 1. The minus 1 is included in the term since we focus on the overhead. We can now define the *write overhead* ω as

$$\begin{aligned}\omega &= \max_{0\le i\le n-1, 1\le k\le m} \omega_k^i \\ &= \max_{0\le i\le n-1, 1\le k\le m} \left\{ \frac{E_k^i + I_k^i}{\frac{E}{T}} - 1 \right\}.\end{aligned}$$

Intuitively, this is means that we are interested in the worst page. In the literature, *write amplification* A is defined as as the total number of write requests W over the total number of external write requests, i.e., $A = \frac{W}{E} = \frac{E+I}{E}$.

We assume that the flash memory has a volatile memory (RAM) of two blocks to temporarily store m incoming write requests and every page of one single block. Hence, after buffering m write requests a block b_i can be read and its valid pages stored in the buffer, then block b_i can be erased, and finally the m write requests can be written into block b_i.

4. Cycling Algorithm

We now present the *cycling algorithm*. It keeps track of the most recent block b_i it has written data into. After m write requests, these write requests are written as a whole to the next block b_j with $j = i + 1 \bmod n$. For every valid page of this block a new internal write request is issued to avoid data loss. Note that if b_j contains only valid pages, then all these pages are written into the next block $b_{j'}$ (with $j' = i + 2 \bmod n$). This process is repeated until less than m write requests are buffered. This must eventually happen because we have $T > V$. The pseudocode is presented in Algorithm 1. This algorithm is also known as circular buffer scheme and a special case of the algorithm presented in (Hu et al. 2009). Note that buffering the data, but writing m write requests at once is not necessary, but it simplifies the proof. Before we start, we need a small helper lemma.

Algorithm 1 Cycling

```
1: i ← 0
2: for every write request do
3:     while number of outstanding write requests ≥ m do
4:         issue write request for every valid page from block b_i
5:         erase block b_i
6:         write data from write requests into b_i
7:         i ← i + 1 mod n
8:     end while
9: end for
```

Lemma 1. *The write overhead ω is always larger than the ratio between the total number of internal and external write requests, i.e., $\omega \ge \frac{I}{E}$.*

Proof. To see that the inequality holds consider the average local write overhead of an arbitrary page p_ℓ^j of block b_j. We have $E_\ell^j = E/T$ and $I_\ell^j = I/T$ and therefore $\omega_\ell^j = \frac{E_\ell^j+I_\ell^j}{\frac{E}{T}} - 1 = \frac{E/T+I/T}{E/T} - 1 = \frac{I}{E}$. The pigeon hole principle now yields that there must be a physical page that has local write overhead of at least $\frac{I}{E}$. There can be a page with even larger write overhead. □

Theorem 2. *The cycling algorithm guarantees $\sigma\omega \le 1$.*

Proof. To see this, consider one run from "left to right", i.e., from block b_0 to block b_{n-1}. We briefly show that no logical address can cause two internal write requests. Consider a fixed logical address. Wlog let b_i be the block that is erased and in which the corresponding data currently is stored in. Since it is valid, this causes an internal write request. Now, the write request for this logical address is in the queue and later written to some block b_j with $j > i$. The next write requests are written into block b_{j+1}. Thus, no second write request for the data of the logical address is issued in this run. In the beginning, there are V valid pages and thus $T-V$ free or invalid pages. While writing T pages, we have to issue up to V internal write requests to avoid data loss. Thus, in one cycle the T write requests are made up by V internal and E external write requests. Since exactly the same number of write requests is written into each block, the average local write overhead is identical to the write overhead. Thus, we obtain $\omega = \frac{I/n+E/n}{E/n} - 1 = \frac{(V)/n+(T-V)/n}{(T-V)/n} - 1 = \frac{V}{T-V} = \frac{V}{V(\alpha-1)} = \frac{1}{\alpha-1} = \frac{1}{\sigma}$. □

5. Lower Bound

We now show that there is no algorithm that can be better than our simple algorithm in the worst case. But before we continue, let us introduce a few terms. We define d to be $\frac{1}{\alpha} \cdot m$. Let a *sparse* block denote a block with less than d valid pages, and let a *dense* block denote a block with at least d valid pages. To improve readability, we assume wlog that d is an integer. Note that each block can be dense, i.e., contain at least d valid pages because we have up to V valid pages in total. To improve readability, we assume that there are m additional valid pages stored on the flash memory. Thus, even when the algorithm has buffered m write requests, every block can still be dense.

Theorem 3. *No algorithm can guarantee $\sigma\omega < 1$.*

Proof. We will first make a few simplifying assumptions that will be lifted later on. We assume that every block contains exactly the same number of valid pages, i.e., $d = \frac{1}{\alpha} \cdot m$, in the beginning and we assume that every algorithm writes only once its buffer is full. Furthermore, we do not allow static wear leveling, i.e., moving valid pages from one block to another without an exogen write request.

The main idea of this proof is that the adversary always invalidates pages in such a way that every block remains dense. Thus, only a "few" external write requests can be written into a block until there are no free pages left in a block. Because the block is still dense, "a lot" of internal write requests have to be issued to avoid data loss. Thus, the write overhead is "high" and therefore also the product of space and write overhead.

Recall that I denotes the total number of internal write requests and E denotes the total number of external write requests. As shown in Lemma 1, we have $\omega \geq \frac{I}{E}$. Thus, it suffices to show that $\frac{I}{E} \geq \frac{1}{\sigma}$ holds. We do this by carefully accounting for the number of external and internal write requests per block and showing that no block achieves a better ratio.

When any algorithm writes the m write requests into a block, it has to issue d internal write requests. Note that this means that the buffer of the algorithm is not empty, but contains d write requests. From now on, it can accept $m - d$ external write requests until its buffer is filled and it writes m pages. Thus, the write overhead is $\frac{d}{m-d} = \frac{\frac{1}{\alpha}m}{m-\frac{1}{\alpha}m} = \frac{\frac{1}{\alpha}}{1-\frac{1}{\alpha}} = \frac{1}{\alpha-1} = \frac{1}{\sigma}$. This is equivalent to $\sigma\omega = 1$.

We start by lifting the assumption that any algorithm writes data only when its buffer is full. The proof above can easily be adapted such that the adversary no longer invalidates pages from the one dense block, but from any dense block. Note that such a block must always exist since the average number of valid pages in a block is d. If we now consider the write overhead of each block separately, it is easy to see that the same approach works with more fine grained writing/invalidating. Every block can accept at most $m - d$ external write requests and issues d additional internal write requests when it is erased. Thus, we obtain $\omega = d/(m - d) = \frac{1}{\sigma}$.

Next, we allow the data to be unevenly distributed on the flash memory in the beginning. Note that once again the adversary only invalidates pages of dense blocks. This page now needs to be written into a new block. Let b_i be the block selected by the algorithm. If b_i is dense, then the algorithm can only write $m - d$ pages into it before it is full. If b_i is now erased, there were at most $m - d$ external write requests but m write requests in total, thus the write overhead is $d/(m - d) = \frac{1}{\sigma}$. Hence, let b_i be a sparse block. The algorithm can write up to d pages per sparse block and thus up to $n \cdot d$ pages in total in sparse blocks until before there is no sparse block left. Note that the adversary will not invalidate a page of a sparse block and thus will not create a sparse block. Hence, at most V external write requests can be written and therefore it does not affect the asymptotic behavior.

We continue by showing how to lift the assumption that the algorithm does not perform static wear leveling. We start with the simple case that an algorithm erases blocks while they still have valid pages. If algorithm $\mathcal{A}$ chooses to erase this block prematurely, then there were at most $m-d-1$ external write requests written to this block. Since the adversary ensures that every block is dense, there are d internal write requests. Thus, the write overhead is at least $d/(m - d - 1) > d/(m - d)$ and therefore $\sigma\omega > 1$.

We now focus on more interesting static wear leveling, i.e., valid pages being moved around blocks. We proof our result by carefully accounting for the external and internal write requests. Let k be the number of pages that have been moved away from block b_i before block b_i needs to be erased. Note that if this block is not erased, then it is clear that issuing the k internal write requests only increases the write overhead. Thus, we can assume that block b_i is erased later on. Note that valid pages need to be preserved and thus internal write requests need to be issued. If pages from a dense block are moved and the block remains dense, then it is easy to see that later, when this block is erased, at most $m - d$ external write requests could have been written into this block and d internal write requests are issued. Thus, we obtain the familiar expression $d/(m - d)$ for the write overhead. The same argument holds if the page becomes dense again before it is being erased. Hence, let $s < d$ be the number of valid pages right after it has been erased the next time. We proceed by showing that between now and the last time this block was erased, its write overhead is $\frac{1}{\sigma}$. Since the block started with d valid pages, it could only accept $m - d$ external write requests. Because of the static wear leveling at least $d - s$ internal write requests were issued. To preserve the data another s internal write requests were issued. Combined this yields $((d - s) + s)/(m - d) = d/(m - d)$. Hence, static wear leveling has not decreased

the write overhead. But now the block contains only s valid pages. We will now proceed to show that the result of it also does not decrease the write overhead.

We now show that between now and until the next time block b_i is erased, the write overhead is at least $\alpha/(\alpha - 1)$. Keep in mind that the adversary will continue to only invalidate pages from dense blocks. Thus, this block will inevitably become dense. We consider the $d - s$ pages that were moved and denote them *emigrant* pages. Until the next time this block is erased, there can be $m - s$ many external write requests and there are at least d many internal write requests. In addition to this, consider the blocks storing an emigrant page. It is easy to see that each of these blocks now can take one less external write request before being erased. We account these to block b_i. Thus, there are $m - s - (d - s) = m - d$ many external write requests. The number of internal write requests is d (simply to preserve the valid pages). Thus, we obtain $\omega = d/(m - d)$. □

5.1 Total Number of Pages Written

Note that the blocks of a flash memory need to be used evenly to maximize the life of the flash memory. In a perfect scenario flash memory with T pages and each block being able to withstand H writes, then this flash memory can endure up to $H \cdot T$ pages being written before its end of life. Due to the fact that not every block is worn equally and write overhead, this cannot be achieved.

It is easy to see that the cycling algorithm wears the blocks out evenly. Hence, the write overhead is the only criteria, which determines the how many write requests can be written. Since its write overhead is optimal, the cycling algorithm can write a $\frac{T \cdot H}{\omega+1}$ pages. This leads to the following corollary.

Corollary 4. *The cycling algorithm maximizes the amount of data written on the flash memory in the worst case.*

6. Different Access Pattern

Until now we have assumed that an adversary chooses which page is invalidated and we have given the adversary complete knowledge about the employed wear leveling algorithm. This is a rather pessimistic point of view. Thus, we now assume a simple access pattern and analyze the performance of the algorithm described above. It is easy to see that if data is simply written sequentially and the "oldest" data is always being invalidated, then the cycling algorithm is optimal and has no write overhead (independent of the space overhead). Hence, the effect of the access pattern should not be neglected.

6.1 Uniform Write Access Pattern

Let us describe a more realistic write access pattern, albeit a very simple one: the logical address of a write request is chosen uniformly at random. We say that a write request is random when its logical address is chosen uniformly at random. Let us give an intuition why this write access pattern should positively influence the write overhead. The worst case analysis assumed that while cycling from block b_0 to b_{n-1} the V internal write requests were issued. But if the write requests are uniform at random, some pages will inevitably be invalidated and thus it is not necessary to issued an internal write request for these.

Lemma 5. *The cycling algorithm has an expected write overhead of at most $\frac{e}{\alpha e^{\alpha} - e}$ if the write requests are random.*

Proof. We first calculate the probability that a page in a block that is currently being written into is still valid when the algorithm has cycled through the flash memory once and then use this to calculate the expected write overhead.

Consider a fixed physical page. We use the same argumentation as in the proof of Theorem 2 to show that there are at least $T - V$ external write requests until the cycling algorithm writes again in this block. The probability that the logical address associated with this physical page is invalidated by one external write request is $\frac{1}{V}$. Thus, the probability that it is not invalidated and still valid after $T-V$ external write requests is at most $\left(1 - \frac{1}{V}\right)^{T-V} = \left(1 - \frac{1}{V}\right)^{V(\alpha-1)} \approx e^{-(\alpha-1)}$. We conclude that the expected number of internal write requests is at most $e^{-(\alpha-1)} \cdot V$ while writing V pages. Note that it suffices to consider V pages because we only need to look at the V logical addresses that are valid. Thus, we obtain $\omega \leq \frac{e^{-(\alpha-1)} \cdot V}{T - e^{-(\alpha-1)} \cdot V} = \frac{e}{\alpha e^{\alpha} - e}$. □

Note that $\sigma\omega \leq \sigma \frac{e}{\alpha e^{\alpha} - e} = \sigma \frac{e}{e(\alpha e^{\alpha-1} - 1)} = \frac{\sigma}{(\sigma+1)e^{\sigma} - 1} = \frac{\sigma}{\sigma e^{\sigma} + e^{\sigma} - 1} \leq \frac{1}{e^{\sigma}} < 1$.

In order to make our access pattern more realistic, we divide the data into two types: static and dynamic. Static data is data that is never invalidated whereas dynamic data is data that is invalidated uniformly at random. Let μ, δ, and φ be the fraction of static data, dynamic data, and free space, respectively. Thus, we have $\mu + \delta + \varphi = 1$.

Theorem 6. *The cycling algorithm has write overhead $\omega \leq \frac{\mu \cdot T + \delta \cdot T \cdot e^{\frac{1-\alpha}{\alpha\delta}}}{T - (\mu \cdot T + \delta \cdot T \cdot e^{\frac{1-\alpha}{\alpha\delta}})}$.*

Proof. It is easy to see that while cycling from "left" to "right", i.e., from block b_0 to block b_{n-1}, every page containing static data is still valid and thus causes an internal write request to avoid data loss. The dynamic data can be analyzed as before. Consider a fixed valid physical page containing dynamic data. The probability that one external write request invalidates this page is $\frac{1}{\delta \cdot T}$. While cycling from left to right, we can lower bound the number of external write requests by $T - V$. Thus, the probability that this page is not invalidated while cycling once from left to right is at most $(1 - \frac{1}{\delta \cdot T})^{T-V} \approx e^{\frac{1-\alpha}{\alpha\delta}}$. Hence, the number of internal write requests from dynamic data is $\delta \cdot T \cdot e^{\frac{1-\alpha}{\alpha\delta}}$ while cycling through the flash memory once. This leads to an expected write overhead of at most $\frac{\mu \cdot T + \delta \cdot T \cdot e^{\frac{1-\alpha}{\alpha\delta}}}{T - (\mu \cdot T + \delta \cdot T \cdot e^{\frac{1-\alpha}{\alpha\delta}})}$ □

References

R. Agarwal and M. Marrow. A closed-form expression for write amplification in NAND Flash. In *GLOBECOM Workshops*, pages 1846–1850, 2010.

N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference on Annual Technical Conference*, ATC, pages 57–70, 2008.

A. Ben-Aroya and S. Toledo. Competitive Analysis of Flash Memory Algorithms. *ACM Trans. Algorithms*, 7(2):23:1–23:37, Mar. 2011.

L.-P. Chang. On Efficient Wear Leveling for Large-scale Flash-memory Storage Systems. In *Proceedings of the ACM Symposium on Applied Computing*, SAC, pages 1126–1130, 2007.

L.-P. Chang, T.-Y. Chou, and L.-C. Huang. An Adaptive, Low-cost Wear-leveling Algorithm for Multichannel Solid-state Disks. *ACM Trans. Embed. Comput. Syst.*, 13(3):55:1–55:26, Dec. 2013.

L.-P. Chang and C.-D. Du. Design and Implementation of an Efficient Wear-leveling Algorithm for Solid-state-disk Microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.*, 15(1):6:1–6:36, Dec. 2009.

L.-P. Chang and L.-C. Huang. A Low-cost Wear-leveling Algorithm for Block-mapping Solid-state Disks. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES, pages 31–40, 2011.

T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, EUC, pages 394–404, 2006.

P. Desnoyers. Analytic Models of SSD Write Performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

T. Frankie, G. Hughes, and K. Kreutz-Delgado. A Mathematical Model of the Trim Command in NAND-flash SSDs. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE, pages 59–64, 2012.

X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write Amplification Analysis in Flash-based Solid State Drives. In *Proceedings of of SYSTOR*, SYSTOR, pages 10:1–10:9, 2009.

Y. Kim, B. Tauras, A. Gupta, D. Mihai, and N. B. Urgaonkar. FlashSim: A Simulator for NAND Flash-based Solid-State Drives, 2009.

J. Lee, E. Byun, H. Park, J. Choi, D. Lee, and S. H. Noh. CPS-SIM: Configurable and Accurate Clock Precision Solid State Drive Simulator. In *Proceedings of the ACM Symposium on Applied Computing*, SAC, pages 318–325, 2009.

S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

Micron. Wear-Leveling Techniques in NAND Flash Devices, 2008.

Micron. NAND Flash 101: An Introduction to NAND FLash and How to Design It In to Your Next Product, 2010.

Micron. Wear-Leveling in Micron NAND Flash Memory, 2011.

M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of theAnnual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 14–23, 2009.

D. Woodhouse. JFFS : The Journalling Flash File System, 2001.

L. Xiang and B. M. Kurkoski. An Improved Analytical Expression for Write Amplification in NAND Flash. *CoRR*, 2011.