Improving Flash Write Performance by Using Update Frequency

Radu Stoica
École Polytechnique Fédérale de Lausanne
radu.stoica@epfl.ch

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
anastasia.ailamaki@epfl.ch

ABSTRACT

Solid-state drives (SSDs) are quickly becoming the default storage medium as the cost of NAND flash memory continues to drop. However, flash memory introduces new challenges, as data cannot be efficiently updated in-place. To overcome the technology's limitations, SSDs incorporate a software Flash Translation Layer (FTL) that implements out-of-place updates, typically by storing data in a log-structured fashion. Despite a large number of existing FTL algorithms, SSD performance, predictability, and lifetime remain an issue, especially for the write-intensive workloads specific to database applications.

In this paper, we show how to design FTLs that are more efficient by using the I/O write skew to guide data placement on flash memory. We model the relationship between data placement and write performance for basic I/O write patterns and detail the most important concepts of writing to flash memory: i) the trade-off between the extra capacity available and write overhead, ii) the benefit of adapting data placement to write skew, iii) the impact of the cleaning policy, and iv) how to estimate the best achievable write performance for a given I/O workload. Based on the findings of the theoretical model, we propose a new principled data placement algorithm that can be incorporated into existing FTLs. We show the benefits of our data placement algorithm when running micro-benchmarks and real database I/O traces: our data placement algorithm reduces write overhead by 20% - 75% when compared to state-of-art techniques.

1 Introduction

Solid State Drives (SSD) are becoming the default storage medium in enterprise applications such as for database workloads. However, the write behavior of the underlying NAND flash technology is problematic. Flash memory cannot be written directly (updated in-place) but needs an erase operation prior to a program operation (erase-then-write). While flash pages can be read and programmed individually, erasures are particularly time consuming (Table 1) and affect large blocks of pages (Table 2). In addition, due to the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 9
Copyright 2013 VLDB Endowment 2150-8097/13/07... $ 10.00.

NAND Flash

Table 1: Raw operations

Page Read	$80\mu s$
Page Program	$200\mu s$
Block Erase	1.5ms
Max. Erasures	5×10^3-10^5

Table 2: Organization

Page size	512B – 16kB
Block size	32kB – 512kB
Pages per block	64 – 128

elevated voltage required by an erase, a block can withstand only a limited number of erase cycles before the accumulated physical damage renders it unusable. The slow program and erase operations result in low NAND flash write performance alongside with limited device lifetime. To make things worse, the hardware trends are discouraging [8]: write and erase latencies of future NAND chips are expected to increase, while erase endurance is expected to decrease.

The alternative to performing in-place updates is to use an extra indirection layer, known in the literature, as the Flash Translation Layer (FTL) that performs out-of-place writes and deals transparently with the technology's constraints. The FTL redirects incoming writes to unoccupied and already erased locations, while old data versions are logically marked invalid and cleaned later, i.e. garbage collected.

The key challenge in implementing an efficient FTL is to control data placement so that invalid pages can be reclaimed in bulk. At one extreme, if a block contains only invalid data, it can be erased immediately and the erase latency is amortized over n future page writes. At the other extreme, if a block contains a single invalid page, we need to read out $n-1$ valid pages, write them back to a different location, and finally perform the erase – all to allow for a single page write. Thus, write performance and device wear can fluctuate by two orders of magnitude as a function of the data placement and the cleaning policy.

Unfortunately, due to the large size of an erase block, valid and invalid data often end up collocated resulting in expensive data moves needed to reclaim space. Experience shows that an FTL is notoriously difficult to "get right" because of to the complex dependency of flash cleaning overhead on: i) I/O workload, and ii) a multitude of FTL design choices such as the amount of space over-provisioning, data placement algorithm, and cleaning policy. In spite of a large body of existing research work, the end result is a low and unpredictable SSD write performance [2]. This stands especially for database I/O intensive workloads that generate small scattered writes.

In this paper, we give a principled answer the question: *How much can we reduce the writing overhead of an FTL by leveraging write skew to guide data placement?* We model the cleaning overhead analytically for a set of basic I/O write patterns (*k-modal* update distributions), data placement policies, and cleaning strategies. Based on the analytical results, we propose a new data placement algorithm that utilizes update frequency to reduce FTL cleaning overhead, and thus improves write performance and device lifetime.

This paper makes the following *contributions*:

1. We model the impact of space over-provisioning, write frequency, data placement policy, and garbage collection strategy on FTL cleaning overhead. The model is useful for validating and guiding the design of FTL data placement algorithms. We verify the analytical results through detailed simulation.
2. We approximate the minimum cleaning overhead achievable for any given I/O workload within an error margin, which allows us to quantify the headroom for reducing the cleaning overhead of any FTL.
3. Based on the modeling results, we propose a data placement algorithm that exploits update frequency in a principled way and is capable to adapt dynamically to the I/O write pattern, without relying on workload-specific parameters. The new data placement algorithm can be integrated into existing hybrid or page-level mapping FTLs (as we discuss in Section 2.4). Experimental results show that our algorithm reduces cleaning overhead by 20%-75% compared to state-of-art techniques across a variety of microbenchmarks and I/O traces collected from standard DBMS benchmarks such as TPC-C [24] or TATP [19].

The paper is organized as follows: In Section 2 we present the challenges of storing data on flash and existing state-of-art data placement algorithms. In Section 3 we explain the FTL concepts and notations used throughout the paper. In Section 4 we model analytically the cleaning overhead for k-modal update distributions and highlight the important lessons found. In Section 5, based on the analytical results, we propose a new data placement algorithm and in Section 6 compare our proposal with existing state-of-art algorithms. Finally, we conclude in Section 7.

2 Related Work

In this section, we highlight the evolution of data placement algorithms for out-of-place updates and give insight on why a novel approach can lower FTL cleaning overhead.

2.1 State-of-art Data Placement Algorithms

The concept of out-of-place updates on NAND flash memory is simple at heart: instead of performing the costly erase-then-write cycle on a large erase block, introduce an extra indirection layer between the logical address written and the physical location (erase block and page number) where the user data is stored. At an update request, the FTL can redirect the write to an unoccupied and already erased location; the old version of the data is logically marked invalid and garbage-collected later. No in-place updates are needed, and, in the most favorable scenario, only one erase is required for writing a full block worth of new data.

The overhead of reclaiming space, i.e. the number of pages in a block we need to move before an erase, dictates the efficiency of out-of-place writes. It is obvious that a good data placement algorithm groups together pages with similar update frequencies. Otherwise, if cold and hot data are collocated, the hot pages are invalidated quickly, while the cold pages are left behind acting as dead-weight and need to be relocated when the block is cleaned. Virtually all FTL data placement proposals try to group data based on its update frequency.

Rosenblum and Ousterhout first encountered the data placement question for out-of-place writes in the context of the LFS log-structured file-system [23]. In LFS, small scattered writes are stored sequentially on magnetic disks, resulting in a high write throughput, while creating the need to reclaim the invalid data left behind. The authors first noticed that storing together data with similar update frequencies is key to minimizing cleaning overhead and introduced a cleaning algorithm that tries to categorize data as hot/cold and balance the amount of free space reclaimed and data age.

The initial ideas of LFS were refined over time and adapted to flash-based systems. Kawaguchi et al. [14] tuned the LFS cost-benefit heuristic to match NAND flash behavior, and eNVy [26], the first FTL-like proposal, further improved on the LFS data placement concepts. eNVy splits space into fix-sized partitions (log structures in reality) composed of segments (erase blocks), while pages migrate between partitions in a way that encourages locality of reference. eNVy aims to equalize the cleaning cost of partitions, defined as the product of the update frequency and the overhead of cleaning a partition.

Chiang et al. proposed Dynamic Age Clustering (DAC) [5], which, similarly to eNVy, shares the overall design of partitioning to space in regions but, unlike eNVy, uses different page migration policies between regions. Pages are promoted to a colder region when garbage collected, and to a hotter region when updated and if their age (measured in seconds) is smaller than a workload dependent threshold.

More recently, Hu et al. proposed ContainerMarking [12], a data placement technique with an overall design similar to eNVy and DAC. Space is partitioned in more fine-grained regions (also behaving as log structures) and data migrates between neighboring regions to encourage separation based on update frequency. Pages are promoted to a hotter partition at every update, and demoted to a colder partition at garbage collection based on a statistical model. The key difference over previous approaches is to take into account the occupancy of the log structures and decreases demotion rate when occupancy is high.

2.2 Data Placement Challenge

There are two main challenges when moving data between two regions. Firstly, there is a *highly non-linear relationship* between the occupancy of a region and its cleaning cost. For example, in Section 4.1, we show that the cleaning cost increases faster than exponentially with occupancy for random updates. Secondly, the increase/decrease in the cleaning cost of the regions cannot be assessed immediately but is experienced at a *much later time*. When pages migrate between two regions, the valid page counts of the oldest blocks – the blocks first in line to be cleaned – remain the same and thus the cleaning cost appear initially unaffected by the data movement. The full effect of the data movement is noticed only after writing all the blocks of a region once.

For example, consider two partitions, the first one with a disproportionately high occupancy and cleaning cost compared to the second. It is clear we need to move some data from the first partition to the second one; however, it is not clear how much or which data to move. A static data promotion policy (e.g. trying to equalize cleaning cost between the two partitions or a simple always promote/demote strategy) results in data movement swings triggered by changes in region occupancy rather than changes in the workload skew. Pages initially move in one direction from the first partition to the second with a lower cost cleaning cost. At some point, the cleaning frequency and/or cleaning costs become comparable and the page migration stops. Unfortunately, by this time, too many pages have been already moved and the cleaning cost of the second partition continues to increases. Finally, the migration process reverts in the opposite direction.

In this paper, we address the limitations of the previous data placement approaches through a principled understanding of the relationship between the cleaning cost and data placement decisions. We model the cleaning cost analytically for sufficiently general update distributions that allows us to reason about the fundamental cleaning overhead of any given I/O workload. Based on the findings of the analytical model, we then propose a principled data placement algorithm that is close to optimal within a margin of error. Our data placement algorithm distinguishes itself from previous proposals in the following ways:

- No tunable parameters. Previous proposals rely on variable parameters that are workload specific. Examples include the number of regions [12, 5, 26], the time threshold to migrate data from one region to the next [5], the empirical probabilistic model of [12].
- No mismatch between region size and the page update frequency. Previous approaches suggest fixed sized regions that possible lead to collocating pages with very different update frequencies; our data proposal determines the optimal number of regions (i.e. log structures) and their optimal size at runtime.
- Accurate page promotion/demotion. A strawman strategy that always promotes pages to a hotter region when updated and demotes them to a colder region when cleaned is known to result in suboptimal performance [12]. When cleaning, we can differentiate if a page is cleaned due to the region occupancy or because it truly has a lower update frequency. When updating, we can identify the precise threshold when page is hot enough to justify promoting it to a hotter region.

These keys differences enable our data placement algorithm to achieve a much lower cleaning overhead than the state-of-art techniques. We show in Section 6 our data placement proposal reduces cleaning overhead by 20-75% for DBMS I/O traces of TPC-C and TATP workloads.

2.3 Other Modeling Approaches

Hu et al.[11] introduce a write performance model for random updates and consider various cleaning policies such as LRU (FIFO), Greedy, and Window-Greedy. However, as the analytic model has certain weaknesses at low over-provisioning values, a second empirical model is fitted directly from simulation results. We propose a different modeling approach for understanding cleaning performance with an almost perfect accordance with experimental results. Most importantly, our model is capable of representing much more general update distributions (k-modal update distributions).

Bux and Iliadis [3] modeled the performance of the Greedy garbage collection policy for random updates. The authors propose two different techniques: a precise Markov chain model applicable to small systems and a statistical analysis of larger systems. The Markov chain model, different from our proposal, introduces a state for every flash block, and is only usable for modeling systems with a few flash blocks. The statistical analysis yields the most important result, which quantifies the limit of the probability to relocate a page as the number of pages per block increase, although no closed formula is provided. We provide a different technique to understand the behavior of the Greedy garbage collection policy, that also applies to any Window-Greedy policy, and identify an analytic formula to Bux and Iliadis' limit, as discussed in Section 4.1.

Table 3: Modeling notations

Symbol	Signification
N	The total number of logical user pages
C	The number of pages in an erase block
α	Over-provisioning: $\alpha = \frac{ExtraCapacity}{UserCapacity}$
p_{gc}	The probability a page is valid when cleaning its block and has to be relocated
GC	Cleaning cost: $\mathrm{GC} = \frac{\mathrm{p_{gc}}}{1-\mathrm{p_{gc}}}$
f_i	The update frequency of a page in set i (e.g. $f_i = 1/N$ for random updates)
s_i	The size of set i, i.e. the fraction of pages updated with frequency f_i: $\sum_1^k s_i = 1$
f_{si}	The fraction of the total updates experienced by set i: $f_{si} = \frac{f_i s_i}{\sum_{j=1}^k f_j s_j}$, and $\sum_{i=1}^k f_{si} = 1$.

2.4 Applicability

FTLs are usually categorized based on the unit size of the logical to physical mapping: page level mapping FTLs [26, 5, 10, 12, 18], hybrid page-block level FTLs [15, 16]), or variable size mapping FTLs [17, 6].

Recent work (DFTL [10], LazyFTL [18]) shows that page-based FTLs are superior, even if the mapping meta-data has to be paged SSD DRAM and flash storage. A page-level mapping allows maximum flexibility to optimize data placement and does not have the drawback of block merges of hybrid FTLs. We share the view that FTL should use a page-level mapping, especially for enterprise SSDs designed for write intensive scattered I/O workloads.

Our data placement proposal can be incorporated either into page-level FTLs, into a hybrid FTL for the page mapping area, or into a variable-sized FTL for small scattered updates. The page-level mapping area of hybrid FTLs is small (few percentages of total capacity), thus it is crucial to use it optimally – precisely what our proposal does best.

3 Notations and Assumptions

In the rest of the section, we introduce the parameters and metrics used to model and quantify the FTL writing overhead. The notations are summarized in Table 3.

FTL metrics: The concept of extra FTL capacity is known in the literature under two names, over-provisioning ($\boldsymbol{\alpha}$) or device utilization ($\boldsymbol{\mu}$). Over-provisioning represents the ratio between extra capacity and user capacity, while utilization is the ratio between user capacity and total capacity. We further prefer over-provisioning as utilization compresses the extra capacity range of real-life SSDs (typical devices have $10\% - 50\%$ over-provisioning). However, the two concepts are interchangeable as $\mu = \frac{1}{1+\alpha}$.

The overhead of cleaning a block (a.k.a. garbage collecting) is given by the fraction of pages that are valid and need to be relocated before erasing the block. This fraction, denoted as $\mathbf{p_{gc}}$, can be interpreted either as a percentage or as a probability.

The benefit resulting from the erase operation, the amount of free space reclaimed, is the capacity of the block minus the space consumed by the page relocations. Thus, we can define the concept of Write Amplification as the number of extra physical writes the FTL makes for every user write:

$$WA = 1 + \underbrace{\frac{\mathrm{p_{gc}}}{1-\mathrm{p_{gc}}}}_{\text{Cleaning overhead } (\mathbf{GC})} \tag{1}$$

The "**1**" term denotes the physical write needed to store the newly written user data and can only be avoided through compression or deduplication. The second term represents the actual cleaning overhead due to out-of-place writes. We further omit the physical write and model only the cleaning overhead for simplicity.

Both $\mathrm{p_{gc}}$ and GC concepts are useful to characterize flash write performance. From a modeling and reasoning perspective, $\mathrm{p_{gc}}$ is easier to estimate and to reason about. On the other hand, GC represents the end-to-end cleaning overhead. Minimizing GC results both in better write performance and in a longer device lifetime. The frequency of block erasures is directly proportional to the write amplification and GC shows how much faster an SSD ages at every user write.

I/O workloads: Throughout the paper we model variants of **k-modal** I/O workloads. A k-modal I/O workload represents an update distribution where the update frequency of any given page can have only k possible discreet values $(f_i, ..., f_k)$. We call all pages with a given update frequency an *update set*. Each update set has an associated size defined as a fraction of the total user data s_i and receives a fraction f_{si} of the overall updates. A k-modal workload has two important advantages: a) it is simple enough to model analytically; b) it is sufficiently general to approximate a real I/O workload. A general I/O workload can be abstracted through a k-modal workload by "binning" together pages with a similar update frequency. As we will see, pages can be safely grouped in bins with an exponentially increasing update frequency.

3.1 Modeling Assumptions

We make the following assumptions when modeling the cleaning overhead:

1. We exclude sequential writes. Exploiting sequential write patterns is an orthogonal topic and is relatively easy to implement, e.g. by detecting sequential writes patterns and storing updates in the same erase block [15]. If the same pages are written again sequentially, the block is fully invalidated and can be erased without any data moves. Our model supports, however, any skewed I/O workloads.
2. We model only the long-term FTL cleaning overhead. A newly formatted SSD has no immediate cleaning overhead, as all flash blocks are erased and available for writing; this results in good but transient write performance.

4 FTL Cleaning Overhead

In this section, we model the cleaning overhead (GC) as a function of over-provisioning, data placement, and cleaning policy for various update distributions. We first model the cleaning overhead for random updates (a 1-modal workload), then introduce update skew by considering a 2-modal update distribution, and we finally generalize the results for arbitrary skewed update distributions represented by k-modal workloads.

4.1 Random Updates (1-Modal Distribution)

We assume a random update workload composed of single page updates. All pages have the same update probability, therefore data placement does not matter, i.e. there is no benefit in storing any pages together, and the cleaning cost only depends on the block selection policy.

Consider two block cleaning policies introduced initially in LFS [23]: i) a Least Recently Used (LRU) policy where the oldest written block is cleaned first, and ii) a Greedy policy that selects the block with the lowest number of valid pages. As all pages have the same update frequency and as updates are uncorrelated, the Greedy policy is optimal (for a more detailed discussion please see [23] [11]).

We first model GC for the LRU cleaning policy, then show that the additional benefits of the Greedy policy are negligible. By viewing the LRU policy as the "worst case" and the Greedy policy as the "best case" we are bounding the benefits of any intermediary cleaning policy such as Window-Greedy [11]. Showing that the LRU cleaning policy is close to optimal is of practical importance. It is the simplest policy to implement, has the lowest memory and CPU overhead, and has a perfect wear leveling behavior (blocks are erased at the same rate). Comparatively, the Greedy policy is more expensive as it requires a data structure that sorts flash blocks on the number of valid pages contained.

LRU cleaning policy. The flash device can be considered a circular log-structure where updates are appended to the tail and cleaning always starts at the log head with the oldest block written, whenever space is needed. If a page has to be relocated for cleaning, we read the page from the log head and write it to the tail followed by advancing both head and tail pointers.

Over a full log wrap-around, writes are caused either by page relocations or by actual user updates. At every user update, the probability that a given page escapes invalidation is $1 - \frac{1}{N}$, while the cleaning process produces no invalidations (a block is immediately erased once its valid pages are relocated). Thus, the probability that a given page remains valid over a log wrap-around is the probability to escape invalidation at one update at the power of the total number of user updates accommodated over a log wrap-around. Assuming there are $(1+\alpha) \cdot N$ total physical pages, a $\mathrm{p_{gc}}$ fraction are "wasted" due to relocations from the cleaning process, while $(1-\mathrm{p_{gc}})$ physical pages actually store new user data. We can therefore deduce $\mathrm{p_{gc}}$ from the limit of the equation:

$$\mathrm{p_{gc}} = (1 - \frac{1}{N})^{N(1+\alpha)(1-\mathrm{p_{gc}})}$$

We apply Euler's limit as N is sufficiently large (there are $2^{17} - 2^{18}$ pages per GB of NAND flash):

$$\mathrm{p_{gc}} \quad = \quad e^{-(1+\alpha)(1-\mathrm{p_{gc}})} \tag{2}$$

which accepts the analytic solution:

$$\mathrm{p_{gc}} = -\frac{W(-(1+\alpha)e^{-(1+\alpha)})}{1+\alpha} \tag{3}$$

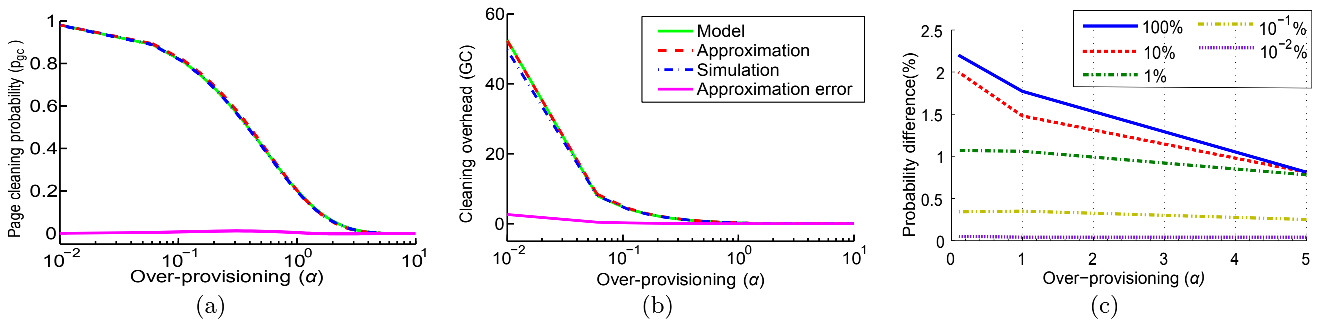

Figure 1: (a) Analytical form, algebraic approximation, and simulation of page cleaning probability $\mathrm{p_{gc}}$; (b) the associated cleaning overhead GC; (*c*) the reduction in page cleaning probability for multiple Window-Greedy policies compared to a LRU policy as a function of over-provisioning and window size (window size expressed as a fraction of the total capacity).

W is the Lambert-W function [7], the solution to the equation $z = W(z)e^{W(z)}$. W cannot be expressed as a composition of algebraic functions. However, W can be computed numerically (e.g. through a Taylor expansion), although this is rather cumbersome from a practical perspective.

To overcome such limitations we propose a simplified approximation. We note that $-(1+\alpha)e^{-(1+\alpha)} \subset (-e^{-1},\ 0)$ for any $\alpha \in \mathbb{R}^+$. Thus, we need to approximate W for a rather small interval. We plot $(1+\alpha)e^{-\alpha}$, $-W(-(1+\alpha)e^{-\alpha})$, and $\mathrm{p_{gc}}(\alpha)$ and observe that the three functions converge to zero for $\alpha > 1$ and are all similar in shape to a negative exponential. We fit the $W(..)$ component of Equation (3) to the $e^{c_1\alpha+c_2}$ family of exponentials and obtain the algebraic approximation:

$$\mathrm{p_{gc}} \approx \frac{e^{-0.9\alpha}}{1+\alpha} \tag{4}$$

Equation (4) gives us a quantitative trade-off between allocating extra capacity and cleaning performance – it shows that $\mathrm{p_{gc}}$ decreases slightly faster than exponentially as over-provisioning increases. We show in the Section 4.2.1 that a similar dependency exists even if we collocate pages with distinct update frequencies.

We validate the model experimentally. In Figure 1(a) we show how the analytic predictions from Equations 3, and 4 compare against each other and against the $\mathrm{p_{gc}}$ found by simulation (the simulation setup is described in Section 5). The difference between any of the three methods is virtually zero for any over-provisioning value (*TotalError* represents the difference between Equation 4 and the simulation results). Figure 1(b) shows the associated cleaning overhead.

Greedy cleaning policy. To quantify the benefit of a Greedy cleaning policy we need to consider more than the average $\mathrm{p_{gc}}$ value as given by Equation 4: we also need to model the distribution of valid pages across the erase blocks.

Consider a generalization of the LRU and Greedy policies: whenever we need to reclaim space, we pick the block with the lowest number of valid pages in the oldest written B blocks. When $B = 1$ the cleaning policy becomes LRU; when $B = (1+\alpha)N/C$ it becomes Greedy. We further explain how to model: i) the $\mathrm{p_{gc}}$ probability distribution function (PDF) as a function of α, ii) the PDF change as a function of the block's position in the cleaning window, and iii) how to compute the benefit obtained by increasing the B window.

i)PDF. The number of valid pages in a block follows a discreet binomial distribution – any two pages in a block have an equal and uncorrelated $\mathrm{p_{gc}}$ probability to be valid. Thus, the probability a block has exactly k valid pages is:

$$p(k) = \binom{C}{k} \cdot {\mathrm{p_{gc}}}^{k} \cdot (1-\mathrm{p_{gc}})^{C-k} \tag{5}$$

Please note that $\mathrm{p_{gc}}(\alpha)$ is given by Equation 4 and the probability distribution function $PDF(\mathrm{p_{gc}})$ is a vector:

$$PDF(\mathrm{p_{gc}}) = [p(C)\ \ p(C-1)\ \cdots\ p(1)\ \ p(0)]$$

and has the associated standard deviation:

$$stddev = \sqrt{C \cdot \mathrm{p_{gc}} \cdot (1-\mathrm{p_{gc}})}$$

ii)PDF variation over the cleaning window. The easiest way of modeling the $\mathrm{p_{gc}}$ PDF at a specific location in the log structure is by using a Markov chain model. The process of invalidating pages can be represented with the following Markov chain transition matrix:

$$M = \begin{bmatrix} \frac{N-C}{N} & \frac{C}{N} & \cdots & 0 \\ 0 & \frac{N-C-1}{N} & \frac{C-1}{N} & \cdots \\ \vdots & \vdots & \ddots & \\ & & \frac{1}{N} & \frac{C-1}{N} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$M(i,i)$ represents the probability that all pages of a block escape invalidation at an update and the block stays in the same state with $C-i$ valid pages. $M(i,i+i)$ represents the probability of migrating to the next state defined by having $C-i-1$ valid pages. The PDF change after n updates is simply $PDF_{new} = PDF_{old} \cdot M^n$. The number of updates n is in turn determined by the position of the block in the cleaning window.

iii)Greedy benefit. We could not deduce an analytic formula to give a concise relationship between $\mathrm{p_{gc}}$ and window size B, however, the $\mathrm{p_{gc}}$ decrease can be computed as:

$$Benefit = \sum_{i=2}^{B} (PDF_{old} - PDF_i) \cdot [C\ C-1\ \cdots\ 1\ 0]'$$

Benefit represents the average number of valid pages saved from relocation by making the cleaning window of size B. Please note all terms are vectors and the last element represents the number of valid pages for the states of the Markov chain model.

The overall findings are that the Window-Greedy policy adds little benefit, as noticed experimentally also by previous work [23] [26] [11]. The gist of why this happens

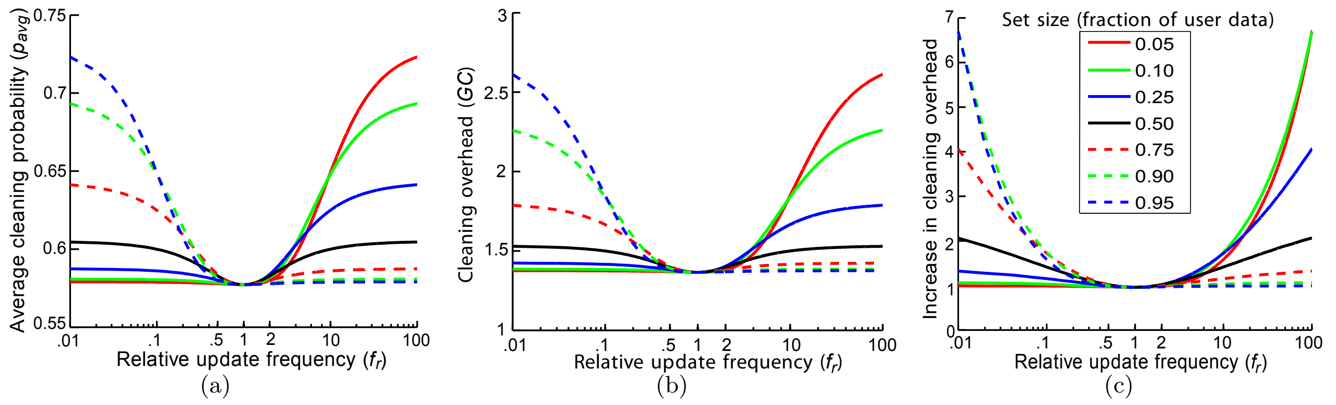

Figure 2: (a) The average page cleaning probability $\mathrm{p_{gc}}$; (b) The corresponding cleaning overhead GC; (c) Relative increase of cleaning overhead due to random placement compared to the minimum overhead possible.

is twofold. First, $\mathrm{p_{gc}}$ increases exponentially as we move through the window B from the oldest written blocks toward the youngest, as can be noticed from Equation 4 when α is decreased. In other words, the window where we can find a block with a lower number of valid pages than in the LRU block is narrow. Second, $PDF(\mathrm{p_{gc}})$ does not have a high variance, meaning that even when blocks with a lower number of valid pages are present, the difference is small.

We also note that Bux and Iliadis' page cleaning probability limit [3] accepts an analytic solution, also in terms of the Lambert-W function, and is precisely Equation 3. This provides additional evidence that the performance of the Greedy policy is equivalent to a LRU policy.

We finally validate the findings by simulation and show in Figure 1(c) the $\mathrm{p_{gc}}$ decrease as we vary over-provisioning and the window size. Overall, $\mathrm{p_{gc}}$ decreases by at most 2%; we therefore conclude that the Greedy or Window-Greedy policies are unnecessary for random updates.

4.2 2-modal Distribution

In this section, we model the cleaning overhead of a 2-modal workload (pages are either hot or cold) in two different scenarios: a) when pages are randomly collocated, and b) when pages are stored separately based on their update frequency.

4.2.1 Random Data Placement

Using the notations from Table 3, the update frequency of a page is either f_1 or f_2, the sizes of the two sets are s_1 and s_2, and let $f_r = f_1/f_2$ be the relative page update frequency between the two update sets. We assume pages are randomly collocated and blocks are cleaned using a LRU policy, i.e. a log-structured data placement.

Let p_{gc1}, p_{gc2} be the probabilities that pages of update set 1 and update set 2 respectively are valid at garbage collection and let p_{avg} be the probability that a page selected at random is valid. p_{gc1}, p_{gc2} can be expressed using a similar logic as in Section 4.1:

$$p_{gc1} = (1 - \frac{1}{s_1 N})^{N \cdot f_{s1} \cdot (1+\alpha) \cdot (1-p_{avg})} \quad (6)$$

$$p_{gc2} = (1 - \frac{1}{s_2 N})^{N \cdot f_{s2} \cdot (1+\alpha) \cdot (1-p_{avg})} \quad (7)$$

After some manipulations (substituting f_{si}, using Euler's limit, applying the logarithm, and dividing the two equations), we finally obtain:

$$\frac{\log(p_{gc1})}{\log(p_{gc2})} = f_r \quad \Leftrightarrow \quad p_{gc1} = p_{gc2}^{f_r} \quad (8)$$

Equation 8 summarizes the relationship between the valid page probabilities of the two sets and shows that the probability of having to relocate colder pages at garbage collection grows towards 1 with the f_r-th root. For example, if set 2 is "cold" then $f_r > 1$ and thus $p_{gc2} = \sqrt[f_r]{p_{gc1}}$.

The relationship between p_{avg} and p_{gc1}, p_{gc2} is deduced from the condition that the total cleaning overhead equals the sum of the partial cleaning overheads associated with each update set. It can be showed that:

$$p_{avg} = 1 - \frac{(1-p_{gc1})(1-p_{gc2})}{f_{s1}(1-p_{gc2}) + f_{s2}(1-p_{gc1})} \quad (9)$$

Equations 6, 7, 9 do not accept a solution in terms of the Lambert-W function as Equation 2 does. However, the probabilities can be determined numerically using an iterative algorithm (e.g. by choosing any initial value for p_{gci} and repeatedly substituting until Equations 6, 7, 9 converge). We validate the analytic predictions through simulation and find a good match: the difference between the numerically computed p_{avg} and simulated p_{avg} is virtually zero for any s_i, f_r, and $\alpha \in (0, 1)$.

Analytic approximation. To ease understanding of the cleaning overhead, we also propose a closed formula approximation for p_{avg}, p_{gc1}, p_{gc2}. The approximation is based on counting the number of pages belonging to each update set. If we take a "snapshot" of all the pages in the log-structure, a $\frac{s_1+s_2}{1+\alpha}$ fraction of the log contains the valid user pages of set 1 and 2. The rest of the pages in the log, representing the $\frac{\alpha}{1+\alpha}$ over-provisioned capacity, contain invalid pages. The rate of creating invalid pages for a set is dictated by its total update frequency. Thus, the over-provisioned capacity can be thought as to be split between the two update sets according to the f_{si} set update frequencies and the page cleaning probabilities can estimated using Equation 3:

$$p_{gci} = -\frac{W(-(1+\alpha_i)e^{-(1+\alpha_i)})}{1+\alpha_i} \quad ; \quad \alpha_i = \frac{f_{si} \cdot \alpha}{s_i}$$

$$p_{avg} = p_{gc1} \cdot \frac{s_1 + f_{s1}\alpha}{1+\alpha} + p_{gc2} \cdot \frac{s_2 + f_{s2}\alpha}{1+\alpha}$$

The analytic approximation tends to slightly underestimate the cleaning overhead; however, it is within 5% the actual cleaning overhead measured by simulation.

Practical implications. In order to interpret the implications of the analytic results, Figure 2(a) shows p_{avg} as a function of the update frequency ratio f_r and set size s_i for an over-provisioning value $\alpha = 0.3$ (middle range for today's SSDs). For example, at value 10 on the x-axis, the page update frequency of the first set is 10 times higher than the update frequency of the second set; a legend value of 0.1 means that the first set contains 10% of the total user data.

As expected, the probability of having to relocate a page when collocating data with different update frequencies is minimized when both update sets have the same update frequency ($f_r = 1$) and then grows to either left or right depending on which update set is dominant (is larger). What is not expected is the large f_r interval where p_{avg} is close to minimum – the update frequencies can differ by $\sim 2\times$ with little penalty. The practical implications are twofold: Firstly, pages should be grouped in "bins" with an exponentially increasing update frequency. Secondly, the frequency range of each "bin" is relatively large as page update frequencies can vary by up to $\pm 50\%$.

We show in Figure 2(b) the corresponding cleaning overhead. The GC shape is similar to p_{avg}, except that the region of minimum penalty tends to be slightly wider. Finally, Figure 2(c) shows the relative GC increase compared to an optimal data placement (as discussed in the next section) and represents the benefit of skew-aware data placement. The relative GC shows how many times the cleaning overhead increases if data is naively collocated compared to separating pages based on update frequency. Overall, p_{avg}, GC, and the relative GC increase are all minimized for $f_r \in (0.5, 2)$.

4.2.2 Frequency-Based Data Placement

Intuitively, the cleaning overhead can be reduced by separating pages based on their update frequency, as was also observed by previous work [22], [23], [26], [12]. In Section 4.2.1 we identified the point where the relative difference between the hot and cold data justifies separating pages based on their update frequency. We show next how to optimally separate data and calculate the minimum cleaning overhead.

Assume the same 2-modal update distribution as in Section 4.2.1, and following the same notations. The only difference in this scenario is that the two update sets are stored in two distinct regions cleaned using a LRU cleaning policy. We assume to have perfect knowledge of the update frequency of each page, and therefore no pages are misplaced.

Let $\beta \cdot N$ ($\beta < \alpha$) be the number of over-provisioned pages allocated to the first log structure. The second log structure receives the rest of the capacity budget of $(\alpha-\beta)\cdot N$ pages. The new p_{gc1}, p_{gc2} page cleaning probabilities can be calculated using Equation (4) as pages have a single update frequency inside each log. The global cleaning overhead, GC_{avg}, is the sum of the individual cleaning overhead for each region weighted by the update frequency of the regions:

$$\mathrm{GC}_{tot} \quad = \quad f_{s1}\frac{p_{gc1}}{1-p_{gc1}} + f_{s2}\frac{p_{gc2}}{1-p_{gc2}} \tag{10}$$

As p_{gc1}, p_{gc2} depend on the allocation of extra capacity, the remaining challenge is how to optimally distribute the extra capacity budget to each update set, i.e. how to select β in order to minimize GC_{tot}.

By differentiating Equation 10 we find β where $\mathrm{GC}(\beta)$ has the global minimum ($\mathrm{GC}(\beta)$ is a convex function). After substituting Equation 3 in 10, we express the derivative as:

$$\frac{\partial \mathrm{GC}_{tot}}{\partial \beta} = \frac{f_{s1}W_1}{s_1(W_1+1)(W_1+z_1)} - \frac{f_{s2}W_2}{s_2(W_2+1)(W_2+z_2)} \tag{11}$$

using the simplifying notations:

$$z_1 = 1 + \frac{\beta}{s_1} \;;\; z_2 = 1 + \frac{\alpha-\beta}{s_2} \;;\; W_i = W(-z_i e^{-z_i})$$

The optimal β is the point where the derivative is null (i.e. $\frac{\partial \mathrm{GC}_{tot}}{\partial \beta} = 0$), which holds when:

$$f_r\frac{W_1}{(W_1+1)(W_1+z_1)} \quad = \quad \frac{W_2}{(W_2+1)(W_2+z_2)} \tag{12}$$

Equation 12 has no analytic solution but we can, however, solve it numerically and deduce its practical implications. Three parameters influence the optimal β value: the amount of over-provisioning (α), the relative update frequency ratio (f_r), and the sizes of the two update sets (s_i).

Figure 3(a) shows the optimal space allocation as a function of s_1 (s_1 determines s_2) and f_r for an over-provisioning value of 0.3. The optimal space allocation is strongly correlated with s_i, while f_r has a smaller influence on the optimal over-provisioning value. Figure 3(a) suggests a logarithmic-like contribution of f_r to the optimal β value.

In Figure 3(b) we see the corresponding cleaning overhead. The shape of GC_{tot} is mirrored by a plane passing through $f_r = 1$ and parallel to the z-axis – the notations of the two update sets can be interchanged (i.e. by "renaming" update set 1 to set 2 and vice-versa). GC_{tot} varies by an order of magnitude with both s_i and f_r and has a maximum when $f_r = 1$. This implies that random updates impose the largest writing overhead once pages are separated according to update skew. The lowest cleaning overhead is seen close to the points $(0.05, 1)$, $(100, 0.05)$, where a small fraction of pages sees a large number of the total updates (heavy skew).

Finally, Figure 3(c) shows the ratio between the cleaning overhead associated to randomly collocating pages and the cleaning overhead achieved by separating pages based on update frequency – i.e. it shows how many times GC is reduced. The decrease in GC is also symmetric, similar to the inverted surface in Figure 3(b) but with higher peaks. The highest benefit of separating data is, as expected, in case of heavy skew. As skew increases, i.e. as f_r grows for fixed α, s_i values, the cleaning overhead of random collocation increases approximately with the logarithm of f_r, while the cleaning overhead achieved by separating pages decreases also approximately with the logarithm of f_r. This two trends explain the position and height of the peaks shown in Figure 3(c). Fortunately, database workloads exhibit a significant amount of skew, as can be seen also in the I/O traces presented in Section 6.

Practical space allocation. As mentioned, solving Equation 12 numerically requires an iterative optimization algorithm (e.g. Newton's method) to find β, while each iteration of the optimization algorithm computes the Lambert-W function in multiple points. Such a process might be too computational intensive to be used online in a FTL. In practice we want to find the solution to Equation 12 using a small bounded number of elementary operations.

We propose to avoid solving Equation 12 and use instead an approximation of the solution $\widetilde{\beta} \simeq \beta_{opt}$. We computed numerically β_{opt} and tried to fit the results to various families of three dimensional functions such as combinations of

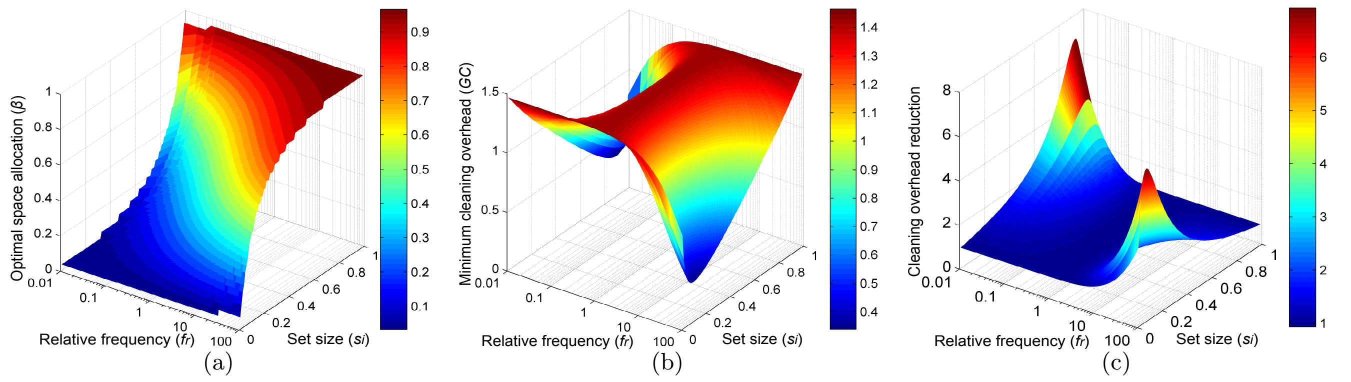

Figure 3: (a) The optimal allocation of over-provisioned capacity β; (b) The associated cleaning overhead GC; (c) The reduction in cleaning overhead possible when separating pages according to update frequency (i.e. GC random placement/GC frequency-based placement).

polynomials, logarithms, exponentials, etc. We could not find a satisfactory fit, therefore we decided to reduce the number of dimensions to two, s_i and α, and consider f_r constant. As mentioned, f_r can be approximated as constant if pages are grouped in exponential decreasing frequency bins (e.g. $f_r = 2, 4, ..., 2^n$). Moreover, if the f_r ratio between two sets of pages is constant, a single approximation for a fixed f_r value suffices.

Once the number of dimensions is reduced, β_{opt} can be approximated using a polynomial function that is both feasible to compute at run-time and also offers a good fit (error less than 0.1% and root mean square error of 0.001 for any values of α, s_i). To balance complexity and precision, we suggest a 1-degree polynomial for the α dimension and 3-degree polynomial for the s_i dimension, although higher degrees reduce further the approximation error:

$$\widetilde{\beta}(s_i, \alpha) \;=\; c_{00} + c_{10}\alpha + c_{01}s_i + c_{11}\alpha s_i + c_{02}s_i^2 + c_{12}\alpha s_i^2 + c_{03}s_i^3$$

Optimal cleaning policy. Equation 11 can also be used to implement an efficient cleaning policy. Whenever we need to select a victim block for cleaning and have a choice between two sets of pages, it is sufficient to know whether the derivative of GC_{tot} is positive or negative. For example, if the right-hand side of Equation 12 is smaller than the left-hand side, then we select a block from the first log structure and vice versa. Such a policy converges to the optimal allocation of space and minimizes cleaning overhead. Equally important, the policy does not depend on monitoring past cleaning overhead; rather, the analytic equations predict and minimize future page relocations. Please note that the Lambert-W values from Equation 12 can be approximated as in Section 4.1 by using $-W(-ze^{-z}) \simeq e^{-0.9(z-1)}$.

4.3 k-modal Distribution

In this section, we first show how to extend the modeling results from a *2-modal* to a *k-modal* update distribution, then explain how to approximate a general I/O workload through a k-modal distribution and analytically compute the minimum cleaning overhead achievable. We assume the same type of workload and notations as in Section 4.2, the only difference being that the number of update sets increases to k.

4.3.1 Random Data Placement

Consider the case where pages are randomly collocated irrespective of their update frequency and cleaned using a LRU policy. Using the same logic as in Section 4.2.1, we can express the relation between the page cleaning probabilities of the k update sets as:

$$p_{gc1}^{1/f1} = p_{gc2}^{1/f_2} = ... = p_{gck}^{1/f_k}$$

Thus, the point where the difference between update frequencies justifies separating pages remains the same as for the 2-modal workload: pages should be grouped in bins with update frequency increasing exponentially, while the range of frequencies in a bin can vary by $\sim 2\times$. All the results from Section 4.2.1 can be straightforwardly extended to k update sets.

4.3.2 Frequency-Based Data Placement

Similarly to Section 4.2.2, assume now that the k update sets are stored separately. The total cleaning overhead is thus:

$$\mathrm{GC}_{tot} \;=\; f_{s1}\frac{p_{gc1}}{1-p_{gc1}} + \cdots + f_{sk}\frac{p_{gck}}{1-p_{gck}} \qquad (13)$$

When the over-provisioned space is optimally allocated, GC_{tot} is minimized and all the partial derivatives of GC_{tot} with respect to all the $\beta_{1..k}$ dimensions are zero. The partial derivative with respect to a β_i dimension is:

$$\frac{\partial \mathrm{GC}_{kmod}}{\partial \beta_i} \;=\; \frac{\partial \mathrm{GC}_1}{\partial \beta_i} + ... + \frac{\partial \mathrm{GC}_i}{\partial \beta_i} + ... + \frac{\partial \mathrm{GC}_k}{\partial \beta_i}$$

There are $k-1$ degrees of freedom for partitioning space between the k logs, therefore only two components of the partial derivative depend on β_i. Considering β_j to be dictated by the other space allocation values ($\beta_j = \alpha - \sum_{i \neq j}\beta_i$), the β_i derivative becomes:

$$\frac{\partial \mathrm{GC}_{kmod}}{\partial \beta_i} \;=\; \frac{\partial \mathrm{GC}_i}{\partial \beta_i} + \frac{\partial \mathrm{GC}_j}{\partial \beta_i}$$

Thus, every partial derivative has the same form as the derivative of the cleaning overhead for a 2-modal update distribution, while the overall optimal space allocation is the solution to a system of $k-1$ such equations. Equation 12 is sufficient for deciding space allocation also for this update workload and can be interpreted as a way of partitioning a local space budget between any two update sets. More importantly, the same practical methods for distributing space and cleaning the log-structures, as described as in Section 4.2, also holds for k-modal update distributions.

Algorithm 1 Analytical Estimation of Write Overhead

```
1: Group pages in bins with ±50% update frequency
2: Set number of logs := number of bins
3: Set $\beta_1, \cdots, \beta_k := s_1, \cdots, s_k$
4: // Iterative space allocation
5: repeat
6:     for i=1:k do
7:         re-distribute space between log $i$ and $(i+1)$ mod $k$
8:     end for
9: until no $\beta_i$ change
10: Calculate $GC_{tot}$
```

4.3.3 Estimate Cleaning Overhead

In practice we want to leverage the modeling results obtained so far to estimate the cleaning overhead of a real workload. Computing the minimum achievable cleaning overhead is useful both for validating new data placement algorithms and for computing the headroom for improving existing FTL implementations.

We present Algorithm 1 as a method for estimating the lowest achievable cleaning overhead of a workload. The algorithm first bins together pages based on update frequency (Line 1). Binning can rely on either I/O traces, a statistical description of the workload, or high-level access statistics from the DBMS buffer pool replacement policy. Next, the number of logs is initialized to the number of non-empty bins (Line 2), and the initial β_i values are set to the size s_i of each log (Line 3). As mentioned, s_i has the biggest influence on β_i and $\beta_i = s_i$ is a good starting point that allows the algorithm to converge faster. Next, we iterate through all the logs and use Equation 11 to set the optimal β_i values (Lines 5-9). We repeat until all β_i converge and, finally, compute GC_{tot} using Equation 13.

5 Frequency-based Data Placement

In this section, we show how to apply in practice the theoretical results found so far. We propose a new data placement algorithm, rather than a full FTL implementation, that can be used by any page-level FTL or hybrid mapping FTL for the page level mapping area.

The data placement algorithm is depicted in Figure 4. The high-level idea is to split pages in update sets based on update frequency. Each set is stored as a log structure, the log number and log size being determined dynamically. The update sets hold pages with update frequencies decreasing in powers of two (Section 4.2.1). The update set with the hottest data is logically represented on top, while the coldest is at the bottom. If a page becomes cold it "sinks" according to its update frequency, while if it becomes hotter is "rises". The design is intuitive, however, it raises a number of questions that we address in the rest of the section:

1. How to estimate update frequency?
2. How to migrate data between logs?
3. What cleaning policy to use?
4. How to determine the optimal log number and their size?

5.1 Estimating Update Frequency

We do not make any assumptions about the algorithm used to estimate update frequency, and our data placement proposal works with any existing algorithms, either general purpose or especially designed for flash memory [20, 21].

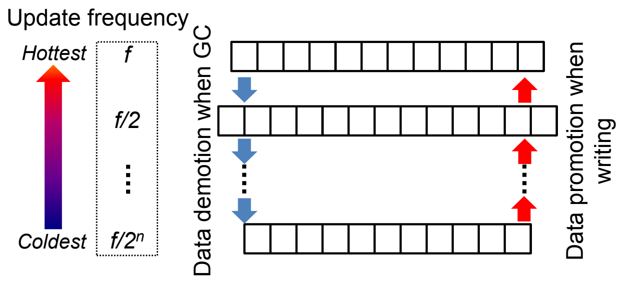

Figure 4: MultiLog data placement

When measuring update frequency, there is a trade-off between higher accuracy (requiring finer grained access statistics) and CPU and memory overhead. As discussed, we can tolerate a high degree of uncertainty when estimating update frequency, ±50% from the real value, and it is an open problem if existing algorithms can be further tuned for the accuracy we are interested in.

We use two methods to measure the page update frequency in our experiments. The first algorithm is a LRU-1 metric that estimates update frequency based on the distance between two consecutive writes to the same page (frequency is the inverse of the update distance). This algorithm is simple enough to implement in any FTL as it does not require storing additional access statistics and has a low CPU overhead. No additional meta-data is required as the logical to physical FTL mapping associates anyway a logical page ID with its last up-to-date physical location (erase block and page offset), while each erase block has an associated version number (write sequence number) for recovery and consistency reasons. Comparing the current version number (write sequence number) with the previous value estimates the update distance, the inverse of the update frequency. We expect more complex frequency estimation methods to be more accurate.

The second algorithm is an "Oracle" that gives us the exact update frequency. Combined, the two algorithms bound the impact of the update frequency estimation algorithm on the data placement proposal and show the improvement headroom over the basic LRU method. The Oracle works by analyzing the workload in advance, both for micro-benchmarks and DBMS I/O traces, and by computing the exact update frequency of each page. For example, in a random update workload, pages have an equal theoretical update probability, but when sampling from the uniform distribution, the number of updates per page are not exactly equal (they follow a Gaussian distribution). Thus, the Oracle measures perfectly the update frequency.

5.2 Page Migration

Page migration raises two questions: i) when to perform page migration, and ii) what pages to move.

Page promotion. We "promote" pages, i.e. move a page to a hotter log when the page is updated. This is desirable as we incur no additional writes – when updating a page, we need to write a new page version and have the liberty to redirect the write to any location on flash. The decision to migrate pages, however, is not based on a simple heuristic such as "promote" any page when updated, or "demote" any page when relocated. Such a strategy is known to fail to separate data according with update frequency [12].

Given a log structure of $s \cdot N$ pages, a page is invalidated on average after $s \cdot N/2$ logical updates to that log. Out of

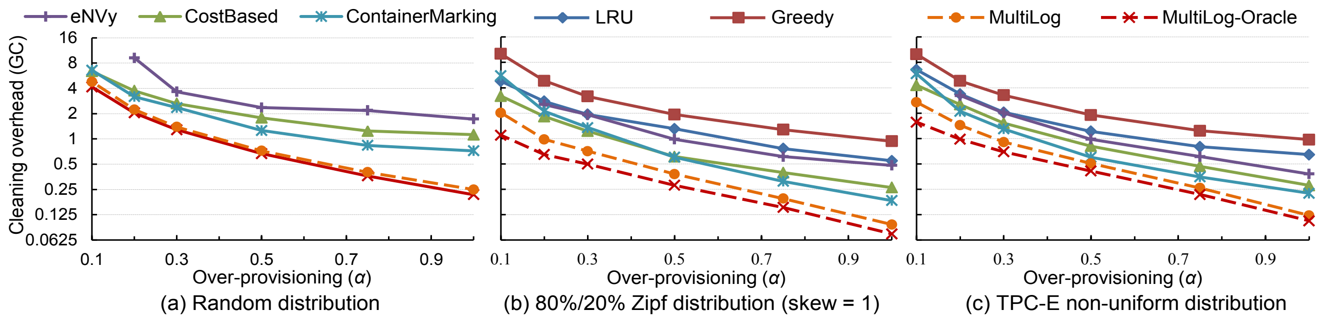

Figure 5: Cleaning cost (GC) as a function of over-provisioning (α) for different update distributions.

the physical writes to the log, the logical writes account for a $1 - p_{gc}$ fraction. Thus, the average expected update distance is: $dst_{exp} = s \cdot N \cdot (1 - p_{gc})/2$. If the measured update distance is lower, we promote the page with probability:

$$p_{promote} = \frac{dst_{expected} - dst_{measured}}{dst_{expected}}$$

The rationale is that the further the measured distance deviates from its expected value, the higher the probability the page has an update frequency significantly higher than the update set average.

Page demotion. Pages are "demoted" when the cleaning process needs to relocate them and, as for the case of page promotion, we avoid additional writes. We estimate the probability that a page is cold enough for demotion starting from the p_{gc} computed from Equation 4. When pages of a block are relocated inside the same log we "pack" and store them together, i.e. preserve their locality. Cold pages tend to accumulate in clusters and increase the percentage of valid pages of a block compared to the expected p_{gc} fraction. When a block with a cluster of cold pages is cleaned, we use p_{gc} to identify the probability that the cluster appeared by chance. The probability that the cluster is made of cold data and needs to be promoted is simply the inverse. For example, if $p_{gc} = 25\%$, the probability of having a cluster of two valid pages is 6.25%, of three pages is 1.56%, etc.; the inverse probability that the cluster is made of cold pages is 83.73% and 98.44% respectively. Thus, the page demotion process has a low false positive rate and accounts implicitly for log occupancy.

5.3 Cleaning Policy

We use a LRU cleaning policy as it is the lightest policy to implement in terms of computation, meta-size, and has the best wear-leveling behavior (erasures are spread evenly across the erase blocks of a log). Policies such as Greedy, Greedy-Window, or Cost-Benefit [23] do not yield significant benefits if pages are stored according to update frequency.

5.4 Number and Size of Log Structures

The number of log structures is automatically decided at run-time. Initially there is a single log where all pages are collocated. A new log is created any time pages are demoted from the last level, provided the size of the last log level is bigger than a minimum value needed to justify the bookkeeping overhead. The minimum log size is set to 10MB in all of the experiments.

Space is re-distributed incrementally at run-time. Whenever log i needs to store an update, we use Equation 13 to clean the LRU block of one of the neighboring logs (either $i-1$, i, or $i+1$) and append the newly erased block to log i to accommodate new updates. The donor log is selected by comparing the GC derivatives as described in Section 4.2.2. The space distribution converges at runtime to the optimal theoretical partitioning that minimizes cleaning overhead.

6 Experimental Results

In this section, we show how our data placement proposal compares with other state-of-art data placement algorithms by using a set of micro-benchmarks and DBMS I/O traces.

6.1 Simulation Setup

Performance metric. We base our comparison on simulating data placement on flash memory and measure the associated cleaning overhead for various data placement algorithms. All figures use a logarithmic y-axis as a linear increase in over-provisioning lowers exponentially the cleaning overhead. As mentioned, GC represents the average number of extra physical writes per logical write. For example, a GC of 0.5 means that final SSD write performance and life are at least 33% lower than the raw aggregate performance and life of the flash chips of the device.

Simulation goal. We do not emulate a whole SSD but only model the cleaning overhead of out-of-place writes on flash memory. An SSD is a very complex mix of hardware and software; we aim to understand data placement on flash memory and thus we need to separate it from other FTL concerns. The actual performance of an SSD depends on a multitude of hardware and software parameters [1].

The only difference between the model validation, micro-benchmarks, and I/O replay traces is how updates are generated. For the model validation and for the micro-benchmarks, we generate updates according to a theoretical distribution, while the traces are the actual I/O write requests collected by running DBMS benchmarks.

Scheduling garbage collection. FTLs usually disconnect cleaning – a high latency process – from servicing I/Os by reclaiming blocks in the background or when the load is low enough. Latency-wise, garbage collection scheduling has a significant impact on the I/O latency, however, it is orthogonal to the problem of data placement. Throughput-wise, any initial reserve of erased blocks is eventually exhausted and write performance becomes limited by the speed of the garbage collection process. We therefore trigger cleaning – for all data placement algorithms – on demand when no free erased blocks are available.

Wear leveling. A possible concern is that blocks belonging to the hotter log structures wear faster than blocks of colder logs. We consider wear leveling an orthogonal topic for two reasons.

Wear leveling can be split in two categories: reactive and pro-active wear leveling. Reactive wear leveling is needed in

	TPCC	TATP
Scale Factor	100	1000
Initial size	13GB	12GB
Growing	Yes	No
Page size	8kB	8kB
Page No.(10^6)	2.85	1.51
Updates (10^6)	59	220

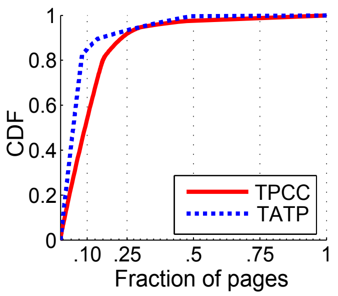

Figure 6: I/O trace statistics

case there is write cold data. If a block sees no updates, it is never cleaned, and therefore it does not age. The content of young cold blocks needs to be swapped periodically with the content of older hot blocks to prevent uneven wearing. Such an activity is needed in all FTLs, no matter their implementation. Pro-active wear leveling happens at writing when updates to hot logical pages are re-directed to young blocks in the hope of evening out future erasures. A straightforward implementation of such wear leveling for our MultiLog proposal, considering that each log structure has a reserve of a few erased blocks, is to simply logically swap hot log blocks with cold log blocks after cleaning. Such an approach results in no extra cleaning overhead.

Also, optimal wear leveling is hardware-specific as the failure patterns of NAND flash chip vary form one generation from the next [9] and also depend on the type of flash memory type (e.g. SLC, MLC, TLC) [4]. Spreading writes uniformly across all erase blocks is beneficial but not sufficient for maximizing device life.

Data placement algorithms. We compare the MultiLog data placement proposal with the LRU and the Greedy cleaning policies, with the Cost-Based policy proposed in FFS [14], with the eNVy proposal (the Hybrid data placement algorithm) [26], and an optimized version of the ContainerMarking [12] scheme with the wear-leveling overhead removed for a fair comparison.

We compare with the Greedy cleaning policy as it represents the most straightforward way to clean flash memory (i.e. pick the block with the lowest number of valid pages for cleaning). DFTL assumes a Greedy cleaning policy, therefore DFTL's cleaning overhead is strictly higher than of the Greedy cleaning policy as DFTL incurs extra overhead for paging data between RAM and flash memory.

We compare with the hybrid eNVy data placement proposal as it offers a practical heuristic for separating data according to access frequency. We could not run all eNVy experiments at lower over-provisioning values (< 0.2) as eNVy's cost-metric that guides cleaning and page migration can create situations when the cleaning collection process is unable to find additional space.

Simulation parameters. For all experiments we use a page size of 4kB and erase blocks of 64 pages and the size of the flash memory was set to 100GB for the micro-benchmarks plus the varying over-provisioning (the actual size of the flash memory has no influence on the final results). For the I/O traces, the size of flash memory was set to the maximum database size plus over-provisioning.

I/O traces. The I/O traces were collected by running the TPC-C [24] and the Nokia TATP [19] standard benchmarks on the Shore-MT[13] DBMS storage engine. The buffer pool size was set at 10% of initial database size.

6.2 Micro-benchmarks

The micro-benchmark results are presented in Figure 5. We use three access distributions: a uniform distribution (random updates), a Zipf distribution with skew factor 1 (80% of accesses target 20% of pages), and the TPC-E [25] non-uniform distribution.

Random updates (Figure 5(a)). For random updates, the MultiLog-Optimal data placement algorithm is equivalent to the LRU cleaning policy as there is no update skew to justify separating data. At the same time, the LRU policy has a cleaning overhead virtually indistinguishable from the Greedy policy. We thus omit both LRU and MultiLog-Optimal for clarity. The MultiLog cleaning cost follows closely the one of the Greedy policy, and is significantly lower (up to 50%) compared to the other data placement proposals. Please note that the Greedy cleaning policy is optimal [11].

Zipf accesses (Figure 5(b)). The importance of data placement grows as update skew increases. Compared to Figure 5(a), all skew-aware data placement algorithms achieve a lower cleaning overhead. For both ContainerMarking and eNVy, the cleaning cost tends to increase sharply at low over-provisioning values and the Cost-Based policy, although theoretically inferior, achieves initially a lower cleaning overhead. We note that eNVy is especially unstable compared to the ContainerMarking and Cost-Based policies. Finally, the LRU and Greedy cleaning policies have the worst behavior as no attention is paid to update skew. It is interesting to note that for the Greedy policy, p_{gc} is determined by device occupancy: All blocks tend to accumulate the same amount of hot and cold data and are cleaned around the same p_{gc} value. As update skew is high and as most updates target a low number of pages, the cold data is spread relatively evenly over all blocks acting as dead-weight when cleaning.

TPC-E accesses (Figure 5(c)). The TPC-E distribution skew can be thought of as an in-between case between the random distribution and the Zipf distribution. Thus, the cleaning overhead of MultiLog increases compared to the Zipf micro-benchmark as there is less skew to exploit. However, MultiLog significantly outperforms the other data placement techniques. eNVy and ContainerMarking have a more robust performance than expected; they achieve a similar or lower cleaning cost compared to the Zipf experiment although update skew decreases.

6.3 DBMS I/O Trace Replay

We show in Figure 6 the high-level statistics of the TPC-C and TATP I/O traces. The high update skew is perhaps surprising as updates in the two workloads are logically uniform at the table level; however, this does not translate in uniformly distributed writes over all the pages. When two objects of different sizes are cached, updates to the smaller object cause a lower number of I/Os as more of the smaller object can be buffered. The same arguments extend to B-Trees where we see the compounded effect of updates being naturally skewed due to the logarithmic nature of the data-structure.

The cleaning overhead is presented in Figure 7(a) for the TPC-C I/O trace, and in Figure 7(b) for the TATP trace. The overall trends are similar to the micro-benchmark and we point out the important differences. The TPC-C workload has a partially shifting working set, as hot newly created data becomes cold over time (mostly due to inserts

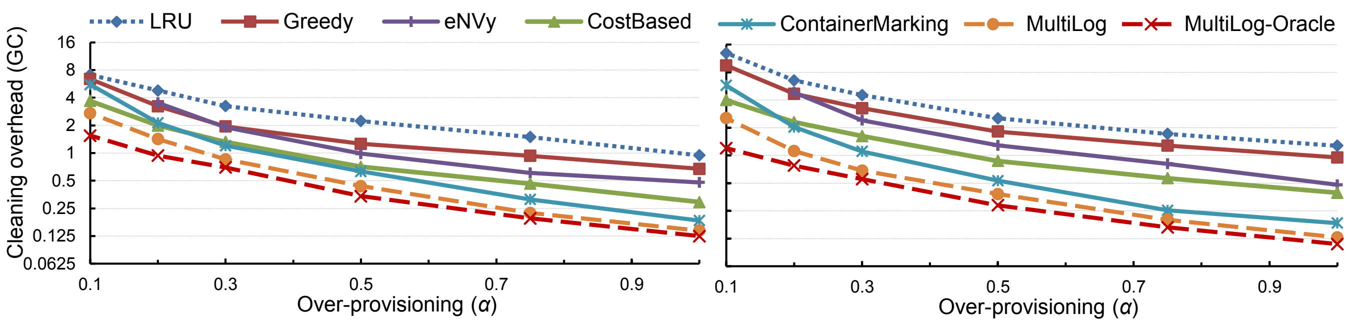

Figure 7: Cleaning cost as a function of over-provisioning for the TPC-C (a) and TATP (b) I/O traces.

to the OrderLine table). The shifting hot set poses the most problems to eNVy, even at high over-provisioning values. Comparatively, ContainerMarking performs much better. Multi-Log achieves on average a 30% GC cleaning overhead reduction compared to ContainerMarking. The reduction is especially significant in the practical 0.1-0.5 over-provisioning range.

The TATP I/O trace is more skewed than the TPC-C trace: around 10% of the pages see 80% of the updates, while updates in the 10% hot set are relatively uniform and constant over time. MultiLog performs comparatively better to the TPC-C traces as it can efficiently exploit the higher update skew. All other data placement algorithms perform equal or worst to the TPC-C workload. For TATP, the key to a low cleaning cost is to be able to fully separate the hot set from the rest of the cold data.

Overall, MultiLog achieves consistently a lower cleaning overhead and reduces garbage collection overhead by 20% − 75% at all over-provisioning values for both I/O traces.

7 Conclusions

This paper demonstrates how to reduce FTL cleaning overhead and improve SSD life by using the write skew of I/O workloads to guide data placement. We developed an analytical model that explains the important trade-offs involved when updating data out-of-place on flash memory. Based on the modeling results, we then proposed a principled data placement algorithm that exploits write skew close to optimally and reduces cleaning overhead by 20%-75% for both synthetic workloads and real DBMS I/O traces.

8 References

[1] N. Agrawal et al. Design tradeoffs for SSD performance. In *USENIX*, pages 57–70, 2008.

[2] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, pages 1–12, 2009.

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

[4] Y. Cai et al. Error patterns in MLC NAND Flash memory: Measurement, characterization, and analysis. In *DATE*, pages 521–526, 2012.

[5] M.-L. Chiang et al. Using data clustering to improve cleaning performance for flash memory. *Software Practice and Experience*, 29(3):267–290, 1999.

[6] H. Choi, S. Lim, and K. Park. JFTL: a flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage*, 4(4):14, 2009.

[7] R. Corless et al. On the LambertW function. *Advances in Computational Mathematics*, 1(5):329–359, 1996.

[8] L. Grupp, J. Davis, and S. Swanson. The Bleak Future of NAND Flash Memory. In *FAST*, pages 2–2, 2012.

[9] L. Grupp et al. Characterizing flash memory: anomalies, observations, and applications. In *MICRO*, pages 24–33, 2009.

[10] A. Gupta et al. *DFTL*: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, volume 44, pages 229–240, 2009.

[11] R. Haas and X. Hu. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. *TR IBM Research*, 2010.

[12] X. Hu et al. Container Marking: Combining Data Placement, Garbage Collection and Wear Levelling for Flash. In *MASCOTS*, pages 237–247, 2011.

[13] R. Johnson et al. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[14] A. Kawaguchi et al. A Flash-Memory Based File System. In *USENIX*, pages 155–164, 1995.

[15] S. Lee et al. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[16] S.-W. Lee et al. A log buffer-based flash translation layer using fully-associative sector translation. *Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[17] Y. Lee et al. μ-FTL: a memory-efficient flash translation layer supporting multiple mapping granularities. In *International Conference on Embedded Software*, pages 21–30, 2008.

[18] D. Ma, J. Feng, and G. Li. LazyFTL: A Page-level Flash Translation Layer Optimized for NAND Flash Memory. In *SIGMOD*, volume 48, pages 366–375, 2011.

[19] Nokia. Network Database Benchmark. Available: `hoslab.cs.helsinki.fi/homepages/ndbbenchmark`.

[20] D. Park and D. Du. Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters. In *MSST*, pages 1–11, 2011.

[21] D. Park et al. HotDataTrap: A Sampling-based Hot Data Identification Scheme for Flash Memory. In *SAC*, pages 1610–1617, 2012.

[22] M. Rosenblum and J. Ousterhout. The Design and Implemention of a Log-Structured File-System. In *SOSP*, pages 1–15, 1991.

[23] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File-System. In *ACM Transactions on Computer Systems*, volume 10, pages 26–52, 1992.

[24] Transaction Processing Council. TPC-C Standard Specification. Available: `www.tpc.org/tpcc`.

[25] Transaction Processing Council. TPC-E Standard Specification. Available: `http://www.tpc.org/tpce` .

[26] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *ASPLOS*, volume 29, pages 86–97, 1994.