Une couche indépendante des FTL pour améliorer les performances des écritures aléatoires sur mémoires flash

Brice Chardin[1,2], Olivier Pasteur[1], Jean-Marc Petit[2]
[1] EDF R&D, 6 Quai Watier, Chatou, France
[2] Université de Lyon, CNRS, INSA-Lyon, LIRIS UMR5205,
7 avenue Jean Capelle, F-69621 Villeurbanne, France
<brice.chardin, jean-marc.petit>@liris.cnrs.fr
olivier.pasteur@edf.fr

Résumé

Les mémoires flash deviennent une alternative compétitive aux disques durs comme support de stockage non-volatile pour les systèmes de gestion de base de données. Cependant, des adaptations spécifiques sont nécessaires pour exploiter au mieux cette nouvelle catégorie de stockage. Pour cela, la définition des modèles d'accès préconisés est un problème complexe car les mémoires flash sont très hétérogènes et difficiles à caractériser, à cause de la Flash Translation Layer – ou FTL – qui leur est intégrée. Dans cet article nous identifions une corrélation forte entre les performances des écritures et leur proximité spatiale pour un sous-ensemble des mémoires flash ; puis définissons une distance pour quantifier cet effet. À partir de cette propriété, nous proposons un algorithme de placement des données permettant d'améliorer les performances en écriture aléatoire en contrepartie d'une diminution de la capacité de la mémoire. L'efficacité de cette technique est validée par une formalisation avec un modèle mathématique et des résultats expérimentaux. Avec cette optimisation, les écritures aléatoires deviennent potentiellement aussi efficaces que les écritures séquentielles, allant jusqu'à améliorer leurs performances de deux ordres de grandeur.

Mots clés Mémoires flash, SSD, FTL, Optimisation, Écritures aléatoires

1 Introduction

For the sake of interchangeability, many flash memories include a Flash Translation Layer – abbreviated as FTL – to comply with the block interface, a rotating disk legacy. In addition to providing block write and read operations, the FTL manages flash chips complex writing mechanism [5]. However, this layer is implemented with proprietary and undocumented software, which makes flash devices appear as "black boxes" from a system's point of view [2].

Advantageously, this FTL allows a straightforward substitution between both storage technologies. Yet, most database management systems include rotating disks-oriented optimizations, which are not relevant for flash memories. Even if both technologies use the same block interface, they have different preferred access patterns. Database management systems could potentially benefit from flash memories as they provide fast random access for read operations. Still, for FTL-based devices, random writes are generally not as efficient as sequential writes [4] and most optimization techniques for flash memories relate to this specific issue.

In this paper, we identify for a subset of FTL-based devices a strong correlation between write performances and their spatial locality; and define a distance to quantify this effect. From this property, we propose a simple data placement algorithm, which trades flash memory space for random write performances. Its efficiency is validated by a formalization with a mathematical model, along with experimental results. With this optimization, random write potentially become as efficient as sequential write, improving random write speed by up to two orders of magnitude.

The rest of this paper is organized as follow. Section 2 introduces NAND flash memories and different types of mapping used in the FTL. Section 3 emphasizes the importance of locality on these devices for write performances and defines a distance between consecutive writes to quantify this effect. In section 4, we derive from this property an optimization technique for random writes, using an indirection layer to minimize this distance, thus avoiding scattered writes. In section 5, we present an approximate model for this algorithm. The results of both our experiments and model are reported in section 6. Related works are described in section 7. Then, section 8 summarizes the contributions of this paper.

2 NAND flash memories

NAND flash memory is a non-volatile storage technology, which allows three low-level data-access operations: read, write (or *program*) and erase. Still, erasing is performed at a different granularity than reading or writing: NAND flash chips are divided into blocks that can be erased independently, each block containing a fixed number of pages, each of which being individually accessible for reading or writing. As overwrites are not allowed, a full block must be erased prior to writing on one of its already used page. Additionally, pages within a block must be written sequentially.

To handle this complex writing mechanism, most flash memories include a Flash Translation Layer (FTL) that redirects writes on available (erased) pages and stores the associations between the logical sector identifier and its physical location in an address translation table.

In most cases, this translation operates on a page-level basis or on a block-level basis [5]. With a page mapping FTL, each logical page has its associated physical page. After an overwrite, the translation table is updated with the new physical location and the old physical location is marked as obsolete to be reclaimed by a garbage collection mechanism.

With a block mapping FTL, each logical block has its associated physical block and an additional logging area, which consists of log blocks. When a page is overwritten, new data are appended to the last log block. Garbage collection merges a data block with all its associated log blocks by copying every valid page on a new (erased) data block and updating the translation table.

Each mapping granularity has its own drawbacks. Page mapping has a higher memory overhead because of its larger address translation table; and has a complex and possibly less efficient garbage collection mechanism [5], while block mapping performances are highly dependent on empty blocks availability, to serve as log blocks.

3 Write locality for FTL-based devices

As FTL enclosed in flash devices are usually proprietary and undocumented, studies have been conducted to identify preferred write access patterns for such devices. Wang et al. study in [11] the effectiveness of using non-in-place update techniques for flash-based DBMS. Their experiments with log-structured file systems validate potential benefits as they achieve up to x6.6 performance improvement. However, our experiments – presented later in this section – show that in-place updates are not an issue for most

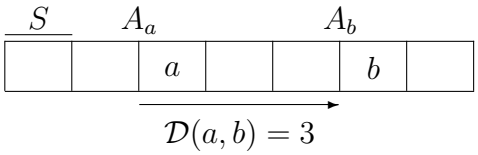


Figure 1: Distance between two sectors a and b

FTL-based devices. In contrast with raw flash chips, constant rewrites at the same logical address are even faster than sequential writes. Consequently, we believe log-structured file systems benefits result from sequencing and localizing writes, considering that scattered writes on flash memories are inefficient.

uFLIP [2] is a component benchmark designed to quantify the behavior of flash-memories when confronted to defined I/O patterns. Some of these patterns relate to locality and increments between consecutive writes. Their results confirm that localizing random writes greatly improve efficiency and large increments lead to performances which could be even worse than random writes. However, small variations of this increment are not tested, which we propose to address in our experiments.

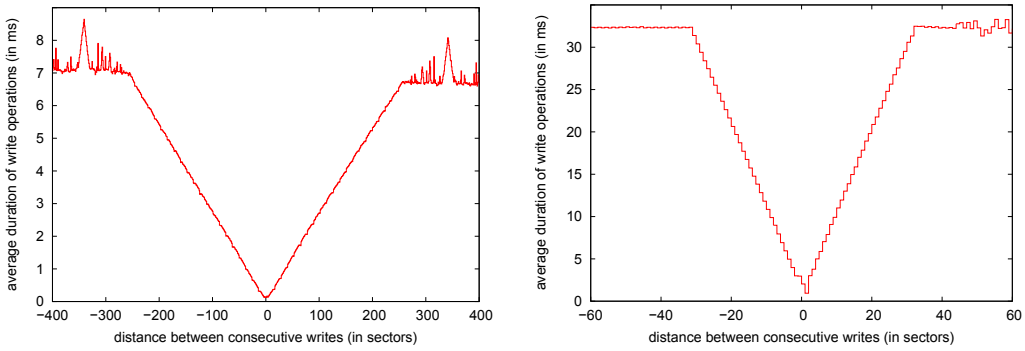
These works reveal that most characteristics are heterogeneous and device-dependent, still, locality has a frequent impact on write performances. While the causes of this behavior are mostly hidden by the FTL complexity, a possible explanation is that, with block-mapping, data modifications belonging to the same logical block are logged on the same log blocks, which reduces the cost of merging logs with data when garbage collection is performed. This mechanism results in a strong correlation between logical write spatial locality and performances.

To quantify this effect on FTL-based devices, we first introduce a notion of distance between consecutive writes. Its definition relies on the "block device" abstraction provided by the FTL. We precise the definition of sectors as non-overlapping, contiguous sequences of S bytes; the address of a sector being the position of its first byte in the flash memory. As the FTL hides the erase mechanism, sectors can only be (re)written and read.

Definition 1 *Let a and b be two sectors of size S located at addresses A_a and A_b respectively.*
Let $\mathcal{D}(a,b)$ be the directed distance between a and b :

$$\mathcal{D}(a,b) = \frac{A_b - A_a}{S}$$

Figure 1 illustrates this distance between two sectors a and b. In absolute value, this distance equals the number of sectors between a and b plus 1. This



(a) SSD (b) USB key

Figure 2: Influence of distance on write duration

metric can be negative to discriminate between increasing and decreasing address values. For instance, if d is the distance between consecutive writes, $d = 0$ refers to in-place updates, $d = 1$ refers to sequential writes and $d = -1$ refers to reverse sequential writes.

From the results of [2, 11], our assumption is that write duration – or cost – is correlated with this distance, as reducing this distance linearly improves the probability to be in the same logical block.

To validate this assumption, we measured the effect of distance on a variety of flash devices. In our experiments, the average write duration for each distance d is evaluated, with write addresses being incremented – respectively decremented if d is negative – by $S \times |d|$, that is to say skipping $|d| - 1$ sectors between consecutive writes. Although individual write durations are erratic, their average value converge when this access pattern is sustained.

Figure 2 shows that our assumption is verified for a flash-based SSD [1] and a USB key [2]. Nevertheless, some flash devices do not have such a strong correlation between distance and writing speed. Consequently, the optimization technique presented in section 4 is only applicable to a subset of flash devices. Because of similar effects of locality, we believe block-mapping FTL to be characteristic of this behavior. Block-mapping is a widespread mapping technique for flash memories [5].

From the results of these experiments, we conjecture a usual behavior where, up to a distance d_{max}, the average cost of a write operation $cost(d)$ is approximately proportional to d.

1. SSD Mtron MSD SATA3035-032, sector size $S = 4$ KiB
2. Flash chip HYNIX HY27UG088G5B with an ALCOR AU6983HL controller, sector size $S = 4$ KiB

Definition 2 *Let $cost(d)$ be the cost of a write operation, given the distance with the preceding write d.*

$$cost(d) \propto |d|, |d| \leq d_{max}$$

Scattered writes (ie. $d \geq d_{max}$) are typically 20 to 100 times slower than sequential writes for these devices [2]. Consequently, and because of this proportional performance pattern, reducing the average distance between consecutive writes can significantly improve efficiency, even if strict sequential access (d=1) is not achieved. Skipping a few sectors when writing sequentially is an access pattern defined as "skip-sequential". The optimization described in the following section focuses on this access pattern, skipping as little sectors as possible.

4 Gathering random writes

Online transaction processing usually have a part of its workload constituted of small random writes [8]. The optimization described in this section converts these random writes into skip-sequential writes, which should increase performances on flash memories. With this optimization, sectors containing valid data (used sectors) and unused sectors are mixed on the device. An additional indirection layer is used to redirect logical writes to unused sectors by minimizing the distance between consecutive writes.

To allow data retrieval, correspondences between physical and logical locations are stored in an address translation table, with every logical sector being associated with a physical sector. Unused sectors – which are not associated with any logical sectors in the address translation table – do not contain any useful data, and therefore constitute a pool of free sectors available for writing.

To overwrite a logical sector, data are assigned to a pool sector adjacent to the previous write. Then the logical-physical association stored in the address translation table is updated, the previously associated sector therefore being freed and added to the pool. Figure 3 illustrates how logical writes are assigned to physical locations, when writing successively on logical sectors 0, 3 and 0.

This optimization does not require garbage collection, as the size of the pool remains constant: physical sectors containing obsolete data are immediately added to the pool, and can be overwritten. Yet, as an independent and internal mechanism, the FTL might still use garbage collection to handle flash erasures.

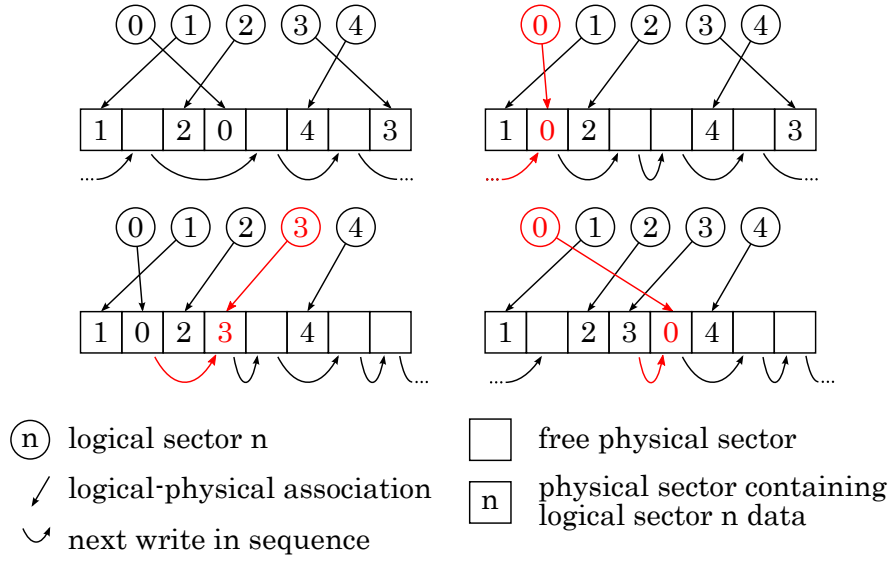


Figure 3: Optimization overview

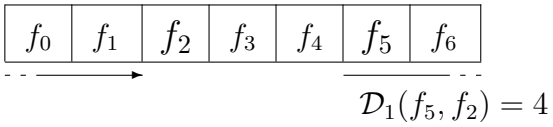


Figure 4: Positive distance between sectors f_5 and f_2

Any logical access pattern will lead to a skip-sequential physical access pattern. Consequently, the average distance (and thus write efficiency) is determined exclusively by the proportion of pool sectors. As increasing pool size requires additional non-volatile memory space, this characteristic can be adjusted to obtain an expected efficiency.

This indirection layer substitutes skip-sequential writes for random writes. As a downside, sequential reads are also transformed into random reads. However, this behavior is not an issue for flash devices, as random reads are as efficient as sequential reads [2]. Still, read operations induce lookups in the address translation table, which is a negligible overhead.

To prevent revisiting regions of the memory recently accessed, where pool sectors should have been exhausted, only positive distances are considered in this optimization. Additionally, the addressable space is assumed to be circular, in order to avoid handling edges differently. Figure 4 illustrates this metric, and flash sectors naming convention.

Notation 1 *Let $\mathbb{F}$ be the set of sectors accessible from the device, hereafter*

referred to as flash sectors. Indices express physical contiguity.

$$\mathbb{F} = \{f_i\}, 0 \leq i < |\mathbb{F}|$$

Definition 3 *Let* $\mathcal{D}_1$ [3] *be the restriction of* $\mathcal{D}$ *to positive values, and the addressable space be circular.*

$$\mathcal{D}_1(f_i, f_j) \equiv j - i \pmod{|\mathbb{F}|},\ 0 \leq \mathcal{D}_1(f_i, f_j) < |\mathbb{F}|$$

Analogously, we use the following notations for the sets of logical and pool sectors, labeled $\mathbb{L}$ and $\mathbb{P}$, respectively.

Notation 2 *Let* $\mathbb{L}$ *be the set of logical sectors.*

$$\mathbb{L} = \{l_i\}, 0 \leq i < |\mathbb{L}|$$

Notation 3 *Let* $\mathbb{P}$ *be the set of flash sectors not associated with a logical sector, referred to as pool sectors. By definition,* $|\mathbb{P}| = |\mathbb{F}| - |\mathbb{L}|$. *T represents the address translation table, described in definition 4.*

$$\mathbb{P} = \{f_i \in \mathbb{F} : \forall l_j \in \mathbb{L}, f_i \neq T(l_j)\}$$

The first data structure used by this algorithm is the address translation table. This table – named T – binds every logical sector in $\mathbb{L}$ to a flash sector in $\mathbb{F}$.

Definition 4 *Let* T *be the address translation table.* $T : \mathbb{L} \mapsto \mathbb{F}$ *is an injective function associating every logical sector with a flash sector.*

As this optimization aims at minimizing the average distance between consecutive writes, the most recent written sector is referred to as $f_{\curvearrowleft}$.

Notation 4 *Let* $f_{\curvearrowleft} \in \mathbb{F}$ *be the previously written flash sector.*

A simple version of the redirection algorithm involves the following operations when writing a logical sector $l \in \mathbb{L}$:

1: $f \leftarrow$ closest pool sector from $f_{\curvearrowleft}$
2: write data on f
3: $T(l) \leftarrow f$ {update the translation table}
4: $f_{\curvearrowleft} \leftarrow f$

3. $\mathcal{D}_1$ is not a distance in a mathematical sense

Operation (1) – searching the pool sector closest to the previous write – has to be implemented carefully with an adequate data structure. In our implementation, to hasten lookups of this sector, we keep references of every sector in the pool in an ordered list, where sectors are arranged by increasing distances from $f_{\curvearrowleft}$.

Definition 5 *Let* $P : \big[0, |\mathbb{P}|\big[\mapsto \mathbb{P}$ *be the list of sectors in* $\mathbb{P}$ *ordered by increasing distances from* $f_{\curvearrowleft}$.

$$i > j \Leftrightarrow \mathcal{D}_1\Big(f_{\curvearrowleft}, P(i)\Big) > \mathcal{D}_1\Big(f_{\curvearrowleft}, P(j)\Big)$$

Including this ordered list of pool sectors allows efficient retrievals of closest pool sectors. Nevertheless, this list has to be updated with each newly freed sector, whenever the translation table is altered. With this additional data structure, writing on a logical sector $l \in \mathbb{L}$ implies the following operations:

1: $f \leftarrow T(l)$
2: write data on $P(0)$
3: $T(l) \leftarrow P(0)$
4: remove $P(0)$ from P
5: add f in P

Operations (4) and (5) can be done asynchronously (ie. during the subsequent write in a write-intensive environment), as the list of pool sectors P can be rebuild from the translation table T. Consequently, P might not be up-to-date for each write request, which results in a slight increase of the average distance between consecutive writes if the closest pool sector from $f_{\curvearrowleft}$ is not yet referenced in this list. However, with large values of $|\mathbb{P}|$, this case appears infrequently (ie. 1 out of $|\mathbb{P}|$ times) and can be neglected.

5 Model

To estimate write speed improvement provided by this algorithm, we propose to model its behavior by evaluating the average cost of a write operation. This model is based on the simplifying assumption that pool sectors are uniformly distributed within flash sectors. This state is also supposed to be stable with occurring writes. Additionally, writing cost is expected to be determined exclusively by its physical distance from the previous write.

Under these approximations, the overall speed improvement can be evaluated given the probability to obtain each possible distance, and their associated costs.

Definition 6 *Let $p(d)$ be the probability that the sector $f_i \in \mathbb{P}$ which minimizes $\mathcal{D}_1(f_\curvearrowright, f_i)$ also verifies $\mathcal{D}_1(f_\curvearrowright, f_i) = d$, namely having a distance d between two consecutive writes.*

With the uniform distribution assumption, the probability $p(d)$ to get a distance d between two consecutive writes can be estimated as the ratio between favorable and possible distributions :

$$p(d) = \frac{\binom{|\mathbb{F}|-d-1}{|\mathbb{P}|-1}}{\binom{|\mathbb{F}|-1}{|\mathbb{P}|}}$$

The cost of a write operation conditioned by its distance from preceding write, $cost(d)$, can be approximated but also measured from the device, as shown in figure 2. For our evaluations, the later is believed to be more accurate.

Given these two parameters, $p(d)$ and $cost(d)$, the average cost of a write operation, named $cost_{avg}$, amounts to :

$$cost_{avg} = \sum_{d=1}^{|\mathbb{L}|} p(d) \times cost(d)$$

Estimations from this model are reported in section 6, together with experimental results. In addition to theoretical performance gains, resource usage can be quantified as this optimization trades CPU, RAM and flash memory space for writing speed.

CPU overheads occur when handling the translation table and the list of pool sectors during a write operation. These overheads relate to the following operations :

- search for the closest pool sector, which is $\mathcal{O}(1)$ when pool sectors references are stored in an ordered list,
- update the translation table, which is $\mathcal{O}(1)$,
- update the list of pool sectors, which is $\mathcal{O}(log\,|\mathbb{P}|)$ with optimized data structures, such as skip-lists.

Updating the list of pool sectors is the only operation with significant CPU cost. However, as stated in section 4, this update might be asynchronous.

For read operations, looking up correspondences between logical and flash sectors in the address translation table results in constant CPU overhead, which is negligible compared to a flash sector read duration.

RAM overheads are caused by the translation table and the list of pool sectors being kept in main memory. These overheads amounts to $\mathcal{O}(|\mathbb{L}| \times$

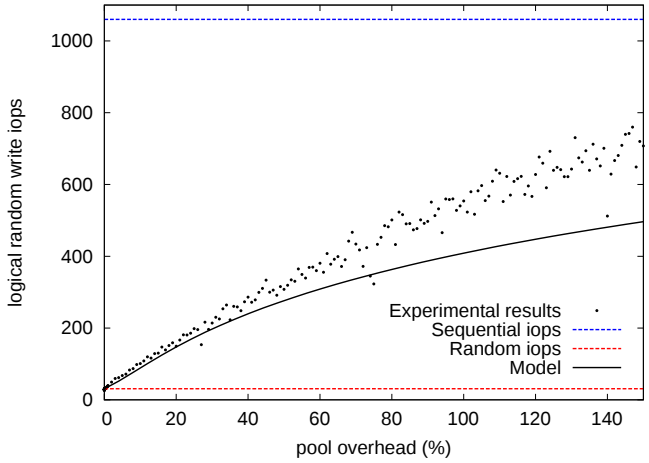


Figure 5: Logical random write performance for 100,000 logical sectors

$log\,|\mathbb{F}|$) for the translation table, and $\mathcal{O}(|\mathbb{P}| \times log\,|\mathbb{F}|)$ for the pool. Total RAM overhead adds up to $\mathcal{O}(|\mathbb{F}| \times log\,|\mathbb{F}|)$.

As pool sectors are stored on the device, and do not hold any useful data, flash memory space overhead amounts to $|\mathbb{P}|$ sectors.

The last significant trade-off involves sequential writes. Since, with this algorithm, performances do not depend on the access pattern, logical sequential writes have the same performances as logical random writes. Existing attempts to sequentialize accesses would not bring any additional performance gain and should be discarded.

6 Results

To validate this optimization together with the model detailed in the previous section, the data placement algorithm is tested on both devices mentioned in section 3. These tests consist in evaluating the average cost of logical random writes for varying sizes of the pool.

Figure 5 shows experimental results and the model expectations for the USB Key. To compare with conventional access patterns, random write and sequential write iops (respectively 30 and 1060) are also reported on this figure.

To obtain performances equivalent to sequential writes, consequent sacrifices have to be made in terms of flash memory space. In our experiment, 95% of sequential write efficiency is achieved when the pool is about three times larger than the logical address space. However, we achieved significant

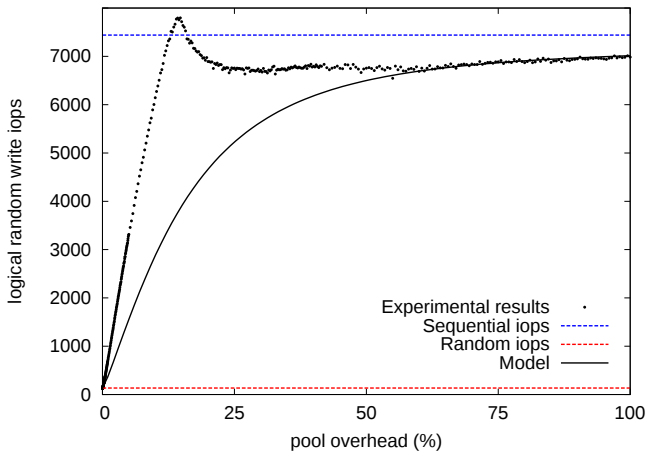


Figure 6: Logical random write performance for 200,000 logical sectors

improvements over random writes with acceptable trade-offs, as we have a ten times improvement with 50% flash memory space overhead.

Contrastingly, writing speed on the SSD is improved with distances below $d_{max} = 256$, instead of $d_{max} = 32$ for the USB key. As a result, notable improvements are achieved with relatively lower sacrifices. Experimental results for this device are reported on figure 6.

Another noticeable difference was suggested by measurements obtained in section 3: figure 7 focuses on small distance values. Remarkably, a skip-sequential access pattern with a distance of 4 sectors between consecutive writes shows relatively good performances. Highest iops are achieved with a pool size of about 29,000 sectors, which results in an average – but still random – distance of 4. This property allows optimal performances to be achieved with much less overhead. Indeed, our optimization reaches 7796 average iops for 4KB logical random writes at a cost of only 687 KB of RAM, and 14.5% flash overhead for 800 MB of usable data space. Compared to physical random writes 134 average iops, performances are improved by $\times 58$.

Determining this optimal pool size is not straightforward, and depends on the sector size. With 16 KiB sectors, experiments give an optimum distance of 2; and about 1.6 for 32 KiB sectors. A possible explanation for this behavior is that interleaving favors non-zero sized skips to access multiple internal flash chips in parallel [12].

Unfortunately, this “peak” behavior might not be representative of flash solid-state drives. Among the twelve SSD with uFLIP results available, only one (by the same manufacturer as ours) expose the same characteristic. This

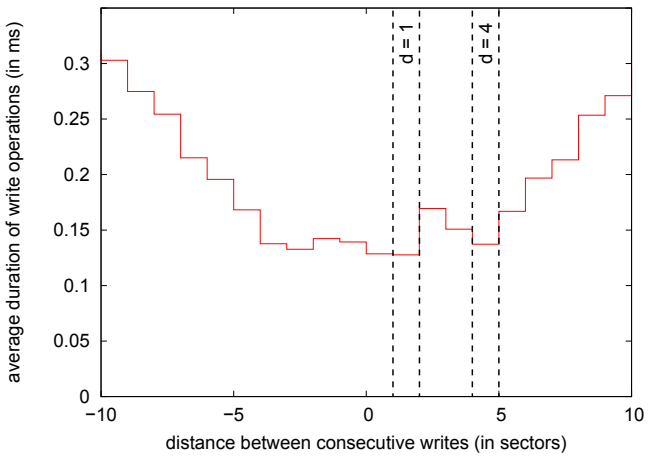


Figure 7: Skip-sequential vs. sequential write cost

singularity is only a facultative additional benefit as it was not part of our initial assumptions. Our model might be closer of what we would expect with a regular SSD, and still shows considerable performances gains, such as a x40 improvement over random writes with 25% flash memory overhead.

Still, these results reveal some limitations of our model. One of its simplifying assumptions is that writing cost is determined exclusively by its physical distance from the previous write, which might not be accurate.

Moreover, experiments outperform our model as memory is accessed in only one direction – increasing physical addresses – to prevent revisiting regions of the memory where pool sectors should have been exhausted. This improvement over our theoretical uniform distribution reduces the average distance between consecutive writes.

7 Related Works

Many optimizations have been conceived with flash chips characteristics in mind. A frequent design avoids in place updates with log-based methods.

The In-Page Logging Approach [7] allocates a portion of each bloc to write updates of its pages. This optimization improves writing speed, as updates are written sequentially inside the erase unit, at the expense of more read operations. Garbage collection consists in merging data pages with their log sectors on a new empty block.

Page-Differential Logging [6] uses a similar approach, except page differential are logged. Writing is improved as differentials of multiple pages can be combined to fit in a single page. Also, differentials are recomputed from

the original page for each overwrite, which implies that reading a logical page involves at most two physical read (the original page and its last differential). The garbage collection mechanism is also improved, as merging a page with its current differential is not required (both can be copied separately).

Log-structured file systems, with a distinction between file systems designed for raw flash chips (without FTL) – like YAFFS, LogFS, JFFS – and those designed for block devices – like LFS – use methods comparable to the in-page logging approach, and therefore provide similar benefits and drawbacks. Additionally, I/O patterns of log-structured file systems for block devices when accessing multiple files tend to be of small size and scattered.

Regarding more specific use cases, B-File [9] is an abstraction layer for self-expiring items on flash memories. Depending on their expiration date, items are written sequentially in appropriate erase units to avoid copying valid data on deletion. Another approach defines an Append and Pack Layout [10], which divide the database in two separate datasets, respectively write-hot and write-cold. These datasets are written sequentially in multiples of the erase block size, with space reclamation when the memory is full.

The main differences between these approaches and our optimization are the necessity of a garbage collection mechanism and decreased read performances as a logical read rely on multiple physical reads. In contrast with these works, we optimize data access exclusively over the FTL. As a result, our approach is not applicable to raw flash chips.

RS-Wrapper [12] is a simple conversion between random writes and sequential writes for FTL-based devices. When random writes are adequately dense, their experiments show that reading the missing pages to overwrite sequentially the entire data range outperforms overwriting exclusively modified pages. However, reorganizing random writes to constitute a skip-sequential access pattern has not been tested.

FlashLogging [3] is an efficient mechanism for synchronous logging on multiple low capacity flash devices. While the use case differs from our proposition, this approach could be used to address the non-volatile issue of our current optimization. Indeed, data written to the device are volatile, since the address translation table is stored in RAM, and is needed to rebuild the database. Logging its modifications on additional flash devices could provide an efficient solution. This issue could also be managed by writing logical addresses together with data, similarly to the FTL internal functioning.

On a different but related subject, enterprise class SSD can provide better random write performance at the cost of additional RAM, processing power and spare blocks (not accessible from the host) [1, 8]. However, these designs focus on random write and provide invariably good performances for the entire device. Most database applications mix random and sequential accesses

and do not require such homogeneous random write efficiency. By adding a software layer, our optimization permit using less expensive personal-class SSD with good, yet spatially limited, random write performances. This is also applicable to removable flash media, which have lessened hardware capabilities.

8 Conclusion

In this paper, we first introduced a notion of distance, and described its impact on flash memories write performances. Based on this property, we proposed a data placement algorithm, which significantly improves random write performances. Our contributions emphasize the importance of locality for these FTL-based devices, and we believe skip-sequential access patterns to be of use for future data placement optimizations.

Compared to native write operations over the FTL, our optimization benefit from the host available RAM and processing power to improve random write efficiency on portions of the device. This method support localized performances adjustment, while flash memories offer homogeneous behaviors.

For the SSD used in our experiments, we achieved an improvement of up to $\times 58$ at a cost of only 14.5% flash overhead. In this best case scenario, this technique even caused random write to perform slightly better than sequential write, by 3.5%. This optimization is, to some extent, also applicable on flash memories with less capacity, as results with a USB key show a $\times 10$ improvement with 50% flash overhead. Conjointly, we proposed a model to predict write performances, however, future works are needed to enhance its accuracy and help tradeoffs adjustments.

Another perspective relate to data volatility, which might be addressed in future works with solutions proposed in section 7. Still, this optimization is already applicable for indexation or temporary tables, where volatility is acceptable.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, 2008.

[2] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR '09: Proceedings of the 4th biennial Conference on Innovative Data Systems Research*, 2009.

[3] S. Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *SIGMOD '09: Proceedings of the 35th international conference on Management of data*, pages 73–86, 2009.

[4] J. Gray and B. Fitzgerald. Flash Disk Opportunity for Server Applications. *Queue*, 6(4):18–23, 2008.

[5] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *EMSOFT '09: Proceedings of the 7th international conference on Embedded software*, pages 295–304, 2009.

[6] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song. Page-differential logging: an efficient and DBMS-independent approach for storing data into flash memory. In *SIGMOD '10: Proceedings of the 36th international conference on Management of data*, pages 363–374, 2010.

[7] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 33th international conference on Management of data*, pages 55–66, 2007.

[8] S.-W. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD '09: Proceedings of the 35th international conference on Management of data*, pages 863–870, 2009.

[9] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *Proc. VLDB Endow.*, 1(1):970–983, 2008.

[10] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMoN '09: Proceedings of the 5th international workshop on Data Management on New Hardware*, pages 9–14, 2009.

[11] Y. Wang, K. Goda, and M. Kitsuregawa. Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems. In *DEXA '09: Proceedings of the 20th International Conference on Database and Expert Systems Applications*, pages 777–791, 2009.

[12] D. Zhou and X. Meng. RS-Wrapper: random write optimization for solid state drive. In *CIKM '09: Proceeding of the 18th conference on Information and knowledge management*, pages 1457–1460, 2009.