From In-Place Updates to In-Place Appends: Revisiting Out-of-Place Updates on Flash

Sergey Hardock
Databases and Distributed Systems Group
TU-Darmstadt, Germany
hardock@dvs.tu-darmstadt.de

Ilia Petrov
Data Management Lab
Reutlingen University, Germany
ilia.petrov@reutlingen-university.de

Robert Gottstein
Databases and Distributed Systems Group
TU-Darmstadt, Germany
gottstein@dvs.tu-darmstadt.de

Alejandro Buchmann
Databases and Distributed Systems Group
TU-Darmstadt, Germany
buchmann@dvs.tu-darmstadt.de

ABSTRACT

Under update intensive workloads (TPC, LinkBench) small updates dominate the write behavior, e.g. 70% of all updates change less than 10 bytes across all TPC OLTP workloads. These are typically performed as in-place updates and result in random writes in page-granularity, causing major write-overhead on Flash storage, a write amplification of several hundred times and lower device longevity.

In this paper we propose an approach that transforms those small in-place updates into small update deltas that are appended to the original page. We utilize the commonly ignored fact that modern Flash memories (SLC, MLC, 3D NAND) can handle appends to already programmed physical pages by using various low-level techniques such as ISPP to avoid expensive erases and page migrations. Furthermore, we extend the traditional NSM page-layout with a delta-record area that can absorb those small updates. We propose a scheme to control the write behavior as well as the space allocation and sizing of database pages.

The proposed approach has been implemented under Shore-MT and evaluated on real Flash hardware (OpenSSD) and a Flash emulator. Compared to In-Page Logging [21] it performs up to 62% less reads and writes and up to 74% less erases on a range of workloads. The experimental evaluation indicates: (i) significant reduction of erase operations resulting in twice the longevity of Flash devices under update-intensive workloads; (ii) 15%-60% lower read/write I/O latencies; (iii) up to 45% higher transactional throughput; (iv) 2x to 3x reduction in overall write amplification.

SIGMOD'17, May 14-19, 2017, Raleigh, NC, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035958

1. INTRODUCTION

The architecture and algorithms of data-intensive systems have been built around the properties of traditional hardware technologies. Many design decisions have been made to efficiently exploit external storage based on magnetic disks and compensate for its shortcomings (for instance significant access gap, high random latency, low I/O parallelism). Many of those elementary assumptions in data management are 20-30 years old and reflect outdated hardware characteristics. Consider, for instance, the following two:
• *The smallest database unit of I/O is a whole DB page, typically aligned to a physical block for performance reasons. It has been constantly growing in size for the last decades [14].*
• *The DBMS uses page layouts co-locating data and page/tuple metadata, regardless of the different volume and update rates.*
Both result in significant write-overhead, increased wear and short longevity of storage devices as well as low performance.

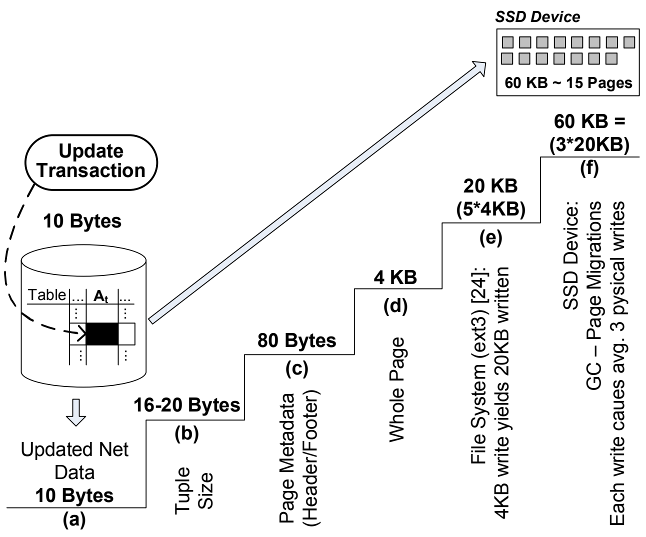
SSD Device
60 KB ~ 15 Pages
60 KB = (3*20KB)
(f)
Update Transaction
10 Bytes
Table | ... | A_t | ...
20 KB (5*4KB)
(e)
4 KB
(d)
80 Bytes
(c)
16-20 Bytes
(b)
Updated Net Data
10 Bytes
(a)
Tuple Size
Page Metadata (Header/Footer)
Whole Page
File System (ext3) [24]: 4KB write yields 20KB written
SSD Device: GC – Page Migrations Each write caues avg. 3 pysical writes

Figure 1: Write amplification caused by update-intensive workloads.

To illustrate the impact of the above claims consider the scenario depicted in Figure 1. Under update-intensive workloads, for instance TPC-C, *70%* of all update operations modify less than 10 bytes per page (100 bytes for LinkBench - Appendix A). (a) Hence, just a few bytes per tuple change. (b) Nonetheless, on pages with traditional NSM-layout (slotted pages), the whole tuple is modified. (c) In the general case, it is written to a new offset within the free space on that page, yielding a change of 20 or more bytes. Furthermore, the whole page header (and footer) must be updated accordingly, amounting to an additional change of approx. 80 bytes on that page. (d) Due to the block-device interface used by most storage systems, the whole 4-8KB DB page is written back to its original location. (e) If a file system is underneath, multiple blocks are changed. A single 4KB database write results in 3.4 file system (ext3) writes and a total of 20KB written [24]. (f) Those cause an additional on-device write overhead (due to garbage collection (GC) and wear-leveling (WL)), when written to a Flash device: one file system host write results in 1 to 5 physical Flash writes. *Thus, a 10 byte update entails a 4-8KB in-place page update, causing a write amplification of 400 – 800 times.* Moreover, the above example holds not only for small updates, but also for other types of modifications ranging from deletions (mark delete) to version invalidations in multi-version DBMSs.

On the one hand, this is partly due to the block-device interface to traditional storage devices. It is dominated by the HDD model, where only whole blocks at immutable physical addresses can be read or written. On the other hand, modern storage technologies in particular Flash memories exhibit very different characteristics: (i) erase-before-overwrite - once written a Flash page cannot be overwritten before the whole Flash block to which it belongs is erased; (ii) wear issues - a NAND Flash cell can be erased/written a certain number of times before it eventually wears out. Flash devices support the same block-device interface as HDDs, masking the properties of Flash storage, to ensure backwards compatibility and foster proliferation. Yet, *Flash memories do allow small appends to the still unused space of a page (see Section 3).* These small appends are Flash-friendly and avoid the erase-before-overwrite rule. Although this property relaxes the above *assumptions*, it is masked by the block-device interface and therefore cannot be utilized by the DBMS. Fortunately, several approaches proposing different Flash interfaces and deeper database integration have been proposed recently [16, 8, 20, 31, 6], demonstrating an impressive performance potential. More importantly, such approaches open the algorithmic and architectural opportunities to realize the above *claim*.

Many existing DBMSs use NSM page layouts. Hence, even with small modifications the whole tuple and the page header/footer are changed, essentially invalidating the whole page (Figure 1. c, b). *Changes to page-layout are essential for reducing the write amplification.* For this purpose, we propose to extend the NSM page-layout (see Section 6.1) to absorb modifications as update deltas (delta-records). They are appended to a reserved portion of the free space within the page (delta-record-area). In addition, only delta-records (instead of whole pages) can be written to Flash by a new *write_delta* command to realize the above claim.

The contributions of the paper are:
(I) We present an approach called In-Place Appends (IPA) that reduces the write amplification 2 to 3 times by allowing small updates to be performed in-place.
(II) IPA can be selectively applied to specific database objects (e.g. frequently updated tables or indices) without extra DBA overhead. The rest of the DB objects are not impacted. We describe an IPA advisor that suggests appropriate parameter values based on the current workload, minimizing DBA complexity and the number knobs.
(III) We propose revising the NSM page-layout to accommodate updates and modifications accordingly.
(IV) We implemented In-Place Appends under Shore-MT with NoFTL [16, 19] (see Section 5). Although native Flash approaches such as NoFTL or CORFU [6] are a natural choice, making implementation easy and efficient, the idea is also viable for traditional SSDs with marginal extensions.
(V) The implementation and evaluation are performed on real hardware (the OpenSSD Flash-research platform) and a Flash emulator.
(VI) Compared to In-Page Logging [21, 29] IPA performs 51% to 62% less reads, 23% to 62% less writes and 29% to 74% less erases on a range of workloads (Sections 2.1, 8.3).
(VII) The performance evaluation under typical update-intensive workloads indicates: (a) 35%-85% reduction of both erase operations and garbage collection write-overhead; (b) up to 45% improvement in transactional throughput; (c) 15%-60% decrease in I/O response time and up to 35% decrease in transactional response time.
(VIII) The proposed approach can be used on different types of Flash memories (SLC, MLC, 3D). We propose a technique called *write_delta* (not to be confused with partial writes on SLC) to write only the update deltas.

The remainder of this paper discusses related work in Section 2 and provides details about the write process of Flash memory and NoFTL in Sections 3 and 5. Section 4 outlines the approach to perform byte-size updates with the effect of a local overwrite, while Sections 6 and 7 present the design and implementation details. Section 8 provides an analysis of OLTP benchmarks and discusses the evaluation results.

2. RELATED WORK

2.1 Comparison of IPL versus IPA

The motivation and ideas behind In-Page Logging (IPL) in [21, 29] are similar to ours. To reduce the write-amplification caused by the garbage collector Lee et al. [21] modify the DBMS buffer and storage managers to avoid in-place updates. All updates are logged per page and placed in in-memory log sectors. As soon as a log sector of a particular page becomes full or the page is evicted from the buffer the log records from the respective log sector are written into reserved Flash page(s) on the Flash block (erase unit), where the original page is located. Thus, the original page is left unchanged while its update log records are written to a new location on Flash. To ”re-create” the up-to-date version of the page, two (or more) read I/Os must be performed: (i) read of the original page; and (ii) read of one or more Flash pages with the corresponding update logs. Hereafter, the update logs are applied to the original page on-the-fly and the up-to-date version of a page is placed in the buffer. Once the size of the update logs becomes greater than the reserved space on a Flash block, all pages on that particular block must be merged with their updates and written to a new block (through the special *merge* operation). The merge operation causes additional I/O as well as CPU overhead. In the worst case under *skew*-conditions, even though only few hot pages in a Flash block are frequently updated, all pages of the block must be read, transferred to the host and written back to a new Flash block during *merge*.

Both IPL and IPA reduce the number of page invalidations on Flash and consequently the write-amplification caused by the garbage collector. Both achieve this by replacing typical in-place page updates with keeping the original version of the data and augmenting it with update logs (similar to REDO logs). *The key difference is that our approach stores the delta-records (update logs) on the very same Flash page containing the original version of the data, instead of using separate Flash pages for them.* In other words, we perform in-page logging solely within a particular Flash page. To do this, we relax the commonly accepted axiom that Flash memory always follows the *erase-before-rewrite principle*. By using the techniques described in Sections 3 and 4) and a clever page layout, we can over-write the original Flash page without executing a previous erase operation.

The differences between both approaches have the following effect:

1. Under IPA no additional READ I/Os are needed to re-create the up-to-date page version since all update logs are co-located on the same physical Flash page as the original data. In contrast, under IPL each fetch operation requires reading at least one additional Flash page storing update logs (those are co-located on the same erase unit as the original Flash page, but are stored on separate Flash pages). The latter results in doubling the read I/O load. In read-heavy workloads with 70%-90% reads, this becomes a performance issue.
2. In-Place Appends do not suffer from *merge* operations present in IPL. They introduce both I/O and CPU overheads. Furthermore, *merges* in IPL are blocking, i.e. the block with the full log region must be merged before the next log record can be stored. As a result, under IPL the unused SSD space can not be utilized to delay/amortize expensive *merge* operations and corresponding erases. In other words, the CPU and I/O overheads (1 merge = N*READ I/O + N*WRITE I/O + ERASE) produced by *merges* are constant for a particular workload and IPL configuration, and independent of the drive's free space, be it 5% or 95%. Moreover, *merges* are always done in foreground and cannot be executed as background requests in order to utilize on-board CPU resources and internal SSD parallelism. *Although, a larger log region in IPL would reduce the write-amplification, this would simultaneously increase the required space and the read-amplification.*
3. IPL reserves at least three times as much physical Flash space as IPA does. Furthermore, under certain conditions IPL can experience poor space utilization – for instance, under workloads dominated by small random updates since those rarely accumulate in memory-resident log-sectors before page evictions.
4. IPL is applied to the complete database, while IPA can be applied selectively (using *regions*, Sect. 5) only to certain database objects, dominated by small-size updates.
5. IPL is well-suited for SLC Flash[1] with the ability to perform partial writes (assuming that the Flash memory can perform small sector writes in 512B granularity [21]). However, on MLC Flash with large physical pages and without partial programming, the disadvantages and overheads of IPL become more significant. In-Place Appends are suitable for SLC and MLC and benefit from the trend of increasing Flash page sizes.

[1]SLC was widely spread at the time [21] was proposed.

2.2 Multi-versioning and delta-writes

[13] has proposed an approach for reducing the write-amplification on Flash storage caused by tuple-version invalidation in multi-version databases (MV-DBMS). Typically, MV-DBMSs perform the invalidation of previous versions of a data item in-place, i.e. an invalidation timestamp is set physically on the tuple-version record. This modification causes a significant write-amplification, since a change of just a timestamp causes the whole page to be overwritten. The proposed SIAS approach avoids the in-place invalidation of records by maintaining a singly-linked list of record versions. The new version of a record contains a pointer to the previous one, thus implicitly invalidating the latter.

[32] describes an optimization approach for indexing in MV-DBMS by using "DeltaBlocks", Flash SSDs and append-based storage techniques. The main goal of the "DeltaBlock" approach is to ensure that the "latest version of any record is retrievable within at most one HDD disk I/O and one (or a constant number of) SSD I/Os" [32]. For this purpose [32] stores the deltas of consecutive versions of a particular record on fast SSD storage. This approach reduces the number of HDD I/Os by substitutining them with SSD I/Os. [32] relies on typical FTL-based SSDs. It addresses neither the traditional write process on Flash (as we do using ISPP and appends within the already written physical Flash page), nor the reduction of SSD GC overhead.

[7] suggests the utilization of in-page logs to avoid in-place updates (costly random updates) in MV-DBMS. However, [7] does not utilise techniques such as ISPP or IPA for reducing GC on wear-prone memories.

2.3 Re-Programming on Flash and ISPP

Our approach is based on the utilization of Incremental Step Pulse Programming (ISPP) for performing In-Place Appends on the original Flash pages. The ability to over-write Flash pages without a previous erase operation is rarely discussed and utilized. To the best of our knowledge only Cai et al. in [35] propose an approach "Correct-and-Refresh", which allows to mitigate retention errors on Flash using this property. The charge on the floating gate of a Flash cell leaks away over time. Those cell charge leakages can lead to bit errors (retention errors) while reading the Flash page. Correct-and-Refresh uses the ISPP to over-write (re-program) the original Flash pages in-place in order to restore the desired charge level on Flash cells. The Flash pages are periodically read, the bit errors are corrected on-the-fly using ECC, and consequently the corrected data is programmed to the *same* Flash pages.

More details about ISPP and the physical working-principles of Flash memory can be found in [34, 4].

2.4 Native Flash

An overview of Flash storage properties is given in [10, 28], design and performance tradeoffs are discussed in [3]. A comprehensive survey of Flash Translation Layer schemes can be found in [25, 11].

Modern Flash SSDs are backwards compatible with HDDs. The compatibility is provided through an additional abstraction layer (FTL). Although it allows for fast and easy replacement of HDDs with SSDs, the black-box architecture of Flash storage produces additional overhead and unpredictable performance variations. In recent years both academia and industry have proposed multiple approaches to eliminate those drawbacks.

Bonnet et al. in [8] proposed the bimodal SSDs. They are claimed to operate in two modes: (i) if a DBMS issues "constrained" I/O patterns (i.e. no in-place updates, no random writes) the SSD uses minimal FTL, providing only block-level mapping and wear-leveling; (ii) for all "unconstrained" I/O patterns the SSD switches to traditional FTL. While for DSS with read-mostly workloads and batch updates such bimodal SSDs would be beneficial, in OLTP systems most of the time the DBMS issues unconstrained I/O patterns.

Kang et al. [20] have suggested to enrich the responsibility of on-device FTLs in order to utilize internal out-of-place updates for providing DBMS transactional atomicity. Through a modified I/O interface the DBMS notifies the SSD about transaction demarcation (begin and end of transactions). The latter maintains an internal transactional table and keeps obsolete versions of modified database pages belonging to running transactions for the recovery process. Although the approach allows for significant reduction of write I/O (i.e. no need for WAL), it has several drawbacks: (i) due to the additional memory and computational overheads for on-board SSD resources it is suitable only for systems with low level of concurrency; (ii) disadvantages of the block-device interface and the black-box architecture of SSDs are not considered in this solution.

Ouyang et al. [31] introduce a new I/O primitive "atomic-write", which is utilized by the MySQL InnoDB storage engine to eliminate redundant writes performed for recovery purposees (i.e. DoubleWrite buffer). The approach, however, does not allow to execute several "atomic-writes" simultaneously, which is an issue in highly concurrent OLTP systems. The problems resulting from the traditional black-box architecture of SSDs are not addressed either.

There are also several proposals that suggest usage of raw Flash memory for objects with special I/O access patterns. CORFU [6] provided the design of a shared log implemented on top of a cluster of raw Flash units. [30] proposes the use of native Flash in the context of buffer management.

In this work we have used the NoFTL architecture [16] for implementation and evaluation of the proposed approach. It attempts to solve all the main problems resulting from the black-box architecture of modern SSDs by giving the DBMS full and exclusive control over the raw Flash memory (Figure 3). This architecture provides the uniform solution for all workloads and DB-objects with different characteristics.

3. THE WRITE PROCESS OF NAND FLASH

It is commonly accepted that once a page is written on Flash memory it cannot be overwritten in-place. To store a new version of a page at the same location one must read into the buffer all valid pages of the same block (usually 32-256 pages), erase the complete block, and afterwards write back the valid pages along with the new version of the updated page. Since reading into the buffer, erasing and writing back multiple Flash pages for each update would be far too costly in terms of time and wear-out of Flash blocks, all modern SSDs implement some kind of out-of-place update strategy. Each updated page is written to a new clean location and the logical to physical mapping scheme points to the most recent version of the page. The garbage collector periodically collects obsolete versions of pages and erases (victim) Flash blocks, thus ensuring enough free space for further write requests. Valid pages from the victim blocks are rewritten by the garbage collector to new locations (page migrations).

It is worth looking a bit deeper. The core of Flash memory is the Flash cell - a floating gate transistor. A cell can store in its floating gate a negative charge. The amount of this charge represents the bit-code of the stored information. In Flash composed of Single-Level-Cells (SLC) each cell stores only one bit: no charge or less than half of maximum charge represents bit 1, while a charge greater than half the maximum represents bit 0. In Multi-Level-Cell (MLC) Flash each cell stores two bits, i.e. between no charge and maximum charge of the cell we distinguish now four different charge levels for bit-codes 11, 10, 01, 00. Similarly a Triple-Level-Cell (TLC) Flash cell is capable of storing eight different charge levels, coding the states 111, 110, 101, ... 001, 000.

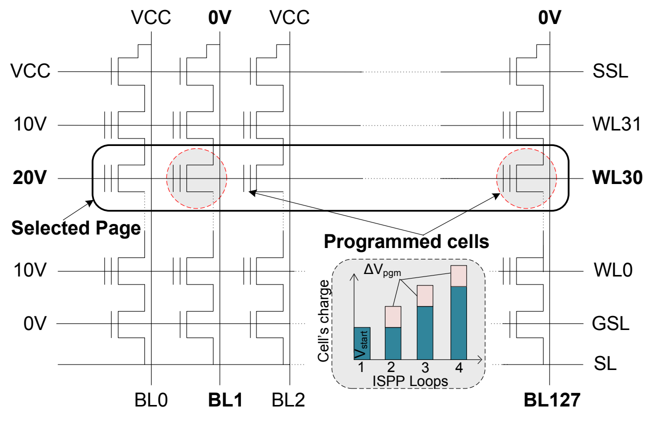

Figure 2: Organization of SLC NAND Flash memory and ISPP programming

Flash memory is a matrix of Flash cells, combined in a lattice where rows represent wordlines and columns bitlines (Figure 2). Cells which share the same wordline build Flash pages (one or more depending on the architecture and whether it is SLC or MLC Flash). The number of cells on the same bitline determines the size of the Flash block, while the number of bitlines determines the size of the Flash page. This organization is the main reason why Flash memory is read and programmed in page units, and erased in block units. This is done by manipulating the voltage on the word- and bit-lines.

SSDs use an Incremental Step Pulse Programming scheme (ISPP) to program on Flash [34, 27], i.e. to increase selectively the charge of individual cells (Figure 2). ISPP programs cells (if needed) in multiple iterations. After each program iteration the charge in the cells is controlled, and if it is lower than the desired charge it is increased in a subsequent iteration. The procedure repeats until each cell has the desired charge level. Therefore, the charge of a cell can be increased individually at any time up to its maximum charge without the need of an erase operation.

However, it is not possible to decrease the charge of an individual cell in the Flash page. While ISPP can increase the charge of individual cells in any Flash page, the erase operation resets all cells belonging to a block (32 - 256 pages).

Now it becomes clear why traditional SSDs must perform out-of-place updates. An in-place update is only possible if the change would require every changed bit of the new value to go from low charge (logical 1 for SLC) to high charge (logical 0 for SLC). Since the probability of this happening for an arbitrary update is virtually zero, SSDs are forced to perform out-of-place updates.

In our approach we exploit the fact that ISPP can increase the charge level of any particular cell at any time to achieve the effect of an update in-place for small updates.

4. IN-PLACE APPENDS ON A FLASH PAGE

The conventional SSD strategy of writing out-of-place and garbage collecting obsolete versions of pages results in massive write amplification. *In our approach, whenever a page is evicted from the buffer, it is written back into the same location on Flash without a previous erase.* To achieve this we reserve a small portion of the original page that is left unprogrammed by ISPP, i.e. the corresponding cells remain without charge. Small updates are appended in the form of redo log entries (delta-records) in the reserved area of the same page. We call this the delta-record area (see Section 6.1). The in-place append is possible since all changed cells get their charges increased. The already programmed cells (by the initial program operation) can either be (i) checked by ISPP (e.g. "Correct-and-Refresh" for correction of retention errors [35]); or (ii) might be completely omitted from the programming process. The latter is similar to a programming of logical "1", i.e. the charge level of those cells is left unchanged by setting the voltage on their bitlines to VCC - the so-called "self-boosting" approach [34].

In their simplest form, small updates to a page can be stored as offset-value pairs, placed in the delta-record area of the same page. When flushed, it is written out to its original Flash location (Section 6.2). Next time it is read into the buffer, the delta-record is applied. This can be repeated as long as there is free delta-record area space.

In-Place Appends is applicable to different types of Flash memories[2]*, namely SLC, MLC/eMLC and TLC in 3D NAND organizations.* The specifics of how the IPA is applied on those Flash types are described briefly in Appendix C. For MLC Flash there are two possibilities to apply IPA: either the pSLC or the odd-MLC modes. MLC Flash maps every wordline N to two pages: the odd-numbered LSB-page $(2N - 1)$ and the even-numbered MSB-page $(2N + 2)$. In case of *pSLC* the MLC Flash is used in pseudo SLC mode, i.e. only LSB-pages are utilized. In the *odd-MLC* mode the whole capacity of MLC Flash is utilized, but IPA may only be applied to LSB-pages, while every MSB-page write is still performed out-of-place. For example, IPA can be applied on the LSB-page 59 on wordline WL30, whereas the MSB-page 62 on the same wordline can only be updated out-of-place.

5. BRIEF OVERVIEW OF NOFTL

The approach presented here was implemented and evaluated as part of NoFTL (Figure 3) [16, 19]. We show that NoFTL or alternative open architectures like CORFU [6] are a natural choice, which makes the implementation of IPA easy and efficient. However, IPA can also be implemented in conventional SSD architectures (see Section 7). The main concepts behind NoFTL are to integrate Flash management into the DBMS (Figure 3). The access to rich DBMS metadata (e.g. object type, format, update frequency) allows for significant optimization of the Flash management functionality. Moreover, NoFTL allows native DBMS subsystems (e.g. buffer management and concurrency control) to benefit from controlled data placement and knowledge of the internal Flash organization.

```
CREATE REGION rgIPA ( MAX_CHIPS=8, MAX_SIZE=512M,
                      MAX_CHANNELS=4, IPA_MODE = pSLC);
CREATE TABLESPACE tsIPA (REGION=rgIPA, EXTENT = 128K);
CREATE TABLE T( t_id NUMBER(3) ) TABLESPACE tsIPA;
```

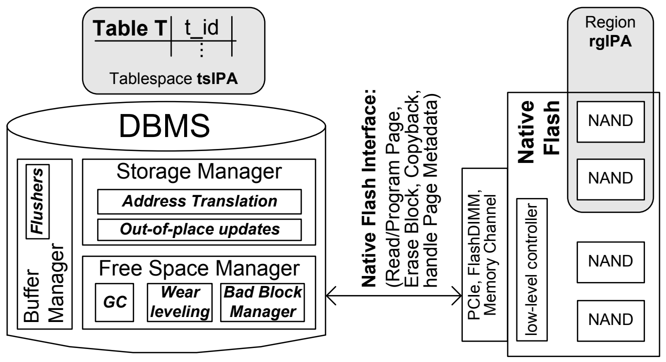

Figure 3: NoFTL Architecture with Regions supporting In-Place Appends.

[19] introduces the notion of NoFTL *regions*, which allows to apply the In-Place Appends selectively to specific DB objects (or sets thereof). [19] claims that NoFTL *regions* do not introduce additional complexity to the DBA since those can be coupled to existing logical storage structures.

On MLC Flash, *regions* make it possible to apply both IPA modes (pSLC and odd-MLC) simultaneously. For instance, write-intensive tables or indexes dominated by small updates can be placed in a region (e.g. region *rgIPA*, Figure 3), which uses pSLC as IPA mode. The less write-intensive objects can be placed in another region, which utilizes IPA in an odd-MLC mode. Read-only objects or objects dominated by large updates can be placed in yet another region, which does not utilize IPA.

6. IPA – DESIGN AND IMPLEMENTATION

The proposed approach reflects two main factors: the first one is the average updated data size on a DB page by the time it is flushed to stable storage, while the second factor relates to the physical properties of Flash memory. Taking into account both factors we propose a $[N \times M]$ scheme for performing IPA. N is the maximum number of possible subsequent In-Place Appends (delta-records), while M is the maximum number of changed bytes per update. If more than M bytes were changed or N delta-records were already appended, the page is written out-of-place, while its old version is marked as obsolete and can be garbage collected.

6.1 Database Page-Layout

[2]Since we have no direct contacts to Flash manufacturers it is virtually impossible to test IPA on all possible technological variations of the main Flash types.

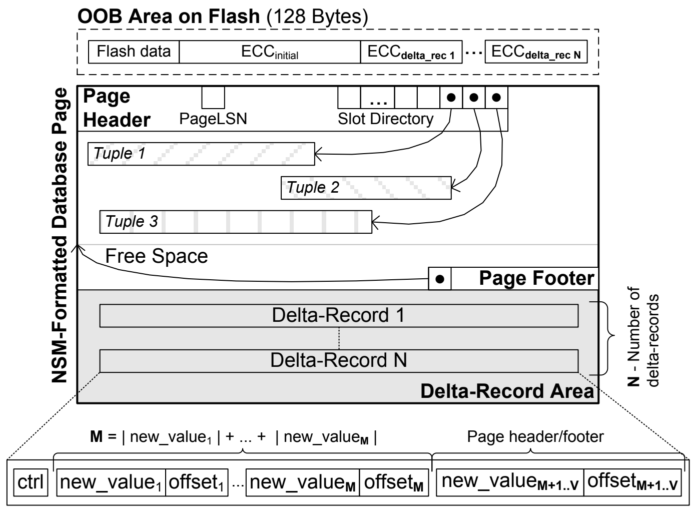

Figure 4: Database page-format for IPA on Flash

Handling small updates requires a revision of the traditional NSM database page-layout (Figure 4). Under IPA we reserve a certain portion of the free space area of each database page called *delta-record area.* Every small in-page update is tracked, and the modified data eventually is placed in a *delta-record.* Hence, *each small in-place update is transformed to an in-place append, which can be programmed using ISPP (Sections 3 and 4), avoiding out-of-place writes or expensive Flash erases.*

Delta-record format. Each delta-record (Figure 4) comprises a control byte and a number of <*new_value,offset*> pairs covering modifications in the page body and in the page metadata (*page_header* and *page_footer*). The *control byte (ctrl)* indicates the existence of a corresponding delta-record. The <$new\_value_{1..M}$,$offset_{1..M}$> pairs account for the modified bytes in different tuple/attribute values: $new\_value_{1..M}$ – represents the modified byte at a given position, specified by its two-byte $offset_{1..M}$ from the beginning of the page. For more details consider the example below. Only modifications of byte-granularity are supported, thus $|new\_value| = 1(byte)$. According to the $[N{\times}M]$ scheme, the number of <$new\_value_{1..M}$,$offset_{1..M}$> pairs per delta-record cannot exceed M (and ideally is equal to M):

$$M \geq \left(number\_of\_pairs = \sum_{i=1} |new\_value_i| \right)$$

Realistically, $M \leq 125$ $(bytes)$, as established in Section 8.2 and in Appendix A. The choice of byte-granularity for modifications instead of tuple-attribute granularity results in higher space efficiency and simplicity.

The NSM page metadata comprises the so called *page_header* and *page_footer.* They reflect the slot table, containing the offsets of each record within the page, transactional information such as PageLSN, and free/used space. It is essential to consider metadata modifications as they occur upon each page update. In the simplest update case only the PageLSN and a fixed-length attribute value change, in the general case all of the above may change. Modifications to page metadata are tracked with the same value/offset mechanism, hence <$new\_value_{(M+1)..V}$,$offset_{(M+1)..V}$>. The byte-level tracking for page metadata allows for space efficiency. For example, the PageLSN is 8-bytes in Shore-MT . However, only the least-significant byte changes very frequently, while the most significant bytes are seldom modified as they reflect the LogFileID. Hence, using change tracking in byte-granularity we record only the modified, least-significant bytes instead of the complete PageLSN. Alternatively, the delta-record may contain the complete page metadata. However, our experiments indicate that the byte-level tracking mechanism reduces the delta-area size by 49% for a [2×3] scheme.

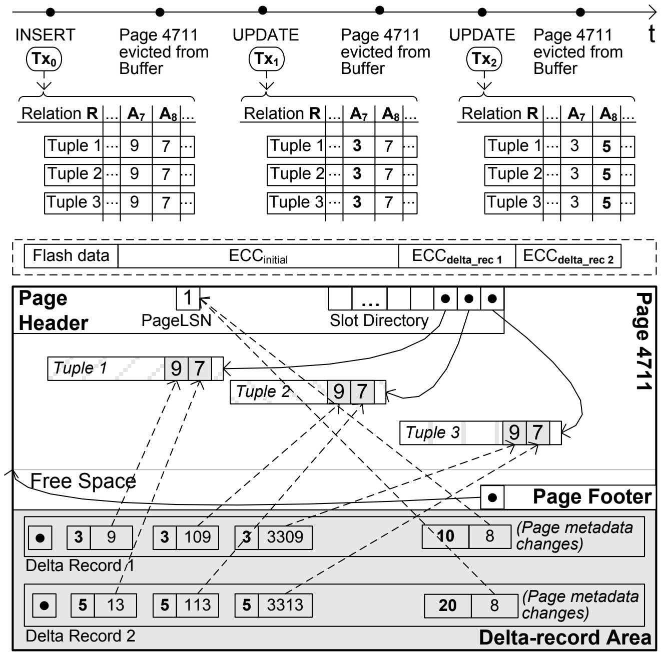

Figure 5: Example: Database page-format for In-Place Appends on Flash. The page contains two delta records, recording changes to three tuples.

Sizing the delta-record area. According to the $[N{\times}M]$ scheme, a database page can host up to N delta-records, each modifying at most M bytes. Obviously, a change of 1 byte yields the creation of a 3-byte <*new_value,offset*> pair in the delta-record (1-byte *new_value* and 2 bytes *offset*[3]). The delta-record size is therefore: $1 + 3M + 3V$. Hence, the size of the delta-record area is: $N{*}(1{+}3M{+}3V)$. V denotes the number of modified bytes in page metadata. In practice, $V \leq 12$ for Shore-MT under OLTP workloads. The above rule is used as a *space reservation scheme* to allocate the delta-record area on every database page. The choice of N, M and V can be automated by an IPA advisor (Sect. 8.4).

Example. Assume a [2×3] configuration for the $[N{\times}M]$ scheme under TPC-C. The delta-record area will have two record slots, while a delta-record can host modifications of at most 3 bytes, yielding at most V=12 bytes of metadata changes. Hence the delta-record size is $1 + 3*3 + 3*12 = 46$ bytes. The delta-record area size is $2*(1{+}3*3{+}3*12) = 92$ bytes. For a 4KB database page this implies 2.2% reserved space. Consider table R and the three transactions Tx_0, Tx_1 and Tx_2 executed in serial order (Figure 5). Tx_0 creates R and inserts Tuples 1 through 3 initializing the values of A_7 and A_8 of all tuples to 9 and 7, respectively. Page

[3]Assuming a max. DB-page size of 64KB, else 3 bytes.

4711 is allocated, formatted and filled. Tx_0 commits successfully and commit/end LSN is 1. Assume page 4711 got evicted; it is written out-of-place since IPA is not applicable for newly allocated pages. Tx_1 modifies all tuples, changing the value of attribute A_7 to 3. Thus, the page 4711 is fetched into the database buffer and modified accordingly. Attribute A_7 is a fixed length integer and only the least-significant byte is modified. Hence, for each tuple a 3-byte value/offset pair is created. The PageLSN is modified to the commit record LSN of Tx_1 (e.g. 10) - 1 byte value/2 byte offset. The free space and the slot tables remain unchanged, since a fixed-length attribute is updated. This is why the $<$*new_value*$_{(M+1)..V}$,*offset*$_{(M+1)..V}>$ pair <10,8> is created. Assume page 4711 got evicted again, which forces the creation of delta-record 1. Analogously, Tx_2 modifies all tuples by changing A_7 to 3. On the next eviction of page 4711 the delta-record 2 is created.

6.2 Page Operations

In this section we describe how the DBMS handles delta-record pages: what operations are defined on them, how tuples are fetched and modified, how such pages are buffered, and how the storage manager controls the writing process.

The page is fetched into the DB buffer. As the page is read from Flash we first check whether it was already updated in-place: the *control_bytes* are read to determine the actual number of delta_records. If none are available, we are done and the page can be passed on for further processing. Alternatively, if delta-records are present, they are applied in forward order: the page data (tuples) and metadata (header and footer) are updated by replacing the changed bytes.

The page is modified in the DB buffer. Updates are performed as usual in-place, but changes on the page body are tracked. If the total size of those changes is smaller than the available delta-record area, the offsets of changed bytes are kept in new delta-record(s). Otherwise, if there is insufficient delta-area space, the page is marked to be written out-of-place upon eviction, and further changes are not tracked. Note that a delta-area overflow does not cause any additional writes, it merely disallows IPA on that page until it is evicted.

The size calculations are performed as follows. Assume the page has been freshly fetched and contains N_E delta-records from the last time it was evicted ($N_E < N$). N_E can be determined by the *control_bytes*. The maximum number of possible changes is: $C_p = (N - N_E) * M$. In other words, at most C_p bytes can be modified and at most $(N - N_E)$ additional delta-records can be inserted. Therefore, as long as total number of changed bytes U does not exceed C_p, IPA are allowed. $\lceil (U - C_p)/M \rceil$ delta-records are added and $(U - C_p)$ offset-value pairs are created. However, once $U > C_p$ we mark the page to be *written out-of-place* and stop tracking further updates.

The page is evicted and flushed to stable storage. When the page gets evicted from the buffer pool we check whether it can be updated using in-place appends or should be written out-of-place. If the page must be written out-of-place, we reset the delta-record area and write the up-to-date page from the buffer to new location on Flash memory. If the in-place append can be performed, we first complete the current delta-record(s) with the new values of the changed bytes (the offsets of those bytes are already in delta-record). Thereafter these delta-records are written to the Flash storage using the **write_delta** command. **Write_delta** appends delta-records to the very same physical Flash page the original database page resides on.

This approach introduces negligible or no overhead for the database management system. First, the NoFTL architecture allows for applying IPA only to selected database objects using *NoFTL Regions* [19] (e.g. solely for the STOCK table in TPC-C, or for 3 from 4 tables in TPC-B), where the small updates of few bytes dominate. Second, the replacement of few bytes upon fetching, as well as the tracking of changes during updates in the buffer produces minimal computational overhead. Remember that IPA targets only small updates. In the schemes used for TPC benchmarks ([2×3] or [2×4]) at most two consequent In-Place Appends are "dealing" with less than 40 bytes in total (body + metadata). In case of social network (graph) workloads this value can increase to 125 bytes. Replacing so many bytes has shown no noticeable performance degradation.

Remaining DBMS functionality. *Please note that the rest of the database functionality (e.g. recovery, locking, etc.) is NOT impacted by IPA.* The DBMS is operating, as usual, performing modifications in-place on buffered pages. The approach touches mainly the process of evicting the pages from and fetching them into the buffer. Consider, for instance, the rollback operation in a WAL-based system. A DB-page P_{db}, stored on a physical Flash page P_{flash}, gets evicted from the buffer pool. P_{db} is dirty and contains uncommitted modifications from an active transaction T_x. If those changes conform to the $[N{\times}M]$ scheme, they are transformed into In-Place Appends (delta-record) and appended to P_{flash}. Assume now that T_x aborts and its changes must be rolled back. P_{flash} is read, the delta-record is applied, and thus the up-to-date version of P_{db} is re-created and placed in a free buffer frame. Now the regular UNDO process on P_{db} can be performed, i.e. the corresponding UNDO log records are read and applied. Given enough delta-record area space the byte-changes of these UNDO modifications are placed there, otherwise P_{db} is marked for out-of-place write. If later on P_{db} is evicted again, it will be re-written back to P_{flash} using IPA, even if further modifications have been performed, given there is enough delta-record area space to host them.

Flash ECC and Page OOB Area. The proposed approach requires an adjustment of the physical error correction (ECC) strategy on Flash. NoFTL allows for a simple adjustment offering two possible alternatives.

The first one shifts the responsibility of performing ECC to the storage layer of the DBMS. In NoFTL the DBMS has direct control over the physical addresses of Flash pages and the access to the page's OOB area. Based on the $[N{\times}M]$ scheme, ECC can be computed in at most N steps. The ECC code of a page will then comprise the page body ECC ($ECC_{initial}$, Figure 4), and an ECC for every delta-record (ECC_{delta_rec1} ... ECC_{delta_recN}, Figure 4). Those codes can also be physically appended in-place to the page OOB area utilizing ISPP. When the page is retrieved, the ECC codes are applied to the corresponding sections of the page, so that bit errors in the page body and delta-record area can be detected and corrected.

Under the second alternative, the adjusted ECC algorithm is running on the on-board SSD controller. In this case, however, the controller requires information about the used

$[N \times M]$ scheme, to apply the resepective ECC parts accordingly, which requires an additional control command.

7. IN-PLACE APPENDS - I/O COMMANDS

To integrate the IPA in the I/O stack, we propose to add the *write_delta()* command as a first class citizen besides *read* and *write*. A DBMS *write* is performed as an out-of-place write on Flash, if the block has already been written, while a *write_delta()* performs an in-place append:

write_delta(LBA, offset, delta_length, delta_bytes[ ]);

Hence, a single delta-record of a certain *delta_length* can be written to a DB-page identified by its logical address *LBA*. The parameter *offset* specifies the byte-offset from the beginning of the DB-page, where the delta-record should be placed, whereas *delta_bytes[ ]* is the payload of the record. The *write_delta* command is general-purpose and independent of the proposed page layout. It can serve different $[N \times M]$ schemes as those can be configured per database object. Please note that delta-writes should not be confused with the so called *partial writes*, available on some SLC NAND chips. Delta-writes can be implemented on conventional SSD and on Native Flash [15].

8. EXPERIMENTAL EVALUATION

8.1 Testbed

We implemented In-Place Appends under NoFTL in Shore-MT[4]. Shore-MT is a recognized storage engine supporting ACID transactions, ARIES-type logging, indices, buffer management, as well as an implementation of the standard TPC benchmarks. Our performance evaluation is carried out using the OpenSSD hardware and the real-time Flash Emulator[16], which has been validated against OpenSSD [18].

The OpenSSD Jasmine board [1] is an open Flash-SSD research platform with two MLC Flash modules from Samsung and a total volume of 64GB. The Flash memory is controlled by an ARM controller from Indilinx and is connected to the host through a SATA2.0 interface. The evaluation of IPA on MLC Flash and OpenSSD was performed using *NoFTL Regions* configured in both possible modes: pSLC and odd-MLC (see Section 4). Additional technical details and evaluation specifics regarding the OpenSSD Jasmine board are pointed out in Appendix D. IPA performance numbers on OpenSSD and the Flash emulator differ due to the missing OpenSSD parallelism (point 1 in Appendix D) as well as the small buffer size (point 3 in Appendix D) causing an I/O bound system and higher effect of IPA.

To extend the evaluation profile with I/O parallelism and larger database buffer sizes, experiments have also been performed on the real-time Flash emulator. Those tests were done on an Intel Xeon server with 32 E7-4830 2.13 GHz CPU-cores (64 Threads, 64 KB L1 cache, 256 KB L2 cache and 24 MB L3 cache) and 128 GB RAM under Ubuntu 12.04.03 LTS 64-bit with kernel 3.8.0. The emulated Flash storage comprises 16 SLC chips, 10% over-provisioning and is managed using a page-level mapping scheme under NoFTL.

8.2 Analysis of update-intensive benchmarks

A detailed analysis of typical update-intensive workloads (TPC-B, TPC-C and LinkBench [5]) substantiates our claim about small update sizes. This analysis has been performed under MySQL InnoDB (LinkBench) and Shore-MT (TPC-B/-C). We recorded live traces of the above benchmarks running for 2 (TPC-C/-B) and 4 (Linkbench) hours. The DB-sizes are roughly 50GB, while the buffer sizes vary from 10% to 90% of the initial DB-sizes.

Number of changed bytes (^{1}net data, ^{2}gross data)	Percentiles		
	TPC-B^1	TPC-C^1	LinkBench^2
≤ 3	10-th	55-th	0-th
≤ 7	62-th	83-th	0-th
≤ 20	99-th	88-th	5-th
≤ 100	99-th	93-th	40-th
≤ 125	99-th	94-th	50-th

Table 1: Update-sizes in TPC-B/-C and LinkBench (Buffer 75%, eager eviction strategy).

In TPC-C and TPC-B with buffer size 75% of the initial DB-size (and the default eager eviction strategy) more than 80% of all DBMS writes change effectively less than 20 bytes in total on any particular page (Table 1). Regardless of those small update sizes, modern DBMSs write the whole database page, resulting in write-amplification of several hundred times. Further analyses of those benchmarks, their transaction profiles and update CDFs for different buffer sizes and eviction strategies are provided in Appendix A.

Social graph workloads (LinkBench[5]) have update-behavior similar to traditional OLTP workloads, and thus can significantly benefit from IPA. However, the update-sizes are larger and the data set does not typically fit in memory. The update CDF for LinkBench is presented in Figure 10 in Appendix A. 47%-76% of all updates modify less than 125 bytes gross per page (including header and footer) with buffer sizes of 20% to 90% of the DB-size. Even such workloads are suitable for IPA. Note that the net update-sizes (excluding page header and footer) are smaller than 100B.

8.3 Performance Comparison of IPL and IPA

To evaluate IPA against In-Page Logging we have used the IPL simulator described in [21], the original source code and I/O traces were kindly provided by the authors. Furthermore, we have compared both approaches based on the very same configuration used in the original IPL paper: (i) logical DB page size – 8KB; (ii) SLC Flash with 64 2KB physical pages per erase unit and 512B partial writes; (iii) the in-memory log sector per logical DB page is equal to the size of a partial write (512B); and (iv) the size of log region per erase unit is 8KB. For a fair comparison we used both: the original traces from [21] as well as newly recorded OLTP traces. The reason for using new traces was twofold: (i) the original traces do not include page fetch events (i.e. they do not contain any READ I/O), and thus do not allow analyzing the read overhead; (ii) the original traces only reflect TPC-C, whereas our goal was evaluating under further OLTP workloads. Thus, we have recorded traces for TPC-C, TPC-B and TATP benchmarks running in Shore-MT, configured with In-Place Appends. Each of those traces has been replayed on the original IPL simulator. The comparative results are shown in Table 2. Detailed description of the *I/O Read and Write Amplifactions* formulae for both approaches is provided in Appendix B.

IPA outperforms IPL (Table 2) by performing 60%, 52% and 51% less reads; 62%, 23% and 37% less writes; as well as

[4]https://sites.google.com/site/shoremt/

	TPC-B		TPC-C		TATP	
	IPA	IPL	IPA	IPL	IPA	IPL
I/O Write Amplific.	0.54	1.43	0.94	1.22	0.64	1.01
I/O Read Amplific.	1.01	2.54	1.06	2.20	1.01	2.07
Erases	35 958	137 962	41 486	58 294	11 873	30 155

Table 2: Comparison of IPA to IPL.

74%, 29% and 61% less erases under the same TPC-B, TPC-C and TATP traces, respectively. Under the above settings IPL reserves 6.25% of the space, while IPA configurations [2×3] and [2×4] require at most 2%. A larger IPL log region would reduce the write-amplification (less *merges*), but would increase the required reserved space on each block and the read-amplification (e.g. a log region of 16KB would require already 12.5% of Flash space to be reserved for update logs and would result in more than 3x read-amplification). *Read reduction (51% – 60%) substantiates claim 1, Section 2.1, while the reduction of writes (23% – 62%) and erases (29% –74%) substantiate claim 2. The space requirements of both approaches substantiate claim 3, Section 2.1.*

8.4 Performance Evalution of IPA

[N×M] **Scheme Selection and Space Utilization.** The value M is chosen based on an analysis of update-sizes of a given OLTP workload (see Table 1 and Figures 7, 8 and 9 in Appendix A). Thus, for TPC-C the natural choice is $M = 3$, since the majority of all updates modify less than 3 bytes of net data (50% - 75% depending on the buffer size). For TPC-B we opted for $M = 4$, since more than 50% to 90% of all updates change exactly 4 bytes of net data.

The choice of N - the maximum number of delta-records per page - is influenced by several factors. (i) Flash specifics (see Appendix C): the value of N for SLC Flash is significantly higher than for MLC Flash. (ii) Space consumption: the larger the value of N, the more space is reserved for the delta-record area, and hence the bigger the database size. (iii) Workload properties (locality, skew) influence the update frequency and distribution over database pages. Throughout the experiments we have selected N to be 2 or 3 primarily based on (i). We assume that this choice matches every MLC or 3D Flash. No issues with increased wear or interference errors were observed throughout the tests on OpenSSD Jasmine with MLC Flash chips. At most, 1 bit error per 16KB Flash pages was detected and corrected (in both cases, with and without IPA).

A sensitivity analysis of M and N for TPC-C and LinkBench is presented in Table 3. Note that the reported increase of relative space consumption (red) for a particular [N×M] scheme represents the worst case, i.e. IPA is applied to all DB-objects. However, using *NoFTL regions* IPA can be applied selectively (only to DB-objects dominated by small-size updates) to decrease the actual space overhead significantly. Throughout all experiments under different workloads we observed an increase of database sizes from 1% to at most 14%, when IPA is applied to all DB-objects.

Moreover, IPA allows decreasing the size of the over-provisioning area without a loss of performance. This is especially true for SSDs that use hybrid mapping schemes (like FASTer [23], where over-provisioning defines the *log area*). The over-provisioning area absorbs all incoming write requests and when full the *FTL merge* operations merge updated data with the data stored in the *data area*. Since IPA results

TPC-C (75% buffer, 4KB pages, M=updated bytes in net data)																
		M=3			M=6			M=10			M=15			M=20		
N	1	34.7	1.1	-32	40.4	1.3	-38	42.7	1.6	-41	42.8	2.0	-41	42.8	2.4	-41
	2	46.1	2.2	-43	53.6	2.7	-51	56.8	3.3	-55	56.8	4.0	-55	56.8	4.7	-55
	3	51.6	3.4	-49	60.1	4.0	-58	63.7	4.9	-62	64.0	6.0	-62	64.0	7.1	-62
	4	54.9	4.5	-52	64.2	5.4	-62	68.0	6.5	-66	68.3	8.0	-65	68.2	9.5	-64

Linkbench (75% buffer, 8KB pages, M=updated bytes in whole page)					
		M=100		M=125	
N	1	28.2	3.7	33	4.6
	2	35.4	7.3	43	9.2
	3	37.9	11.0	47	13.8

Table 3: Fraction of update IOs performed as IPA [%] (in black), space overhead (in red) [%], and reduction in erases per host write [%] (in blue) for different $N \times M$ schemes under TPC-C and LinkBench.

in less out-of-place writes the over-provisioning area is populated much slower, which postpones the expensive merge operations for a longer period. Consequently, the size of the over-provisioning area can be reduced. Therefore, the space overhead due to the delta-record area may be compensated by lower over-provisioning.

IPA Advisor. An IPA advisor automates the choice of the appropriate M, N and V values [17], letting the DBA weight the general optimization goals: (i) performance (for buffer size $\leq$ 50% depending on the workload and the data size); (ii) longevity – larger [N×M] result in less erases and page migrations; (iii) space consumption – effective cost/GB. The IPA advisor is based on a background DB log-file profiling mechanism, analyzing the current workload at run-time. This is possible since the DB-log contains all information regarding update sizes, frequencies or skew. In addition, under NoFTL, we can compute these per DB-Object. The newly computed [N×M] configurations determine the size of the delta-record area and can only be applied offline.

Legend. The performance results on both, OpenSSD Jasmine and the Flash emulator, are presented in Tables 4 - 10 and Figure 6. In Tables 6, 7, 8, 9, 10 the [0×0 *Absolute*] columns show the absolute values for experiments without In-Place Appends. Additionally, in Tables 6 and 8 the columns [2×N *Absolute pSLC/odd-MLC*] present the absolute values for experieements with IPA on OpenSSD using pSLC or odd-MLC mode. The columns [N×M *Relative [%]*] show the relative improvement of IPA with [N×M] over the corresponding [0×0 *Absolute*]. The first row *[Out-of-Place Writes vs. In-Place Appends]* in Tables 6, 7, 8, 9, 10 shows the ratio of out-of-place writes vs. In-Place Appends among all DBMS write requests for each particular scheme (not a relative change to the traditional approach, as the column names may suggest).

DB I/O Write Amplification. This type of write amplification (see Figure 1.d) is caused by the DBMS, since it performs writes in page-granularity, even if only a few bytes are modified on a page. By performing delta-writes in delta-record-granularity using the format described in Section 6.1 this write amplification can be reduced up to 2.8x under typical OLTP workloads (see Table 4). The detailed results for LinkBench with different buffer sizes 20% - 90% are presented in Table 5. The write amplification was calculated as follows: $WriteAmplification = \dfrac{Gross\_Written\_Data}{Net\_Changed\_Data}$

Benchmark	TPC-B (M=4)		TPC-C (M=3)		LinkBench (M=125)	
Buffer Size	75%	90%	75%	90%	75%	90%
IPA [2*M]	2.03	2.00	1.95	1.89	1.71	1.66
IPA [3*M]	2.83	2.77	2.54	2.47	1.83	1.75

Table 4: Write amplification reduction (x times) under TPC-C/-B and LinkBench: traditional approach (no IPA $[0\times0]$) vs. $[2\times M]$ and $[3\times M]$ schemes.

			NxM					
			1x100	1x125	2x100	2x125	3x100	3x125
Space overhead [%]			*3.67*	*4.59*	*7.35*	*9.18*	*11.02*	*13.77*
Reduction of DBMS write amplification [*x times*]	**Buffer**	**20%**	1.67	1.74	2.12	2.27	2.42	2.65
		50%	1.54	1.63	1.84	2.02	2.01	2.28
		75%	1.38	1.48	1.53	1.71	1.59	1.83
		90%	1.35	1.45	1.47	1.66	1.52	1.75

Table 5: Space overhead and reduction of DBMS write amplification in Linkbench (MySQL InnoDB).

Where $Gross\_Written\_Data$ equals $Host\_Writes{*}Page\_Size$ (without In-Place Appends, $[0\times0]$), while for $[N\times M]$ scheme it equals $(Out\_of\_Place\_Writes{*}Page\_Size)+(Delta\_Writes{*}Delta\_Record\_Size)$.

	0x0 Absolute	2x4 Absolute *pSLC*	2x4 Relative *pSLC* [%]	2x4 Absolute *odd-MLC*	2x4 Relative *odd-MLC* [%]
Out-of-Place Writes vs. **In-Place Appends**			33/**67**		50/**50**
Host Reads	3 779 926	5 567 683	+47	4 902 012	+30
Host Writes	2 028 626	3 054 292	+51	2 395 514	+18
GC Page Migrations	605 047	152 973	-75	316 128	-48
GC Erases	15 839	7 238	-54	7 745	-51
Page Migrations per Host Write	0.2983	0.0501	-83	0.1320	-56
GC Erases per Host Write	0.0078	0.0024	-70	0.0032	-59
Transactional Throughput	260	383	+48	316	+22

Table 6: TPC-B benchmark on OpenSSD: traditional approach (no IPA $[0\times0]$) vs. $[2\times4]$ scheme in modes pSLC and odd-MLC.

	Buffer 10%			Buffer 20%		
	0*0 Absolute	**2x4** Relative [%]	**3x4** Relative [%]	**0*0** Absolute	**2x4** Relative [%]	**3x4** Relative [%]
Out-of-Place Writes vs. **In-Place Appends**		33/**67**	24/**76**		35/**65**	25/**75**
Host Reads (4KB)	61 805 479	+33	+44	55 028 782	+37	+47
Host Writes (4KB)	34 652 703	+32	+41	34 784 847	+32	+43
GC Page Migrations	38 374 571	-48	-58	37 920 807	-42	-52
GC Erases	1 045 622	-55	-64	1 040 622	-51	-59
GC Page Migrations per Host Write	1.1074	-61	-70	1.0902	-56	-67
GC Erases per Host Write	0.0302	-66	-75	0.0299	-63	-71
Response Time [ms] READ I/O (4KB)	2.43	-46	-52	2.64	-41	-50
WRITE I/O (4KB)	0.70	-34	-40	0.69	-30	-41
Transactional Throughput	4956	+31	+41	5223	+34	+42

Table 7: TPC-B on Flash Emulator: traditional approach (no IPA $[0\times0]$) vs. $[2\times4]$ and $[3\times4]$ schemes.

On-Device Write Amplification. In-Place Appends allow up to N updates to the same Flash page without

	0x0 Absolute	2x3 Absolute *pSLC*	2x3 Relative *pSLC* [%]	2x3 Absolute *odd-MLC*	2x3 Relative *odd-MLC* [%]
Out-of-Place Writes vs. **In-Place Appends**			49/**51**		70/**30**
Host Reads	4 977 335	6 390 032	+28	5 671 727	+14
Host Writes	1 347 515	1 768 552	+31	1 524 552	+13
GC Page Migrations	422 753	79 718	-81	230 497	-45
GC Erases	7 151	2 862	-60	3 819	-47
Page Migrations per Host Write	0.3137	0.0451	-86	0.1512	-52
GC Erases per Host Write	0.0053	0.0016	-70	0.0025	-53
Transactional Throughput	25	37	+46	28	+11

Table 8: TPC-C benchmark on OpenSSD: traditional approach (no IPA $[0\times0]$) vs. $[2\times3]$ scheme in modes pSLC and odd-MLC.

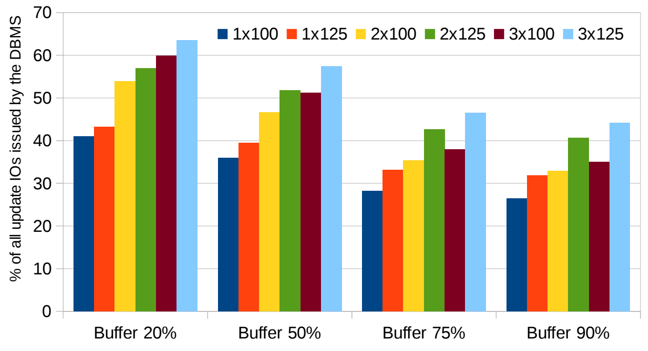

Figure 6: Fraction of update IOs performed as in-place appends in Linkbench (MySQL InnoDB).

performing out-of-place writes. The first row *[Out-of-Place Writes vs. In-Place Appends]* in Tables 6, 7, 8, 9, 10 and on Figure 6 shows that 30% to 76% of the DBMS writes (*[Host Write]*) can be performed as In-Place Appends, throughout all the workloads both on OpenSSD and the Flash emulator. This reduces the number of invalidated Flash pages, allowing the garbage collector to perform space reclamation less frequently (see Figure 1.f). In other words, for the same amount of DBMS write requests (*[Host Write]*) the garbage collector performs less page migrations and erase operations. Thus, under TPC-B we observed a reduction of *[GC Page Migrations per Host Write]* of up to 83% on OpenSSD and up to 70% on the Flash emulator; for TPC-C 86% and 62% respectively. The reduction of *[GC Erases per Host Write]* was 70%/75%, 70%/62% on OpenSSD/emulator for TPC-B and TPC-C respectively. The higher performance gain of In-Place Appends on the OpenSSD compared to the Flash emulator is caused by the 1.5% buffer, as well as, the limited parallelism of the OpenSSD board (see Section 8.1).

IPA with different DBMS buffer sizes. We varied the DBMS buffer size between 10% and 90% of the initial DB-size (Table 9). Naturally, the transactional throughput increases with larger buffer sizes. Furthermore, the relative performance improvement due to IPA decreases, and with large buffer sizes (75% and 90%) it practically disappears. However, IPA has an unchanged positive effect on write-amplification and longevity of Flash SSD even with large buffers: consider the rows *[GC Page Migrations per Host Write]* and *[GC Erases per Host Write]* in Table 9. The

		Buffer 10%		Buffer 20%		Buffer 50%		Buffer 75%		Buffer 90%	
		0x0 Abs.	**2x3** Rel. [%]	**0x0** Abs.	**2x3** Rel. [%]	**0x0** Abs.	**2x3** Rel. [%]	**0x0** Abs.	**2x3** Rel. [%]	**0x0** Abs.	**2x3** Rel. [%]
Out-of-Place Writes vs. **IPAs**			51/**49**		51/**49**		54/**46**		56/**44**		56/**44**
Host Read I/Os (4KB)		44 190 546	20.93	25 120 600	25.89	3 275 550	11.44	614 639	4.43	279 258	0.44
Host Write I/Os (4KB)		36 735 568	15.73	39 530 323	16.25	51 570 434	9.41	62 627 983	9.81	64 345 377	0.54
GC Page Migrations		25 648 523	-38.39	27 886 888	-36.00	37 521 497	-31.74	46 874 908	-29.08	47 558 375	-28.51
GC Erases		939 888	-40.83	1 018 624	-39.51	1 357 349	-37.67	1 676 376	-34.83	1 713 844	-33.77
GC Page Migrations per Host Write		0.6982	-46.76	0.7055	-44.95	0.7276	-37.61	0.7485	-35.42	0.7391	-28.89
GC Erases per Host Write		0.0256	-48.87	0.0258	-47.97	0.0263	-43.03	0.0268	-40.65	0.0266	-34.13
Response Time [ms]	READ I/O (4KB)	0.45	-29.05	0.77	-31.60	3.90	-31.07	8.44	-21.34	9.10	-2.89
	WRITE I/O (4KB)	0.53	-22.01	0.53	-21.36	0.53	-19.17	0.54	-17.88	0.53	-15.38
Transactional Throughput		865	15.33	1 001	15.42	1 480	6.28	1 984	1.22	2 191	0.21

Table 9: TPC-C: traditional (no IPA $[0{\times}0]$) vs. $[2{\times}3]$ schemes with large buffer pools (eager eviction).

		Buffer 10%		Buffer 20%		Buffer 50%		Buffer 75%		Buffer 90%	
		0x0 Abs.	**2x10** Rel. [%]	**0x0** Abs.	**2x10** Rel. [%]	**0x0** Abs.	**2x30** Rel. [%]	**0x0** Abs.	**2x40** Rel. [%]	**0x0** Abs.	**2x40** Rel. [%]
Out-of-Place Writes vs. **IPAs**			41/**59**		44/**56**		51/**49**		63/**37**		67/**33**
Host Read I/Os (4KB)		59 417 771	22.28	39 383 139	16.29	4 462 332	5.33	676 580	0.50	265 543	3.19
Host Write I/Os (4KB)		29 559 808	19.68	28 591 074	20.16	11 767 036	4.43	8 486 996	3.46	8 802 867	3.25
GC Page Migrations		19 513 382	-55.59	16 595 099	-40.27	5 027 818	-30.98	2 877 442	-20.07	2 982 080	-19.52
GC Erases		731 549	-53.95	670 807	-46.07	227 215	-36.13	142 339	-21.63	148 312	-19.10
GC Page Migrations Per Host Write		0.6601	-62.89	0.5804	-50.29	0.4273	-33.91	0.3390	-22.75	0.3388	-22.06
GC Erases per Host Write		0.0247	-61.53	0.0235	-55.12	0.0193	-38.84	0.0168	-24.25	0.0168	-21.65
Response Time [ms]	READ I/O (4KB)	0.24	-32.08	0.30	-19.46	0.41	-16.95	0.43	-19.30	0.64	-11.53
	WRITE I/O (4KB)	0.51	-28.27	0.49	-21.56	0.43	-12.33	0.43	-7.87	0.56	-8.21
Transactional Throughput		915	15.37	1 291	6.96	2 220	3.26	2 360	1.06	2 259	3.67

Table 10: TPC-C: traditional (no IPA $[0{\times}0]$) vs. $[2{\times}M]$ schemes with large buffer pools (non-eager eviction).

garbage collection overhead is reduced by 29%-43% in the experiments with buffer sizes 50%-90%.

Although with increasing buffer sizes the number of *[Host Read I/Os]* (Table 9) decreases rapidly, the number of *[Host Write I/Os]* increases. *Why is that? Why does the DBMS write even with 90% buffer size?* Both InnoDB and Shore-MT implement *eager buffer eviction* strategies: flushing dirty pages eagerly when a threshold is reached (e.g. 12.5% hard-coded in Shore-MT). In addition, InnoDB and Shore-MT employ an *eager log-space reclamation* strategy (e.g. when 25%-50% of the log-space is consumed in Shore-MT). The combination of both strategies aims at reducing recovery times [33] and alleviating the checkpoint-overhead. Thus, larger buffers result in a higher throughput and consequently in more dirty pages in the buffer and larger log-files. The background writers flush therefore more dirty (cold) pages to stable storage. Modern main-memory DBMSs implement different strategies [12, 26], targeting another tradeoff between throughput and recovery times.

To analyze the update accumulation effects with large buffers, we turned off the eager eviction and eager log-space reclamation by setting extreme values for the respective thresholds in Shore-MT (75% and 100% respectively). Thereby, we observe significant reductions of *[Host Write I/Os]* with increasing buffer sizes (Table 10) as well as clear update accumulation effects (Table 11 and Figure 9 in Appendix A). For instance, with Buffer 10%, 80% of all updates change less than 6 bytes, while with Buffer 90% only 4% of all updates do so. To account for the update accumulation effects, larger values of M are needed. Even, with configuration such as $[2{\times}40]$ we observe that with 90% buffer and "non-eager" eviction strategy at least 33% of all host writes can be performed as in-page appends, reducing the overhead of the garbage collection by more than 20% (Table 10). Yet, due to the large number of dirty pages and log volume the recovery times in those experiments increase by 4x on average.

Number of changed bytes (net data)	Percentiles				
	Buffer 10%	Buffer 20%	Buffer 50%	Buffer 75%	Buffer 90%
≤ 3	61-th	34-th	1-th	1-th	1-th
≤ 6	80-th	64-th	5-th	5-th	4-th
≤ 10	88-th	83-th	14-th	13-th	10-th
≤ 30	89-th	88-th	74-th	58-th	60-th
≤ 40	90-th	89-th	76-th	71-th	72-th

Table 11: TPC-C Update-sizes (non-eager eviction).

Under LinkBench we increase the M to 100 and 125, which results for $N = 3$ in about 14% higher space consumption, but allows to reduce the DBMS write-amplification 1.75x - 2.65x in experiments with buffer sizes 20% - 90% (Table 5).

I/O and Transactional Response Times. It is well known that the garbage collector overhead leads to unpredictable performance degradation on all Flash SSDs [10], since it interferes with host I/Os. By reducing the number of out-of-place writes and therefore the number of erases, response times of host reads and writes are improved. Lines *[READ I/O]* and *[WRITE I/O]* in Tables 7 and 9 show the average DBMS I/O latencies without In-Place Appends, and the relative improvement for the $[N{\times}M]$ schemes (with IPA). Clearly, the latency of *[READ I/O 4KB]* improves by up to 52% for TPC-B (in experiments with 1% buffer and on OpenSSD even more than 60%), by up to 32% for TPC-C. The maximum *[WRITE I/O 4KB]* latency improvements are 41% and 28% for TPC-B and TPC-C respectively.

Please consider the differences in response times for different buffer sizes. Those are due to an inherent tradeoff between waiting time (I/O contention for Flash) and execution time (GC overhead). On the one hand, larger buffer sizes allow for caching more hot pages, which improves the *[Transactional Throughput]*. The larger the number of submitted transactions, the higher the I/O rate, which increases the contention on 16 chips of the emulated Flash storage. As a result the average *I/O waiting time* as well as the resulting response time increase. On the other hand, IPA improve *execution time* of write I/Os, by reducing the on-device write amplification and the GC activity. Faster on-device processing of write requests results in a higher I/O throughput and decreases *waiting times* of host I/Os.

The lower I/O latencies impact the transactional response times. Since Shore-MT uses a steal/no-force policy the write requests are performed mainly by the background cleaners and checkpointing threads. Therefore, the transactional response times are influenced primarily by the *[READ I/O]* latencies. The impact varies, depending on the I/O profile of each transaction (write intensive, read-only) and its working set. Under TPC-B the response time of the *Account Update* transaction is decreased by roughly 30%. Under TPC-C the decrease varies between 3% and 23%.

Last but not least, the decreased response times of transactions allow the DBMS to perform more transactions in the same measurement interval. The line *[Transaction Throughput]* shows that the utilization of In-Place Appends results in up to 48%/42% and 46%/15% higher transactional throughput on OpenSSD/emulator under TPC-B and TPC-C.

Note, the improvement in all those values (response times and throughput) is the result of the reduced garbage collector overhead. It depends strongly on the parameters of Flash memory, such as size of over-provisioning and address mapping scheme. Throughout all experiments we used 10% over-provisioning area and a page-level mapping scheme, which is the most efficient for OLTP workloads. Typical SSDs have usually 7%-10% over-provisioning and use hybrid address mapping schemes. In those systems the positive effect of the reduced overhead of the garbage collector on the overall system performance is supposed to be higher.

Longevity of Flash Storage. Besides the improved system performance, In-Place Appends have a huge impact on the longevity of Flash storage. The wear-out limits of modern Flash memories (100.000 Program/Erase cycles for SLC, 10.000 for MLC, and 4.000 for TLC Flash) impact enterprise systems with OLTP-like workloads. The row labeled *[GC Erases per Host Write]* in Tables 6, 7, 8, 9, 10 indicates that different $[N \times M]$ schemes reduce the average number of erases per write request by up to 75% and 70% under TPC-B and TPC-C respectively. Consequently, the lifetime of Flash memories is significantly prolonged.

9. CONCLUSIONS

Under traditional update-intensive workloads, small updates dominate the write behavior: more than 70% of all updates change less than 10 bytes across all TPC OLTP benchmarks. These updates are performed in-place on the original database page and due to their random nature they result in random writes in page-granularity. Thus, a modification of less than 10 bytes net, yields an expensive write of a whole 4KB or 8KB database page back to stable storage. The result is a major overhead on Flash storage and a write amplification of several hundred times. The root cause are outdated architectural assumptions such as: (i) the alignment of the database unit of I/O to a page or (ii) the use of the HDD-tailored block-device interface to perform I/O.

In this paper we propose an approach that transforms those small in-place updates in small delta-records on the original page that are written out as In-Place Appends to Flash storage. IPA makes use of the commonly neglected fact that Flash memories can natively handle those as physical page appends through the use of various low-level techniques such as ISPP, given the charge of individual NAND cells is always increasing. Such techniques help avoiding expensive erase or out-of-place write operations. To be able to take advantage of those small appends, we propose extending the traditional NSM page-layout with a delta-record area that can host small updates. If only few bytes change we compute the update deltas. These are then appended to delta-records in the delta-record area, leaving the rest of the page unchanged. Thus, when the page is written back to storage only the new delta-records should be ISPP-programmed on the original Flash page, minimizing GC overhead and expensive erases. We also describe how DBMS modules such as buffer or storage manager must be adapted to handle operations on the modified page format. In addition, we propose an $[N \times M]$ scheme to control: (a) the space allocation on database pages and sizing of the delta-records; (b) the write behavior; (c) buffering and fetching operations.

The experimental evaluation is performed under Shore-MT, real Flash hardware and various update-intensive workloads such as standard TPC workloads. It leads to the following results. *First*, 33%-85% reduction in erase operations. Furthermore, we observe approximately the same reduction in garbage collector write-overhead. Hence, the proposed approach doubles the longevity of Flash devices by reducing wear and yields better performance. *Second*, 15%-60% lower read and write I/O latencies. *Third*, up to 45% higher transactional throughput. *Fourth*, better utilization of physical Flash space and the reduction of the over-provisioning area size, under unchanged database performance. *Fifth*, IPA can be selectively applied to individual DB objects, depending on their update properties. We propose an IPA advisor that can suggest appropriate values for N, M and V for the current workload, minimizing the additional DBA complexity and the number of required knobs. *Sixth*, implementation and evaluation are performed on real hardware as well as on a Flash emulator. IPA can be realized on traditional SSDs, by extending the block-device interface and the on-board controller functionality at the cost of lower performance compared to IPA under NoFTL. However, on-device write-amplification and longevity improvements compared to conventional SSDs will still be significant. *Seventh*, compared to In-Page Logging [21] IPA performs up to 62% less reads and writes and up to 74% less erases on a range of workloads. Last but not least, the write amplification is reduced 2x-3x. IPA can be used in combination with ISPP on a wide range of Flash and SSD devices, if native Flash and a technology similar to NoFTL regions are supported.

Acknowledgements

The authors are grateful to the anonymous SIGMOD reviewers for the insightful comments, which helped to significantly improve the quality of the presentation. We also like to thank Sang-Won Lee and his group for providing the original IPL code and Goetz Graefe for his remarks on earlier drafts of this paper. This research was supported under grants DFG "Flashy-DB" and "Software Campus" (01IS12054).

10. REFERENCES

[1] The openssd project. http://www.openssd-project.org/wiki/The_OpenSSD_Project, 2014.

[2] Samsung v-nand technology. http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf, 2014.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proc. USENIX ATC'08*.

[4] S. Aritome. *NAND flash memory technologies*. IEEE Press series on microelectronic systems. 2016.

[5] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proc. SIGMOD'13*.

[6] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. Davis. Corfu: A shared log design for flash clusters. In *Proc. USENIX NSDI'12*.

[7] B. Bhattacharjee, M. Canim, M. Hamedani, K. Ross, and A. Storm. Supporting transient snapshot with coordinated/uncoordinated commit protocol. US Patent 14/748,438, 2016.

[8] P. Bonnet and L. Bouganim. Flash device support for database management. In *Proc. CIDR'11*.

[9] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai. Program interference in mlc nand flash memory: Characterization, modeling, and mitigation. In *Proc. ICCD'13*.

[10] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. SIGMETRICS'09*.

[11] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *J. Syst. Archit.*, 55(5-6):332–343, 2009.

[12] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proc. SIGMOD*, 2013.

[13] R. Gottstein, I. Petrov, and A. Buchmann. Append storage in multi-version databases on flash. In *Proc. BNCOD'13*.

[14] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proc. ICDE'00*, pages 3–10, 2000.

[15] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. In-place appends for real: Dbms overwrites on flash without erase. In *Proc. EDBT'17*.

[16] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Noftl: Database systems on ftl-less flash storage. In *Proc. VLDB'13*.

[17] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Selective in-place appends for real: Reducing erases on wear-prone dbms storage. In *Proc. ICDE'17*.

[18] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Noftl for real: Databases on real native flash storage. In *Proc. EDBT*, pages 517–520, 2015.

[19] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Revisiting dbms space management for native flash. In *Proc. EDBT*, 2016.

[20] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min. X-ftl: Transactional ftl for sqlite databases. In *Proc. SIGMOD'13*.

[21] S.-W. Lee and B. Moon. Design of flash-based dbms: An in-page logging approach. In *Proc. SIGMOD'07*.

[22] S. T. Leutenegger and D. Dias. A modeling study of the tpc-c benchmark. In *Proc. SIGMOD'93*.

[23] S.-P. Lim, S.-W. Lee, and B. Moon. Faster ftl for enterprise-class flash memory ssds. In *Proc. SNAPI'10*.

[24] Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proc. FAST'13*.

[25] D. Ma, J. Feng, and G. Li. A survey of address translation technologies for flash memories. *ACM Comput. Surv.*, 46(3):36:1–36:39, 2014.

[26] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proc. ICDE*, 2014.

[27] R. Micheloni, L. Crippa, and A. Marelli. *Inside NAND Flash Memories*. Springer, 2010.

[28] R. Micheloni, A. Marelli, and K. Eshghi. *Inside Solid State Drives (SSDs)*. Springer, 2012.

[29] G.-J. Na, B. Moon, and S.-W. Lee. In-page logging b-tree for flash memory. In *Proc. DASFAA'09*.

[30] Y. Ou, J. Xu, and T. Härder. Towards an efficient flash-based mid-tier cache. In *Proc. DEXA'12*.

[31] X. Ouyang, D. W. Nellans, R. Wipfel, and D. Flynn. Beyond block i/o: Rethinking traditional storage primitives. In *Proc. HPCA'11*.

[32] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Exploiting ssds in operational multiversion databases. *VLDB*, 25(5):651–672, Oct. 2016.

[33] C. Sauer, G. Graefe, and T. Härder. An empirical analysis of database recovery costs. In *RDSS*, 2014.

[34] K.-D. e. a. Suh. A 3.3 v 32 mb nand flash memory with incremental step pulse programming scheme. In *Proc. ISSCC'95*.

[35] C. Yu, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Crista, O. S. Unsal, and M. Ken. Error analysis and retention-aware error management for nand flash memory. *Intel Tech. Journal*, 17(1):140 – 164, 2013.

APPENDIX

A. ANALYSIS OF OLTP BENCHMARKS

The statistics presented in Figures 7, 8, 9 and 10 were collected over the 2-hour (TPC-C/-B) and 4-hour (LinkBench) duration of the benchmarks, and show the cumulative distribution of update-sizes in bytes. For TPC-B and TPC-C the update-sizes correspond to net data (tuple data) changes, while for LinkBench they correspond to gross data changes (body + metadata). *In all benchmarks more than 93% of all write I/Os performed by the DBMS are updates to existing DB pages, and the remaining (1%-7%) are appends to new pages.* Due to the clear dominance of update I/Os we excluded the latter (appends to new pages) from the statistics, thus 100% of the Y-axes on the figures in the following sections cover solely update I/Os.

A.0.1 TPC-B Benchmark

Although TPC-B is officially obsolete its workload profile can be found in real world systems even nowadays. The benchmark has only one transaction, which simulates a deposit or withdrawal to/from some bank account. The transaction modifies all four tables in the TPC-B schema and is thus update-heavy. In three out of four tables the transaction changes only one numeric attribute of a single tuple, while a new tuple is appended to the fourth table.

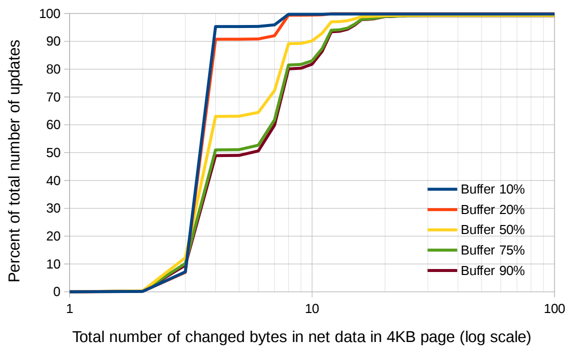

Figure 7: CDF of update-sizes in TPC-B in net data.

The distribution of update-sizes in TPC-B is shown in Figure 7. 50% - 90% of all update I/Os change only 4 bytes of net data per page, while more than 80% change 8 bytes or less. A short analysis of the database schema and transaction profile explains this. The *Account_Update* transaction appends one tuple to the *History* table (about 20 bytes net), and it modifies a numeric attribute value (4 bytes net) in a single tuple in each of the other three tables. Therefore, upon commit four pages are updated: the net data modified on 3 of them is at most 4 bytes, whereas in the forth page it amounts to 20 bytes. Since the cardinality of *Branch* and *Teller* tables is significantly smaller compared to the *Account* table (1:10:100000 respectively), those are usually completely buffered over the benchmark duration. Under the steal/no-force buffer policy employed by Shore-MT, pages of these tables are flushed to stable storage predominantly by background page cleaners and during checkpoints. Due to its cardinality the *History* table plays an insignifcant role, since it is a kind of "logging" table, which is seldom queried. All write I/Os to the *History* table are therefore appends to new pages (about 2% of all write I/Os).

Thus, the lion's share of update I/Os goes to the *Account* table, and in most cases those updates change only 4 bytes of net data, i.e. changes of only one transaction - one numeric attribute (*Account_Balance* $+= \Delta$). Due to the large cardinality and random access pattern only a small number of *Account's* pages can accumulate updates of multiple transactions, while in buffer.

A.0.2 TPC-C Benchmark

The TPC-C benchmark emulates an order-entry environment. The update-size statistics (see Figures 8, 9) show that about 70% of all update I/Os change less than 6 (eager eviction) and 40 (non-eager eviction) bytes of net data per page. According to previous research [22] and our own analysis, in the TPC-C benchmark the *STOCK* table clearly dominates the write behavior. The table is updated only by the *NewOrder* transaction (the backbone of the workload). Each *NewOrder* transaction modifies on average 10 random tuples

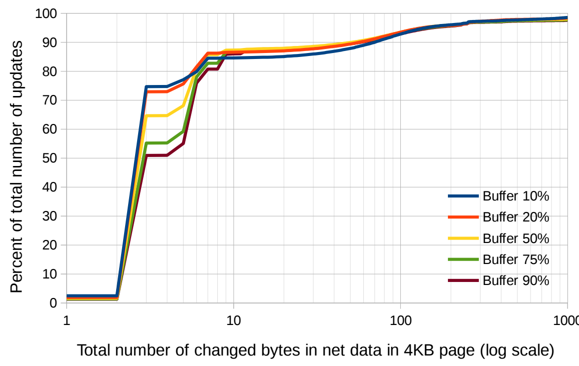

Figure 8: Cumulative distribution of update-sizes in TPC-C in net data (default eager eviction strategy).

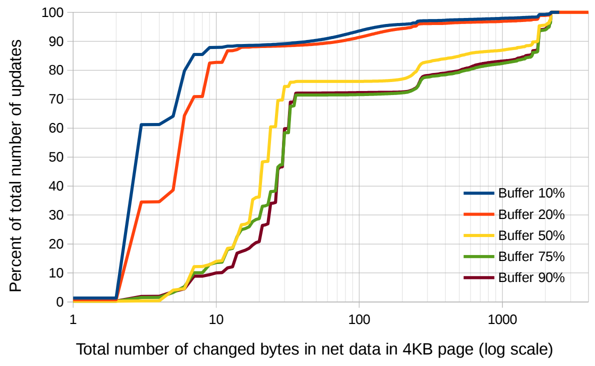

Figure 9: Cumulative distribution of update-sizes in TPC-C in net data (non-eager eviction strategy).

in the STOCK table. Three numeric attributes per tuple are modified: *(i) S_QUANTITY* $+= \Delta_1$; *(ii) S_YTD* $+= \Delta_2$; *(iii) S_ORDER_CNT += 1 or S_REMOTE_CNT +=1.* According to the specification $\Delta_{1,2}$ are usually less than 10, therefore in all three numeric fields typically only the least significant byte is changed. Therefore, each *NewOrder* transaction changes on average 10 data pages (due to the randomness of selected rows), modifying 3 bytes net on each page. Moreover, updates on the *District*, *Warehouse* and *Customer* tables modify a single or multiple numeric fields (except for 10% of Customers where also the *C_DATA* attribute is changed).

The high access skew of the workload (75% of accesses go to 20% of data [22]) allows for efficient buffering of hot pages. Thus, given a sufficient buffer size (e.g. 20%) the majority of write I/Os are caused by eviction of cold dirty database pages from the buffer, which rarely can accumulate multiple updates while being cached. Hence, the dominance of small-sized write I/Os.

A.0.3 LinkBench Benchmark

LinkBench [5] is a benchmark that emulates a social network workload. It was designed in 2013 by the Database Enginieering Team at Facebook for the purpose of evaluation and testing of DBMSs used for storing the social graphs (e.g. Facebook production data). The development of the workload was based on the comprehensive analysis of the large-scale social graph workloads captured from the pro-

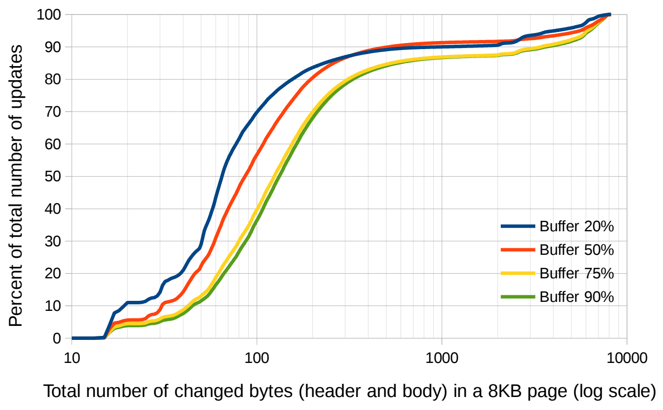

Figure 10: CDF of update-sizes in LinkBench.

duction databases at Facebook. The data scheme consists of three relations representing objects (nodes in the graph), associations (directed edges between nodes) and the number of associations. The workload is read-intensive (2.19:1 ratio of read to write queries) and is generated by executing the mix of ten operations on the graph.

Interestingly, the average payload size of the objects is less than 90 bytes, while the one of associations is less than 12 bytes (almost half of the associations do not have payload at all). Over a third of all updates on objects and associations do not change the payload size, and modify only one or few numeric fields (e.g. timestamp, version, etc.). The majority of the remaining update queries change the payload sizes only slightly. The cumulative distribution of update sizes is presented in Figure 10. About 70% of all updates change less than 100 bytes on 8KB DB-pages in experiments with buffer sizes 20%, and less than 200 bytes in experiments with larger buffer sizes (50%, 75%, 90%)

B. FORMULAE FOR IPL VERSUS IPA

The following expressions hold for the settings (Flash memory, DBMS) from the original IPL paper [21]. The *[I/O Write Amplification]* for IPL was calculated as:

$$WA\_IPL = \frac{\left(\begin{aligned}\#merges * 15 * 4io + \#imlog\_full * 1io \\ + \#page\_evictions * 1io\end{aligned}\right)}{\#page\_evictions * 4io}$$

Here, each *merge* operation results in writing out 15 8KB DB-pages, each resulting in programming 4 2KB physical Flash pages. When the in-memory per-page log-buffer gets full (*imlog_full*) it is written out (partial write of 512B has the same latency as a write of a whole 2KB Flash page). Similarly, when the DB-page is evicted from the buffer the corresponding log buffer is flushed. The number of those 2KB I/Os is compared to the single I/O the DBMS without IPL is assumed to perform (writing out 8KB logical page corresponds to 4 writes of 2KB chunks). The *[I/O Read Amplification]* for IPL is equal to:

$$RA\_IPL = \frac{\#page\_fetches * 2 * 4io + \#merges * 16 * 4io)}{\#page\_fetches * 4io}$$

Whenever the DBMS under IPL fetches the 8KB logical page, in addition to reading this page from Flash (four physical Flash pages) the whole log region (8KB) on the same erase unit must also be read. Therefore, the read load doubles. Furthermore, each *merge* operation requires reading and transmitting to the host (for merging purpose) the complete erase unit (i.e. 15 logical pages and log region).

The number of erases equals the number of IPL *merges*.

For IPA the corresponding formulas are described below.

$$WA\_IPA = \frac{\left(\begin{aligned}\#write\_deltas * 1io + \#out\_of\_place\_writes * 4io + \\ \#gc\_page\_migrations * 4io\end{aligned}\right)}{\#page\_evictions * 4io}$$

Upon eviction of a dirty page from the buffer, depending on the number of changed bytes the DBMS using In-Place Appends writes either the page as a whole (in an out-of-place manner on Flash) or only the delta record will be appended to the original Flash page. The size of the delta record is less than 100B, however we calculate here one 2KB I/O. The overhead of the traditional garbage collector is measured in the number of performed page migrations.

$$RA\_IPA = \frac{\#page\_fetches * 4io + \#gc\_page\_migrations * 4io}{\#page\_fetches * 4io}$$

As already mentioned, in case of IPA no additional read I/Os are required, when a logical DB-page is fetched. In order to migrate the valid pages from the victim block the garbage collector needs to read them first. Note, however, that in case of IPA the read and write I/O overheads caused by the garbage collector are internal Flash-device I/Os, i.e. no data transfer to or from the host is happening. Conversely, under the IPL approach to *merge* one erase unit its whole content must be first read into the host, then merged and then transferred to the Flash again, which increases the delay caused by *merge* operations.

C. IPA ON SLC, MLC AND 3D NAND

In-Place Appends can be applied to different types of Flash memory, namely SLC, MLC, eMLC and TLC in 3D NAND organizations. The specifics of those Flash types are described briefly below.

C.1 In-Place Appends on SLC Flash

IPA can be applied without specific limitations to SLC Flash. The reason is that the difference (distance) between different threshold voltages (indicating different logical bitcodes of the Flash cell: 1 and 0) is large enough to compensate small deviations. Such deviations may appear due to program interference (parasite capacitance-coupling), while (re-)programming the Flash-page (appending the delta-record). Some SLC Flash manufactures allow performing so-called *partial writes*, i.e. a Flash-page can be programmed incrementally in equally sized chunks (usually 512B). Note that IPA is not based on SLC partial writes. In contrast to partial writes, IPA allows to flexibly vary the number (N) and the size (M) of programmed/appended chunks to the Flash-page. Furthermore, IPA can be applied on other types of Flash memory (MLC, 3D).

C.2 In-Place Appends on MLC Flash

MLC modes. For MLC Flash there exist two possibilities to apply IPA: either the pSLC or the odd-MLC modes. MLC Flash can be dynamically configured in the so-called *pseudo-SLC (pSLC)* mode, for which the Flash capacity is reduced by half since only LSB-pages[5] are used. The benefit of using IPA in pSLC mode is twofold: (i) the GC overhead is reduced more than twice; and (ii) I/O latencies are lower since the programming time of LSB pages is significantly less than that of MSB-pages. Under the *odd-MLC* mode, the whole MLC Flash capacity is utilized, however, IPA are only applied to LSB-pages[5] (odd numbered pages), whereas MSB-pages[5](even numbered pages) still need to be programmed in standard out-of-place manner.

Program Interference Errors. The MLC Flash is more susceptible to program interference errors than SLC, due to the shorter distances between different voltage thresholds. To minimize those errors MLC manufacturers suggest programming Flash pages within a block in an incremental order (in-order programming). This order ensures that the cells on a particular wordline can only be influenced by an interference from programming a MSB-page on the successor wordline [9].

IPA does not cause program interference errors on MLC Flash in *pSLC* mode. Small updates on MLC Flash can cause program interference errors, however they are solved as follows by IPA on MLC Flash in *odd-MLC* mode.

Initially, all pages are programmed in order, hence there is no extra programming interference. If now IPA is applied on a page, an interference can occur on neighboring cells, which are located on adjacent pages. These are pages that lie on the predecessor and successor wordlines. For example, if the LSB page 59 on wordline WL30 is updated by IPA (see Figure 2), interferences can occur on pages on WL29 (LSB page 57 and MSB-page 60) and WL31(61, 64)[5]. It is very important to emphasize that such interferences impact ONLY the delta-record areas not the page body[6]. What's more, those voltage shifts in cells of delta-record areas of LSB-pages do not result in bit errors. This is because by reading LSB pages the Flash controller differentiates only between two voltage thresholds (having large distance between them). In contrast, bit-errors are possible in delta-record areas of neighboring MSB-pages, since there are four different thresholds. However those are simply ignored, since IPA is not applicable for MSB-pages, i.e. MSB-pages are always written out as a whole in a standard out-of-place manner. Therefore, a small update in the reserved area of page 59 does not incur bit errors to the reserved areas of pages 57 and 61, but does so on pages 60, 64, which are simply ignored.

Efficiency of *pSLC* and *odd-MLC*. Even though in odd-MLC mode In-Place Appends are applied only to half of the physical Flash pages, its performance is not necessarily twice as low as on SLC Flash or in pSLC mode. If the DB page size is larger than a physical page, then the DB page comprises multiple adjacent physical pages (MSB and LSB). The efficiency of odd-MLC mode is similar to SLC or pSLC since IPA always go to the physical LSB page. For example, if the database page size is 16KB (a standard today) and the size of Flash page is 8KB (also the most common case), then each database page comprises two adjacent physical pages (MSB and LSB). In this case, IPA can be applied to all logical DB pages, since all appends are performed on the LSB pages.

[5] Every wordline on MLC Flash contains two pages an LSB/odd and a MSB/even page. Thus wordline N (Figure 2) contains the LSB page (2N-1) and the MSB-page (2N+2). LSB/MSB pages are not shown in Figure 2 since it depicts SLC Flash.

[6]The program interference occurs on neighboring cells when the current cell is programmed with a certain charge, i.e. increase of cell's charge can cause a small voltage shift on neighboring cells. In other words, whenever no charge is applied to a cell, there is negligible or no program interference

C.3 In-Place Appends on 3D NAND

The 3D NAND is the current trend in Flash memory, addressing multiple technological issues: minimization and scaling as well as program interference. First, 3D Flash architectures allow to increase memory density (and thus capacity) by adding more layers without shrinking NAND cells. Second, interference issues are negligible in 3D Flash, which significantly prolongs the endurance, for instance through the usage of Flash CTF (Charge Trap Flash). According to Samsung their 3D V-NAND chips are: "Bitline Interference Free" and "Wordline Interference Almost Free" [2]. *Therefore we assume that IPA can be applied using SLC/pSLC or odd-MLC techniques.*

D. OPENSSD JASMINE

The following technical details of the OpenSSD Jasmine [1] board (Figure 11) need to be pointed out:

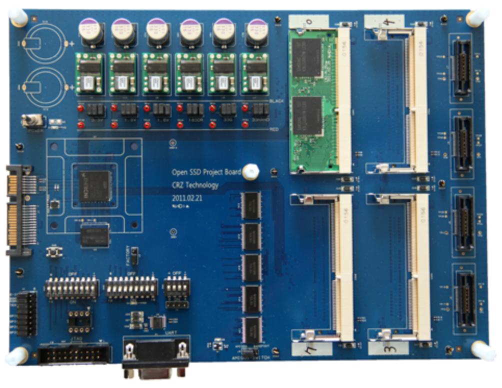

Figure 11: OpenSSD Jasmine with MLC Flash [1].

1. Although OpenSSD is said to support SATA NCQ (as stated in [1]), this support is not implemented by default. We could not retrofit it with NCQ due to the inability to obtain the specification of the proprietary controller. The provided board firmware tries to mitigate this issue by using write-cache to execute incoming write I/Os in parallel on different chips. However, without full-fledged NCQ there is no means to parallelize read I/Os, which dominate in OLTP workloads. Although the board has 8 dual-die packages and can theoretically execute up to 16 concurrent requests, its effective host-level I/O parallelism equals one I/O at a time.
2. The OpenSSD Jasmine board does not allow to program the Flash controller and the ECC engine. Furthermore, it does not provide an access to the OOB area of the Flash pages. Therefore, we were not able to modify the hardware ECC/EDC implementation. As a workaround this default implementation was turned off and an adjunct BCH-ECC was implemented in the DBMS.
3. Due to the 4GB RAM capacity of the dedicated test machine hosting board the evaluation on OpenSSD has been performed with a DB buffer size of 1GB (1.5% of DB size).