Modeling the Performance of Algorithms on Flash Memory Devices

Kenneth A. Ross
IBM T. J. Watson Research Center and Columbia University
rossak@us.ibm.com, kar@cs.columbia.edu

ABSTRACT

NAND flash memory is fast becoming popular as a component of large scale storage devices. For workloads requiring many random I/Os, flash devices can provide two orders of magnitude increased performance relative to magnetic disks. Flash memory has some unusual characteristics. In particular, general updates require a page write, while updates of 1 bits to 0 bits can be done in-place. In order to measure how well algorithms perform on such a device, we propose the "EWOM" model for analyzing algorithms on flash memory devices. We introduce flash-aware algorithms for counting, list-management, and B-trees, and analyze them using the EWOM model. This analysis shows that one can use the incremental 1-to-0 update properties of flash memory in interesting ways to reduce the required number of page-write operations.

1. INTRODUCTION

Solid state disks and other devices based on NAND flash memory allow many more random I/Os per second (up to two orders of magnitude more) than conventional magnetic disks. Thus they can, in principle, support workloads involving random I/Os much more effectively.

However, flash memory cannot support general in-place updates. Instead, a whole data page must be written to a new area of the device, and the old page must be invalidated. Groups of contiguous pages form erase units, and an invalidated page becomes writable again only after the whole erase unit has been cleared. Erase times are relatively high (several milliseconds). Flash-based memory does, however, allow in-place changes of 1-bits to 0-bits without an erase cycle [5]. Thus it is possible to reserve a region of flash memory initialized to all 1s, and incrementally use it in a write-once fashion.

Traditional measures of algorithm complexity do not model flash I/O behavior well, because the high cost of a general update (relative to a 1-to-0 update) is not accounted for. Previous models for write-once memory ("WOM") have been proposed to model devices like paper tape and optical disks in which the write process is destructive, so that once a bit is set it cannot be unset [10]. Maier proposes using write-once storage for a "Read-Mostly Store" (RMS) where the memory is gradually consumed as updates occur [8]. However, these models are too restrictive for devices like flash memory where a bulk erase allows memory to be reused.

Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008), June 13, 2008, Vancouver, Canada.

1.1 The EWOM Model

We propose a new model for evaluating an algorithm on a flash-like device. We call it the "Erasable Write Once Memory" model, or the "EWOM" model. In addition to counting traditional algorithmic steps, we count a page-write step whenever a write causes a 0 bit to change to a 1 bit. If an algorithm performs a group of local writes to a single page as one transactional step, we count the group as a single page-write step. Even if only a few bytes are updated, a whole page must be written.

The true cost of a page-write step has several components. There is an immediate cost incurred because a full page must be copied to a new location, with the bits in question updated. If there are multiple updates to a single page from different transactional operations, they can be combined in RAM and applied to the flash memory once, although one must be careful in such a scheme to guarantee data persistence if that is an application requirement.

There is also a deferred cost incurred because the flash device must eventually erase the erase unit containing the old page. It is a deferred cost because the write itself does not have to wait for the erase to finish; the erase can be performed asynchronously. Nevertheless, erase times are high, and a device burdened by many erase operations may not be able to sustain good read/write performance. Further, in an I/O intensive workload a steady state can be reached in which erasure cannot keep up, and writes end up waiting for erased pages to become available.

There is an additional longer-term cost of page erases in terms of device longevity. On current flash devices an erase unit has a lifetime of about 10^5 erases. Thus, if special-purpose algorithms reduce the number of erases

needed by a factor of f, the expected lifetime of the device can in principle be multiplied by f.

Our model can distinguish between situations where the I/O device is saturated, and where the device is lightly loaded. Algorithms might include a low-priority background process that asynchronously traverses data structure elements and reorganizes them to improve performance. The extra I/O workload will not be noticeable in a lightly-loaded setting, and most data structure elements will end up in the optimized state. In a saturated or near-saturated scenario, however, the background process will rarely run, and the data structure elements will remain in the unoptimized state.

We choose not to model "seek" time for flash memory. While there is a small overhead involved in moving from one memory location to another, this overhead is small relative to the erase costs. Further, this cost is orders of magnitude smaller than seek times for magnetic disks, whose performance models often distinguish between sequential and random I/O.

Traditional I/O devices have a fixed block transfer size, and it is customary to count the number of blocks transferred when measuring I/O complexity. RAM allows fine-grained data access, and so it is customary to simply count the number of computational steps to perform a given operation as the complexity measure. Flash memory occupies a middle-ground between traditional I/O devices and RAM. Some flash devices require transfers to happen in block-sized units, where a single device may support multiple block sizes, while others allow fine-grained access. For the purposes of the present work, we will adopt the convention that the flash memory is a fine-grained access device for reads and 1-to-0 writes, and we measure complexity by counting the total number of computational steps. For general updates, we also count a page write.[1]

1.2 Pages and Erase Units

Erase units are typically large, around 128KB. Copying a full erase unit on every update would not be efficient. It is therefore common for data copying to happen in page-sized units, where the page size P depends on how the device is configured. A typical value of P might be 2KB, meaning 64 pages in a 128KB erase unit.

We assume that there is a memory mapping layer that maps logical page addresses to physical page addresses. Such mapping is commonly implemented in hardware within page-granularity devices: when an update happens, the physical address changes, but the logical address remains the same so that updates do not need to be propagated to data structures that refer to the data page. When the device itself does not provide such a layer, it is common to implement such a layer in software. The mapping layer also ensures that wear on the device is shared among physical pages, because flash pages have a limited lifetime of approximately 10^5 erase cycles. The mapping layer can also hide faulty or worn-out pages from the operating system. The EWOM model assumes that a logical-to-physcial mapping layer is present.

If updates are performed on pages, then at any point in time, an erase unit may contain some valid pages and some invalid pages that need to be erased. If an erase unit contains valid pages, then those valid pages must be written to alternate locations before the erase unit can be erased. We assume that the same hardware or software that monitors the logical-to-physical mapping of pages also monitors the validity of pages for the purposes of managing erase units for garbage collection.

In a lightly loaded device, such extra copying might not be noticeable. However, in a heavily loaded system, with high demand for new erase units, this overhead will be noticeable.

A "best-case" workload for erase-unit recycling would occur when all pages in an erase unit are invalid at erase time. This kind of workload might happen if the data access pattern is highly clustered, such as when a file is sequentially updated, page by page. In that case, each page write contributes to approximately P/E erases, where E is the size of an erase unit.

A "worst-case" workload would occur when all erase units available for recycling hold just one invalid page. This kind of workload might happen on a device that is almost full, and for which the data access pattern is scattered over the various erase units. In this case, each page write causes an erase.

There are obviously many intermediates between the best and worst cases, and the range is wide. Thus it is not always possible to predict the erase frequency given just the page update frequency. More information about workload characteristics is usually needed.

2. COUNTING

We begin our analysis with a simple task: maintain a counter in EWOM storage. The counter is initialized to zero, and may be incremented. A naive, in-place solution would rewrite the counter, stored in conventional binary form, on every update. Since an increment always changes some 0 to a 1, every update requires a page write. Reads have cost proportional to the word size W of the counter in binary.[2]

An alternative solution represents the counter in unary form, with the number of zero bits indicating the count. An increment operation can be handled by changing a bit from 1 to 0 without a page-write operation. Unary counters are severely limited in their counting capacity since they have space complexity linear in the current value of the counter. Reads and writes can be handled in logarithmic time using an exponential expansion followed by a binary search to find the first 0 in the bit array.

A hybrid scheme stores a binary base counter, together with a unary increment counter of fixed length L, where $L \leq P - W$ so that the counter fits in a page [2]. The counter is computed by adding the base counter to the offset of the first zero in the unary array, which can be found using binary search. A page-write is needed

[1] If we fill a page using simple 1-to-0 writes, there are no page write operations counted.

[2] Depending on one's memory model, W is either constant or $O(\log n)$, where n is the value of the counter.

Method	Space (bits)	Read time	Write time	Page-Writes
Naive	W	W	$2W$	1
Unary	n	$\Theta(\log n)$	$\Theta(\log n)$	$\frac{1}{P}$
Hybrid (lightly loaded)	$W + L$	$W + 1$	2	0
Hybrid (saturated)	$W + L$	$W + O(\log L)$	$O(\log L) + \frac{2W}{L}$	$\frac{1}{L}$

Figure 1: Amortized complexity for counting

every L steps, at which time the base counter is recomputed, and the unary counter is reset.

A low-priority asynchronous operation may look through pages containing counters, and also perform this recompute/reset operation. We assume that in the lightly loaded case, the asynchronous background updates happen at least as often as writes, and promptly after those writes. This assumption means that the state of the counter on the flash device will usually have a zero unary increment value, with the binary part of the counter containing the current count. Reads become simpler (because they don't have to traverse the unary increment value), and writes become simpler (because there is always space for unary increments — no page-writes are necessary).

The complexity of these alternatives is summarized in Figure 1, where n is the number of increment operations, and P is the size of a page in bits. Read time is measured in terms of the number of bit operations needed. For the write step, we assume that the writer does not know the previous value of the counter, only that the counter needs to be incremented. As Figure 1 shows, the hybrid counting method amortizes page-writes almost as well as the unary method, while keeping read and write performance close to the naive method.

2.1 Arbitrary Increments

One can generalize the hybrid method if increments (or decrements) by arbitrary amounts are possible. A single base counter is maintained in binary form. A unary counter is kept for recording increments by multiples of 2^0, 2^1, 2^2, etc. An increment is broken down into its binary form, and the corresponding unary counters are updated. A separate set of counters is maintained for decrements. Read operations need to scan through the various counters to compute the net change to the binary stored value.

In the event that one of the unary counters is full, it may still be possible to process an addition without a page write by decomposing the addition into a larger number of smaller increments. For example, if the unary counter corresponding to 2^5 is full, we could add the value 2^5 by appending two bits to the unary counter corresponding to 2^4.

Other configurations are also possible. For example, instead of recording increments using a unary counter for each power of 2, one could use unary counters for powers of an arbitrary value k. The number of bits to set for each counter would be determined by the corresponding digit of the value to be added when written in base-k notation.

3. LINKED LISTS

A linked list is a commonly used data structure. In an EWOM context, standard list operations would require a page write. A page write would be needed to keep track of the tail of the list, to implement list element deletion, to insert an element into the list, and to update nodes within the linked list.

Suppose that we interpret the all-1 bit pattern as a NULL pointer. Then one can append to the list using only 1-to-0 updates by updating the NULL pointer in the last element of the list to point to a new element. The new element itself would be written in an area of the page initialized to all-1s. Unlike traditional append operations to a list, this variant would need to first traverse the entire list. On the other hand, a page-write is avoided.

Deletions would need to be handled in an indirect way, such as by using a "deleted" flag within the node structure. This would complicate list traversal slightly, because deleted nodes would remain in the list and need to have their flags checked.

Like for counting, we could implement a low-priority background process that "cleans up" lists on a page and writes a new page. In this new page, the deleted elements would be omitted. One could also store a shortcut to the current tail, so that future append operations do not have to start from the head of the list.

4. BLOOM FILTERS

Some data structures are inherently monotonic in their update behavior, and map well to the EWOM model without modification. An example is the Bloom filter [1]. If we interpret a vector of 1 bits to mean the empty Bloom filter, then every insertion can be achieved by setting some 1 bits to 0 bits. In the EWOM model, insert operations do not need to perform any page-writes.

5. B-TREES

Within a database system, one of the places where random I/O occurs frequently is in accessing B-tree indexes in response to OLTP workloads. Indexes are searched (to find the record to update), new records are inserted, and old records are deleted. One way to deal with B-tree update-heavy workloads is to batch the updates. That way, the costs associated with restructuring a page can be amortized over many updates. Batching happens implicitly when a page resides in the database system's buffer pool.[3] Batching can also happen close to the

[3]Note that the database system is ensuring persistence in this case by maintaining a recovery log.

physical device in a RAM-based cache. However, if the locality of reference of the database access is poor, such as when the table and/or index is much bigger than the buffer pool and records are being accessed randomly, there will be little effective batching in practice.

We therefore propose a new way to organize leaf nodes in a B-tree to avoid the page-write cost most of the time, while still processing updates one at a time. We focus on leaf nodes because that is where the large majority of changes happen.

Suppose that an entry in a leaf node consists of an 8-byte key, and an 8-byte RID referencing the indexed record. We assume a leaf node can hold L entries, taking $16L$ bytes. We shall assume that a leaf node has size that exactly matches the page size of the device.

With the requirement that leaf nodes be at least half full, a conventional B-tree leaf node will contain between $L/2$ and L entries stored in sorted key order. The ordering property allows for keys to be searched in logarithmic time using binary search.

A first attempt at a page-write-friendly leaf node would be to store all entries in an append-only array in the order of insertion [8]. A bitmap would be kept to mark deleted entries. When the node becomes full, it is split, and (nondeleted) entries are divided among the two resulting pages. The obvious drawback of this approach is that search time within the node will be linear rather than logarithmic, dramatically slowing down both searches and updates.

5.1 The Proposed Approach

Apart from the initial root node, all leaf nodes are created as a result of a split. When a split happens, we sort the (nondeleted) records into key order, and store them in that order in the append-only array. We keep track of the endpoint of this array by storing it explicitly in the leaf node. Subsequent insertions are then appended to the array as before.

So far, we have improved performance slightly because one can do a binary search over at least half of the entries, followed by a linear search of the remaining entries to find a key. However, the asymptotic complexity is still linear in the size of the array.

To speed up the search of the newly-inserted elements we store some additional information. Choose positive integer constants c and k. For every c entries in the new insertions, we store a c-element index array. Each entry in this index array stores an offset into the segment of new insertions, and the index array is stored in key order. (It is not maintained incrementally; it is generated only when there have been c new insertions.)

To search an array of m new elements ($m \leq L$), we need at most $(m/c)\log_2 c + (c-1)$ comparisons. While we have reduced the asymptotic search time by a factor of $c/\log_2 c$, it remains linear in m. The trick is to apply this idea recursively.

Suppose that after kc elements, instead of a c-element offset array, we store a kc-element offset array covering the previous kc newly inserted records. Now we need at most one linear search of at most $c-1$ elements, at most $k-1$ binary searches of c elements, and $\lfloor \frac{m}{kc} \rfloor$ binary searches of kc elements. If we keep scaling the offset array each time m crosses c, kc, k^2c, k^3c etc., then the total cost is $O(\log^2 m)$. (There are $O(\log m)$ binary searches, each taking $O(\log m)$ time.)

A complete search therefore takes $O(\log(n/L)+\log^2 L) = O(\log n+\log^2 L)$ time, where n is the number of elements in the tree.

The space overhead of this approach is the total size of the index arrays. This size is equal to

$$\begin{aligned} & \lfloor \tfrac{m}{c} \rfloor c + \lfloor \tfrac{m}{ck} \rfloor (ck-c) + \lfloor \tfrac{m}{ck^2} \rfloor (ck^2-ck) + \ldots \\ \approx \quad & m \log_k (m/c) = O(m \log m). \end{aligned}$$

The overhead for one node is thus $O(L \log L)$, and the overhead for the entire tree is $O(n \log L)$. This is a classical computer-science trade-off in which we use more space to reduce the time overhead. Different choices for c and k represent alternative points in the space-time trade-off.

In practice, the space overhead is unlikely to be onerous. For example, suppose that the page size is 16KB. 8KB can be devoted to new entries and the offset arrays. This places an upper bound of 512 new entries. If $c = 32$ and $k = 3$, the largest index array we will build will have 288 entries. The total space in bytes to store m new entries is then

$$16m+32(\lfloor m/32 \rfloor)+(96-32)(\lfloor m/96 \rfloor)+2(288-96)(\lfloor m/288 \rfloor).$$

(Here, we're assuming one byte offsets for up to 255 elements, and two-byte offsets for 256 or more elements.) Based on these numbers, we could store 446 new entries in the leaf node before we ran out of space. 1056 bytes out of 16K bytes (6.4%) is the space overhead, ignoring the pointer to the start of the new elements and the bits to record deletions.

Under lightly loaded conditions, where one has spare cycles to do background leaf optimization, one could convert a leaf node to sorted format and reset the pointers to new entries, writing the resulting node to a new memory location. For such "fresh" leaf nodes, search time goes down from $O(\log^2 m)$ time to $O(\log m)$ time. Note that because of the logical-to-physical page mapping, parent nodes are unchanged by leaf freshening.

5.2 Analysis

Every c entries, an updating transaction needs to sort c elements costing $O(c \log c)$ time. When the system gets to a $k^i c$-byte boundary, it only needs to sort the last c elements, then merge k ordered lists of size $k^{i-1}c$, which can be done in $O(c \log c + k^i c \log k)$ time. Amortizing over all insertions, the cost per insertion has order

$$\begin{aligned} & \textstyle\sum_{i=1}^{\lfloor \log_k (m/c) \rfloor} (k^i c \log k)(\lfloor \frac{m}{ck^i} \rfloor)/m \\ \approx \quad & \log k \lfloor \log_k (m/c) \rfloor \approx \log(m/c) \end{aligned}$$

Similarly, split processing can merge the array segments rather than fully sorting the array.

One needs to know where the array of new values ends, in order to decide when to terminate the search, and where to append new values. The simplest way to do this is to assume that a pattern of all 1-bits is not a valid (key,RID) pair. One can then binary search to find the last valid pair. One could try to explicitly store

Method	Space (bits)	Read time	Write time	Page Writes
Standard	$O(n)$	$O(\log n)$	$O(\log n)$	1
Append-Only	$O(n)$	$O(\log n + L)$	$O(\log n)$	$\frac{1}{L}$
Hybrid (lightly loaded)	$O(n \log L)$	$O(\log n)$	$O(\log n)$	0
Hybrid (saturated)	$O(n \log L)$	$O(\log n + \log^2 L)$	$O(\log n + \log L)$	$O(\frac{\log L}{L})$

Figure 2: Amortized B-tree complexity for a tree of size n, treating c and k as constants.

Pointer to the endpoint of initial sorted prefix.
Prefix P containing sorted (key,RID) pairs. Calculated during most recent page-write.
Sequence S of (key,RID) pairs in insertion order.
Indexes of first group of c elements of S, in sorted order.
Indexes of second group of c elements of S, in sorted order.
...
Indexes of $k - 1^{\text{st}}$ group of c elements of S, in sorted order.
Indexes of first group of kc elements of S, in sorted order.
Indexes of $k + 1^{\text{st}}$ group of c elements of S, in sorted order.
...
Deletion bits
Log Sequence Number (using generalized counter)

Figure 3: The final structure of a B-tree node of (key,RID) pairs.

the offset using the counters of Section 2, but such a method would consume more space than necessary.

Figure 2 shows the amortized asymptotic complexity for the proposed B-tree structure. We assume that writes do not need to check whether the key already exists in a node before insertion. If such a check is necessary, the entry for the append-only method would become $O(\log n + L)$ and the entry for the hybrid method with saturated writes would become $O(\log n + \log^2 L)$. Note that even in the saturated setting, the hybrid method only needs a page-write every $O((\log L)/L)$ insertions, while having better asymptotic read complexity than the append-only method.

5.3 Refinements

We have assumed that leaf nodes contain (key,RID) pairs. Sometimes, to save space, B-tree leaf nodes are designed to associate a key with a list of RIDs. The proposed structure can be modified so that at the time of reorganization (i.e., when a page-write occurs), the initial segment of data is in (key,RID-list) form. An alternative would be to keep a linked list of RIDs for each key, using the linked-list techniques described in Section 3.

In real B-tree implementations, a leaf node contains a log sequence number (LSN) recording information relevant for node recovery in case of failure. On an EWOM device, the LSN could be implemented using a generalized counter as described in Section 2.1. Note that LSNs are monotonically increasing, meaning that only increments, not decrements, need to be considered.

The final structure of a B-tree node is summarized in Figure 3. This figure shows a node containing (key,RID) pairs. If RID-lists were used, a region within the page would be used as a heap for allocating new RID nodes to add to RID-lists.

6. RELATED WORK

An interesting technique related to counting was proposed by Rivest and Shamir for the WOM model [10]. They show, for example, that it is possible to overwrite an arbitrary number from $\{0, 1, 2, 3\}$ with another arbitrary number from that set (a) using only monotonic bit changes, and (b) with only 3 bits of storage. Each possible number has two valid 3-bit encodings, such as

$$0 : 000, 111;\ 1 : 001, 110;\ 2 : 010, 101;\ 3 : 100, 011$$

The first code for a number is used for the initial write. The second code is used for the subsequent write, unless the second write has the same value, in which case there is no change. One could extend techniques like this to the EWOM model by erasing when necessary, which in this example would be after two or more updates.

Others have studied B-tree implementations for flash devices. Wu et al. [11] describe a B-tree method that uses a combination of RAM-resident buffers and "index units" representing flash-resident incremental changes to a B-tree node. The logical view of a B-tree node is reconstructed using the node together with these index units. The work of Wu et al. assumes a page-level interface to the flash device, without fine-grained access.

Nath et al., also study B-tree indexes on flash devices, with the aim of minimizing power and maximizing performance on a low-power mobile device [9]. Their system optimizes B-tree parameters in a self-tuning fashion, based on the workload and device characteristics. They also employ a page-level interface to the flash memory.

OLTP workloads frequently need to perform small updates in place. Lee and Moon show how to restructure

database pages and modify the logging protocol to minimize the required number of page erases [7]. Multiple versions of a data element are kept on a page in a write-once log-like structure within the page, and reads must consult the log to look for changes. Data is written to the flash storage in sector-sized units (512 bytes in [7]).

7. CURRENT DEVICES

The two basic types of flash memory available today are NOR-flash and NAND-flash. These technologies have contrasting behaviors that make them suitable for different classes of application [4]. For example, NAND-flash tends to have larger capacity, faster writes and erases, and page-level data access. NOR-flash tends to have faster reads, and fine-grained random access to data. Hybrid NAND/NOR devices exist (e.g., [6]).

The types of flash memory interaction allowed by a device vary. Some devices implement only a page-level API such as FTL [5], and updates to pages always cause a new page to be written. Such a choice allows an SSD device to resemble a magnetic disk device, and be used in existing systems that employ disk devices. Other devices (together with a software layer) expose flash as a "Memory Technology Device" (MTD) via UBI [3], which allows partial updates to pages. Low level flash interfaces have been defined by the ONFI working group[4]. In this paper, we assume an interface in which partial writes to a page are allowed, as long as they only involve trasitions from a 1 bit to a 0 bit.

Not every flash device may provide an interface that allows fine-granularity in-place 1-to-0 updates. As mentioned above, flash-based solid-state disks currently provide disk-like APIs, with pages or sectors as the unit of data transfer. Nevertheless, future devices may provide finer-grained APIs if there is a potential performance improvement. The results of this paper are a step in this direction, showing what is possible with such an API.

Some flash devices store error-correcting codes in reserved portions of the flash memory. Incremental changes to pages would also require incremental changes to the error-correcting codes. Even if the data changes are monotonic 1-to-0 writes, the resulting error-correcting code changes are unlikely to be monotonic. It may thus be necessary to reserve space for an array of error-correcting code values, and to write a new element into the array after each write.

While our EWOM model is motivated by flash memory, it is also possible that other technologies such as PRAM memory may, in the future, have similar block erase characteristics.

8. CONCLUSIONS

We have described a new model for measuring the performance of algorithms on write-once devices with an erase capability. We have adapted several standard algorithms to take account of the high page-write cost of arbitrary updates, and have analyzed their performance.

The results of this paper are unlikely to represent the final word on how to implement even the few techniques we have addressed. For example, it may be possible to trade space for time (or write performance for read performance) in different ways to get new algorithmic variants.

Acknowledgements

Thanks to Bishwaranjan Bhattacharjee, Christian Lang, Bruce Lindsay, Tim Malkemus, George Mihaila, Haixun Wang, and Mark Wegman for helpful discussions and suggestions.

9. REFERENCES

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[2] Paul England and Marcus Peinado. System and method for implementing a counter, 2006. US Patent Number 7,065,607.

[3] T. Gleixner, F. Haverkamp, and A. Bityutskiy. *UBI - Unsorted Block Images*, 2006.

[4] Toshiba Inc. NAND vs. NOR flash memory, 2006. Downloaded May 2008 from http://www.toshiba.com/taec/components/Generic/Memory_Resources/NANDvsNOR.pdf.

[5] Intel Corp. *Understanding the Flash Translation Layer (FTL) Specification*, 1998.

[6] T. H. Kuo et al. Design of 90nm 1Gb ORNAND flash memory with MirrorBit technology. In *Symposium on VLSI Circuits, Digest of Technical Papers*, pages 114–115, 2006.

[7] Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66, New York, NY, USA, 2007. ACM.

[8] David Maier. Using write-once memory for database storage. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 239–246, New York, NY, USA, 1982. ACM.

[9] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419, New York, NY, USA, 2007. ACM.

[10] Ronald L. Rivest and Adi Shamir. How to reuse a write-once memory (preliminary version). In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 105–113, New York, NY, USA, 1982. ACM.

[11] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient B-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3):19, 2007.

[4] `www.onfi.org`