Flash Device Support for Database Management

Philippe Bonnet
IT University of Copenhagen
Rued Langaard Vej 7
Copenhagen, Denmark
phbo@itu.dk

Luc Bouganim
INRIA Paris-Rocquencourt
Domaine de Voluceau
Le Chesnay, France
Luc.Bouganim@inria.fr

ABSTRACT

While disks have offered a stable behavior for decades -thus guaranteeing the timelessness of many database design decisions, flash devices keep on mutating. Their behavior varies across models, across firmware updates and possibly in time for the same model. Many researchers have proposed to adapt database algorithms for existing flash devices; others have tried to capture the performance characteristics of flash devices. However, today, we neither have a reference DBMS design nor a performance model for flash devices: *database researchers are running after flash memory technology.* In this paper, we take the reverse approach and we define how flash devices should support database management. We advocate that flash devices should provide DBMS with more control over IO behavior without sacrificing correctness or robustness, exposing the full potential of the underlying flash chips in terms of performance. We suggest two approaches: (a) keep the narrow block device interface, or (b) provide a rich interface that allows a DBMS to explicitly control IO behavior. We believe that these approaches are natural evolutions of the current generation of flash devices, whose complexity and opacity is ill-suited for database management. We describe the design space for the two proposed approaches, discuss how they would benefit many existing techniques proposed by the database research community, and identify a set of new research issues.

1. INTRODUCTION

For some time now, flash devices have been poised to replace disks as secondary storage [12]. Today, many different types of flash devices are finding their way into the memory hierarchy of database management systems (DBMS), from SSD to PCI-based racks (e.g., fusionIO and RamSan) and energy efficient FAWNs [5][1]. However, despite significant efforts [2, 9, 8, 17, 21, 29, 31, 20, 32], a reference design for database management with flash devices has yet to emerge. Flash devices have so far been a moving target for the database community.

[1]We do not consider in this paper architectures providing direct access to the flash chips, e.g., embedded flash [4]

Indeed, flash devices do not exhibit consistent characteristics. They embed a complex software called Flash Translation Layer (FTL) in order to hide flash chip constraints (erase-before-write, limited number of erase-write cycles, sequential page-writes within a flash block). A FTL provides address translation, wear leveling and strives to hide the impact of updates and random writes based on observed update frequencies, access patterns, temporal locality, etc. Their performance characteristics and energy profiles vary across devices [9, 8]. For instance, random writes are faster than reads on FusionIO's ioDrive [7] while random writes are much slower than the other operations on the Samsung model [9]. For some devices, performance varies in time based on the history of IOs, e.g., the performance of the Intel X25-M varies by an order of magnitude depending on whether the device is filled with random writes or not. What is the value of a DBMS design based on a storage subsystem whose behavior is not well understood and keeps on mutating?

By contrast, successive generations of disks have complied with two simple axioms: *(1) locality in the logical address space is preserved in the physical address space; (2) sequential access is much faster than random access.* As long as hard disks remained the sole medium for secondary storage, the block device interface proved to be a very robust abstraction that allowed the operating system to hide the complexity of IO management without sacrificing performance. The block device interface is a simple memory abstraction based on read and write primitives and a flat logical address space (i.e., an array of sectors). Since the advent of Unix [30], the stability of the interface and the stability of disks characteristics have guaranteed the timelessness of major database system design decisions, i.e., pages are the unit of IO with an identical representation of data on-disk and in-memory; random accesses are avoided (e.g., query processing algorithms) while sequential accesses are favored (e.g., extent-based allocation, clustering).

We must address the tension that exists between the design goals of flash devices and DBMS. Flash device designers, especially SSD and PCI-based racks designers, aim at hiding the constraints of flash chips to compete with hard disks providers. They also compete with each other, tweaking their FTL to improve overall performance, and masking their design decision to protect their advantage. Database designers, on the other hand, have full control over the IOs they issue. What they need is a clear and stable distinction between efficient and inefficient IO patterns, so that they can adapt their allocation strategies, data representation or

query processing algorithms to the characteristics of the underlying storage devices. They might even be able to trade increased complexity for improved performance and stable behavior across devices.

So the problem is the following: How can flash devices provide DBMS with guarantees over IO behavior? Interestingly, flash chips already provide a clear, stable distinction between efficient patterns (page reads, sequential page-writes within a block), and inefficient patterns (in-place updates). Our key insight is that flash devices should expose this distinction instead of aggressively mitigating the impact of inefficient patterns at the expense of the efficient ones (e.g., trading reduced read performance to obtain improved random writes). We see two approaches:

1. *Narrow Interface Device*: The most immediate approach is to keep the existing block device interface. Since flash devices have no knowledge of the manipulated data, we should (a) let the DBMS optimize its accesses and, (b) avoid uncontrolled FTL optimizations. This approach is similar, in spirit, to the way DBMSs interact with virtual memory [30]. Plagiarizing Stonebraker, this consists in replacing a "not quite right" service provided by the flash device with a comparable, application specific, service within the DBMS. The key questions here are: What abstractions should the FTL provide? How to handle the increased complexity at the DBMS level?

2. *Rich Interface Device* An alternative approach would be to rely on a rich interface to let DBMS and flash device collaborate on how to optimize performances. Indeed, there is an emerging consensus that the block interface is too narrow [28, 22, 23, 25, 26]. Coping with the block interface forces flash devices to perform complex tasks (i.e., wear leveling, space reclamation, garbage collection) independently from the application, possibly against its best interest. The key questions here are: What information should be passed from the DBMS to the flash device so that it can optimize its performance[2]? What kind of optimizations can be defined on flash devices? How should we design DBMSs to leverage a rich flash device interface?

We see Narrow and Rich devices as natural evolutions of the current generation of flash devices whose complexity and opacity is ill-suited for database management. Such an evolution is particularly important for database machines designers (e.g., Oracle's Exadata or Neteeza's TwinFin) that have to specify well-suited flash components for their systems. More generally, we understand that SSD manufacturers will only move if the gain is clear for their business. We see here an opportunity for the database community to influence the evolution of flash devices for the benefits of commodity database systems.

In this paper, we define the guarantees that a FTL should provide to a DBMS, we detail Narrow and Rich flash devices and we explore how they will impact database management. Throughout the paper, we illustrate how ideas expressed in the literature would benefit from these new classes of devices. We also describe new research challenges.

[2]Note that we focus on IO performance; we do not consider integrating high-level database abstractions within storage devices, e.g., active disks [24]. Whether a rich interface naturally leads to active disks is a topic for future work.

2. FLASH DEVICES

At their core, flash devices rely on NAND flash chips that store data in independent arrays of memory cells. Each cell accommodates 1, 2, or 3 bits of information (SLC, MLC, TLC). Each array is a *flash block*, and rows of memory cells are *flash pages*[3].

The Good. A single flash chip can offer great performance (e.g., 40 MB/s Reads, 10 MB/s write) with low energy consumption [8]. Thus, tens of flash chips wired in parallel can deliver hundreds of thousands IOs per second. At the chip level, random operations are as fast as sequential ones. Recent flash chips can interleave operations and include multiple independent planes, thus processing operations concurrently [3]. A flash device is composed of a collection of flash chips, wired in parallel to a controller. The controller includes some cache (e.g. 16-32MB), potentially safe with respect to power failure [3]—e.g., cache can be RAM with capacitors or other NVM (eg. PCM). From this perspective, the potential of flash device is impressive.

The Bad. Unfortunately, flash chips have severe constraints: *(C1) Write granularity.* Writes must be performed at a page granularity[4]. *(C2) Erase before write.* A costly erase operation must be performed before overwriting a flash page. Even worse, erase operations are only performed at the granularity of a flash block (typically 64 flash pages). *(C3) Sequential writes within a block.* Writes must be performed sequentially within a flash block in order to minimize write errors resulting from the electrical side effects of writing a series of cells[5]. *(C4) Limited lifetime.* SLC, MLC and TLC flash chips can support respectively up to 10^6, 10^5, 5×10^4 erase operations per flash block. The trend is that flash chips store more bits per cell (e.g., TLC) with a smaller process geometry (e.g., 25 nm), larger page size, larger number of page by blocks and smaller lifetime. None of these evolutions challenge the nature of C1-C4.

And the FTL. The controller embeds the so-called Flash Translation Layer (FTL) in order to hide the aforementioned constraints. Typically, the FTL implements out-of-place updates to handle C2 using some reserved flash blocks called *log blocks*. Each update leaves, however, an obsolete flash page (that contains the before image). Over time such obsolete flash pages accumulate, and must be reclaimed by a *garbage collector*. A mapping between the logical address space exposed by the FTL and the physical flash space is necessary to handle writes smaller than a flash page (C1), updates (C2), random writes (C3), and to support wear leveling techniques (C4), which distribute erase operations across flash blocks and mask bad blocks. This mapping is implemented based on a mapping table located in the controller cache, on flash or both [13]. *Page mapping*, with a mapping entry for each flash page generates large maps that do not fit in the controller cache for large capacity devices. *Block mapping* reduces drastically the mapping table to one entry per flash block [15]—the challenge is then to minimize the overhead for finding a page within a block. Also, block mapping does

[3]Flash pages may further be broken up into *flash sub-pages*
[4]While the write granularity is the page (4KB-8KB) for MLC NAND (no sub-pages), the read one can be smaller [31]. Actually, read granularity depends on the ECC sector size (512B - 1KB).
[5]Electric side effects may generate write errors effectively managed with error correction codes (ECC) at the hardware level.

not support random writes and updates efficiently. More recently, many *Hybrid mapping* techniques [19, 18, 13, 15] have been proposed that combine block mapping as a baseline and page mapping for log blocks. Besides mapping, existing FTL algorithms also differ on their wear leveling algorithms [10], as well as their log blocks management and garbage collection methods (block associative [16], fully associative [19], using detected patterns [18] or temporal locality [15]).

3. TUNNELING DBMS IO

Our overall problem is to resolve the tension that exists between FTL and DBMS design goals. From the point of view of a DBMS designer, the trivial solution is to take the FTL out of the equation and let a DBMS directly access flash chips. Obviously, the first step that DBMS designers would take is to introduce modularity to hide the complexity of dealing with flash chips, in effect re-introducing some form of FTL. So, the interesting question is not how to bypass the FTL, but what kind of FTL can DBMS designers rely on?

3.1 Minimal FTL

To tackle this issue, we define the minimal level of service that a FTL should provide—because a DBMS cannot. The idea is that a minimal FTL would give maximal control and maximal stability to the DBMS. Let us look back at the constraints imposed by flash chips. First, a DBMS can trivially issue IOs at the granularity of a flash page (C1). This would require the FTL to advertise how flash pages should be aligned at the logical level. Second, a DBMS could rely on the *Trim* command[6] to tell a flash device to erase a flash block before it is written (C2). This would require the FTL to expose an abstraction of flash blocks at its interface. Third, a DBMS could write flash pages in sequence within flash blocks (C3). This would require the FTL to advertise how flash blocks are aligned, and how flash pages are mapped into flash blocks. Fourth, a DBMS cannot implement wear-leveling (C4). It must be provided by the FTL. Indeed, wear-leveling is absolutely necessary to guarantee the lifetime of a device. Flash device manufacturers will never release a product that can wear-out after some minutes of focused and intensive write/erase cycles.

One can thus imagine building flash devices with an FTL that implements wear leveling and provides abstractions for flash pages (a *logical page*) and flash blocks (a *logical block*). Such a FTL would rely on a block level map which is compact enough to be efficiently kept in the flash device safe cache (e.g., 16MB cache for 1 TB of 256 KB flash blocks)[7] With such a FTL, the DBMS would have to handle constraints C1-C3 —i.e., the DBMS could not submit IOs for in-place updates or random writes. On the other hand, the FTL would guarantee that the IOs that respect these constraints would be tunneled to the underlying flash chips as directly as possible (see Figure 1).

Would flash device manufacturers be interested in providing such FTLs? Probably not. Could DBMS designer always circumvent random writes on flash devices? Maybe in some cases, but definitely not for general-purpose databases. So our problem is to define a bimodal FTL which achieves optimal performance as long as the DBMS manages constraints C1-C3 (minimal FTL), while providing best effort performance for all other IO patterns[8]. Interference between these two modes of operations should be minimized.

[6]The Trim command has been introduced in the ATA interface standard [28] to communicate to a flash device that a range of LBAs are no longer used by an application

[7]Otherwise, the block mapping has to be partially written on flash (as was the case in early block mapping FTLs [11]).

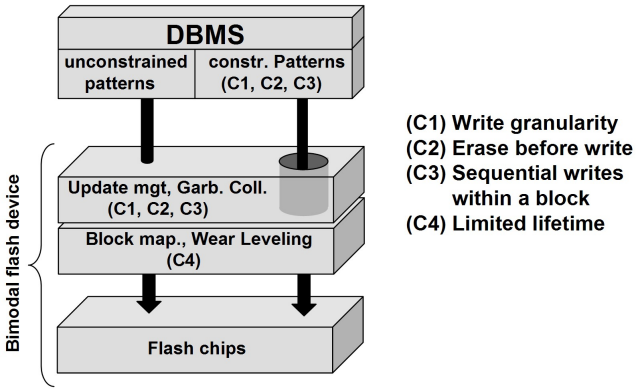

Figure 1: Bimodal FTL that combines near-optimal performance for IO as long as the DBMS respects constraints C1-C3, and best effort performance for other IO patterns where the FTL must enforce these constraints.

3.2 Bimodal FTLs

While flash device designers have focused on efficiently enforcing the flash chip constraints for updates or random writes, there has not been much work on characterizing optimal performance for a flash device. More precisely, what is the most optimal mapping that a FTL can achieve? We have seen that a FTL must at least implement a form of block mapping to support wear-leveling. A mapping is thus optimal if (a) the block look-up is performed in the controller cache, and (b) the offset of the page within the block is derived from the logical address (i.e., consecutive logical addresses are written sequentially within a block). In the following, a flash block for which mapping is optimal is called an *optimal block*.

A bimodal FTL must provide optimal mapping for those logical blocks for which the DBMS guarantees that writes are performed sequentially (C2) at the granularity of a flash page (C1), while any update is preceded by a Trim command (C2). The other logical blocks—on which all IO patterns are allowed—are mapped to one or more physical flash blocks, depending on the algorithms used by the FTL to manage constraints C1-C3. The design of the FTL optimizations for non-optimal blocks is not in the scope of this paper. State of the art techniques can be used [16, 18, 19, 15, 13] as long as they do not directly interfere with optimal blocks. We argue the feasibility of this approach in Sections 4 and 5.

Obviously sequential writes will result in an optimal mapping. Interestingly, semi-random writes[9], i.e., random write

[8]A bimodal FTL must implement a form of wear-leveling and garbage collection. This requires some work, and as a result, a DBMS can never obtain an IO throughput strictly equal to the throughput of the underlying flash chips even when the FTL guarantees optimal mapping.

[9]The notion of semi-random writes was introduced in [21]; in our approach, it is the number of logical blocks that limits the degree of parallelism in the semi-random writes.

IOs which are mapped sequentially on different logical blocks will also result in an optimal mapping. Sequential reads and random reads benefit equally from an optimal mapping. An interesting side effect is that an optimal block never needs to be garbage collected; while for non-optimal blocks, the goal of the garbage collector must be to tend towards optimal mapping.

3.3 Impact on DBMS design

Bimodal flash devices provide a stable and optimal basis for DBMS design. Even if more work is needed to investigate the actual impact of our approach, we can already make some preliminary observations[10].

Existing work focused on making database writes sequential (e.g., Append and Pack [29], ReSSD [20], NIPU [32]) is today restricted to repairing the bad random write performance of low-end SSDs. Such a technique would be required to leverage the optimal performance of sequential writes in case of a bimodal FTL. Also, techniques designed for flash chips (e.g., Lazy adaptive trees [2], or Page-differential Logging [17]), which are not today adequate in the context of SSDs, could naturally be applied to bimodal FTLs since they would generate only optimal blocks. Other techniques based on hashing or sorting (e.g., online maintenance of very large random samples [21]) can today only be applied to those SSDS that support (a limited form of) semi-random writes. Such techniques as well as hash-join or sort-merge would clearly benefit from the performance of semi-random writes on a bimodal FTL as long as bukets are aligned on block boundaries. Finally, the FlashScan and FlashJoin algorithms proposed in [31], that aggressively make use of random read IOs of small granularities are today rather well supported in current SSDs, even if random reads are slower than sequential reads on many SSDs. A bimodal FTL could ensure optimal performance with random reads as performant as sequential reads.

While it is impossible to develop a performance model of existing SSDs (because of their opacity and complexity), it is possible to envisage both analytic models and simulation models of bimodal flash devices, in order to explore the DBMS design space.

4. NARROW BIMODAL FLASH DEVICES

Is it possible to implement a bimodal FTL without violating the constraints of a block device interface, i.e., fixed size IOs, flat name space of logical addresses, interface reduced to read/write/trim commands? In this section we focus on how to guarantee optimal mapping when a DBMS enforces the flash chip constraints, since many existing techniques can be used to mitigate the effects of updates or random writes for non-optimal blocks. More specifically, the issues are (a) How to represent logical blocks and pages with a block device interface and (b) How to detect whether a DBMS enforces the flash chip constraints?

We propose an obvious implicit and immutable scheme for associating logical addresses to logical blocks and pages. The flash device must expose two constants[11]: *Logical Block Size* or LBS and *Logical Page Size* or LPS. LBS (resp. LPS) correspond to the size, in bytes of a flash block (resp. a flash page)[12]. For a logical address A, the logical block number LBN and logical page number LPN are obtained using the following trivial formulas: $LBN = A/LBS$ and $LPN = (A - LBN \times LBS)/LPS$, where / is the integer division.

While page size IOs are submitted sequentially within a logical block (starting from page 0), flash pages are written in the order the IOs are submitted. This way, the layout of flash pages within a flash block is optimal, and each read (either sequential or random) can be trivially mapped to an offset within a flash block. This is detected by maintaining in the safe cache, for each flash block, the physical position of the last write within the block. A bit indicates whether the mapping is optimal or not (initially, free blocks are optimal). We thus maintain in the controller cache a mapping with 4 bytes per block (22 bits for physical block id + 6-8 bits for current position within block (64-256 pages per blocks) + 1 bit flag for the optimal mapping).

An optimal block might become non-optimal if the submitted IOs are no longer sequential (e.g., updates, unaligned IOs (across page and block boundaries), random IOs). Conversely, classical garbage collector algorithms [19, 15], triggered in case of updates or random writes, will tend to convert non-optimal blocks into optimal ones.

The obvious question is whether any of the currently available flash devices implement such a FTL. We have devised a set of tests to check whether a device can provide optimal mapping (with safe cache) or near optimal one (no safe cache). None of the devices tested (including FusionIO, Intel and Samsung devices) were found to neither provide optimal nor near optimal mapping[13].

While a significant improvement with respect to today's devices, the narrow approach is not perfect: (i) Wear-leveling and garbage collection are performed without any knowledge of the stored data, (ii) The utilization of the controller cache is not optimized; (iii) A lot of complexity related to flash chips is managed at the DBMS level. In the next Section, we introduce the Rich approach that addresses these shortcomings.

5. RICH BIMODAL FLASH DEVICES

Extension of the block device interface have already been proposed in the context of flash devices [28, 23, 22] to manage complexity, control trade-offs and optimize embedded resources. While, Shu et al. [28] introduce the Data Management Set in the ATA protocol— connecting host and peripherals— that allows an application to communicate information about its I/O access behaviors to the underlying flash device, Rasjimwale et al [23] propose to use expressive interface such as object-based storage, and Prabhakaran et al. [22] focus on providing atomic writes. In the following, we describe how rich interfaces could be used to optimize

[10]We have identified the impact of bimodal FTL on a large set of existing techniques proposed in the literature based on the assumptions they make. If this paper gets accepted, we will conduct a detailed discussion, probably in a separate section at the end of the paper.

[11]Such constants can be retrieved by the DBMS using specific command like GetDriveGeometry

[12]Parallelism within the flash device might lead to define LBS (resp. LPS) as a multiple of the flash block size (resp. the flash page size.

[13]If this paper gets accepted, we will integrate our test and some results obtained on a range of SSDs in the final, longer version of the paper

the minimal FTL mode–most optimizations have already been presented in the literature–, and we quickly discuss non-optimal blocks. We deliberately avoid discussions of implementation or syntax issues, which are left for future works.

5.1 Optimal Blocks

From optimal blocks to optimal chunk: Since the block device interface does not provide any means to explicitly declare a set of optimal blocks (called hereafter *chunk*), these have to be detected by the FTL. More importantly, this detection leads to the management of a large number of small blocks, having an impact on the volume of metadata (mapping, statistics, and detection data) that must be stored into the safe cache. Explicit declaration of larger chunks may thus bring significant savings in terms of safe buffer. For instance, a single 100 MB optimal chunk, explicitly declared to the flash device, e.g., for the log file of a DBMS, may bring a two order of magnitude reduction of the metadata wrt the implicit detection of 400 optimal blocks. While a rich interface has no direct impact on the performance of optimal blocks (the mapping is already optimal), it allows for improved caching (see below).

Providing metadata: In a block device, wear leveling and garbage collection proceed blindly, without knowledge of the access patterns on the managed data. Choosing highly erased blocks to store data with low erase frequency and conversely will avoid useless blocks movements to balance erase counts. A similar strategy, based on detection, has been proposed in [10] in the context of a block device interface. Note that minimizing blocks movements increase performance and lifetime of the device.

Caching data: DBMS and FTL could collaborate to avoid producing non-optimal blocks for append data structures (e.g., log records, tables in append mode). The FTL can avoid update operations (and thus degrading an optimal block to non-optimal) if the last written flash page can be kept inside the safe buffer and only written to flash when it is full.

Transmitting and caching payload: A rich flash device no longer has to be restricted to a flat, regular address space based on logical block addresses. We can consider write IOs which only transmit useful data, called hereafter the *payload*, i.e., fragments of pages as opposed to pages (note that we do not consider here that the flash device manages a representation of the database page layout). The main advantage of using payload instead of pages is that write operations can be buffered efficiently. (e.g., n inserted tuples in a database table will lead to buffer the aggregate size of the tuples plus some offsets-to recompose the pages-, compared with n IO sectors with a narrow interface!). Indeed, the safe buffer is not polluted with data that already exists in the flash memory. Note that even if the safe cache is full, it may be more interesting to temporarily flush the payload part of the cache in a dedicated flash area (as a swap file) rather than flushing on their destination, in order to keep the destination blocks optimal. Write payload shares some ideas with the Page-Differential Logging approach [17].

Reading payload: Payload optimization is also very interesting for read operation since it avoids transmitting useless data from the chip to the controller and then to the host (typically, reading a 4KB page costs around 150 μs from which 125 μs are transmission costs through the serial interface from the chip to the controller). FlashScan [31] already mentioned in section 3.3 could benefit further from read payload, reading only subpages (i.e., corresponding to the ECC granularity).

Nameless writes: Nameless writes were introduced in [6]: the DBMS does not provide any destination address but let the FTL decides on the placement of the data and returns a handle for future reference. Nameless writes thus generates optimal blocks but are somehow easier to manage at the DBMS level, typically for temporary data.

Reorganization for free: A further improvement would consist in letting the FTL expose some aspects of the wear-leveling process to the DBMS to support additional optimizations. The wear leveling process sometimes has to move static data to balance the erase count across the whole device. In this case, the FTL reads a whole block that it rewrites on another physical location, thus bringing an opportunity to reorganize the block for free. The FTL could involve the DBMS in this process using a call-back mechanism in the way that external pagers interact with the operating systems [1]. Indeed, to allow using optimal blocks, DBMSs will probably resort to log-based approaches [2, 17] which might greatly benefit from such cleaning (almost) for free.

5.2 Non-optimal Blocks

While this paper focus on optimal blocks, we may mention that the extension mentioned above also apply to other blocks. Actually, there is a greater potential for minimizing the important overheads generated by the management of non-optimal block: mapping cost (metadata management), garbage collection costs (with statistics like R/W frequency, expected pattern, etc.) and wear leveling costs (update frequency). For instance:

Hot-random-chunks and cold-random-chunks: The FTL can use different techniques for mapping different chunks as long as the DBMS can tell the FTL about its access patterns for a given chunk. The FTL can manage hot random chunks (i.e., frequent writes, scattered on the whole chunk) with a page mapping cached in the safe buffer, while mapping can be performed at a larger granularity and on flash for cold random chunks. These strategies share the basic principles proposed in [13, 14], but are based on information delivered by the DBMS (and not detected by the FTL).

6. CONCLUSION

We argued that flash devices should provide guarantees to a DBMS so that it can devise stable and efficient IO management mechanisms. Based on the characteristics of flash chips, we defined a bimodal FTL that distinguishes between a minimal mode where sequential writes, sequential reads and random reads are optimal while updates and random writes are forbidden, and a mode where updates and random writes are supported at the cost of sub-optimal IO performance. Interestingly, the guarantees of a minimal mode have been taken for granted in many articles from the database research literature. Our point with this paper is that these guarantees are not a law of nature, we must guide the evolution of flash devices so that they are enforced.

We described the design space for bimodal flash devices in the context of a block device interface, and in the context of a richer interface. Which one is most appropriate for a DBMS is an open issue. An important point is that providing op-

timal mapping guarantees does not hinder competition between flash device manufacturers. On the contrary, they can compete to (a) bring down the cost of optimal IO patterns (e.g., using parallelism), and (b) bring down the cost of non-optimal patterns without jeopardizing DBMS design. Future work includes designing and building a bimodal FTL in collaboration with a flash device manufacturer.

We can also derive a future work roadmap for database designers: (1) Define a performance model for bimodal flash devices in order to explore the DBMS design space. The reference here is the work of John Wilkes et al. on hard disks models [27]; (2) Explore the DBMS design space on top of narrow bimodal flash devices. The first challenge here is to compare and integrate the many ideas already proposed in the literature to establish a baseline. The interesting problem is then to optimize this baseline design; and (3) Explore the design space for the collaboration of DBMS and rich bimodal flash devices.

Finally, future work includes studying how operating systems as well as many data-intensive applications would benefit from flash devices with optimal mapping guarantees (e.g., warehouse scale distributed systems, game engines, IO-conscious algorithms).

7. REFERENCES

[1] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *USENIX Summer*, 1986.

[2] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1), 2009.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX ATC*, 2008.

[4] T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. L. Folgoc, B. Nguyen, P. Pucheral, I. Ray, I. Ray, and S. Yin. Secure Personal Data Servers: a Vision Paper. *PVLDB*, 3(1), 2010.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a Fast Array of Wimpy Nodes. In *ACM SOSP*, 2009.

[6] A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and V. Prabhakaran. Removing The Costs Of Indirection in Flash-based SSDs with Nameless Writes. In *USENIX HotStorage*, 2010.

[7] M. Bjørling, L. L. Folgoc, A. Mseddi, P. Bonnet, L. Bouganim, and B. Jónsson. Performing sound flash device measurements: some lessons from uFLIP. In *SIGMOD Conference*, 2010.

[8] M. Bjørling, P. Bonnet, L. Bouganim, and B. Jónsson. Understanding the Energy Consumption of Flash Devices with uFLIP. *IEEE Data Eng. Bull.*, to appear, 2010.

[9] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[10] L.-P. Chang and C.-D. Du. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.*, 15(1), 2009.

[11] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2), 2005.

[12] J. Gray. Tape is dead, disk is tape, flash is disk. `http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt`.

[13] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, 2009.

[14] J. Hu, H. Jiang, L. Tian, and L. Xu. PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD. *MASCOTS*, 2010.

[15] D. Jung, J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Trans. Embed. Comput. Syst.*, 9(4):1–41, 2010.

[16] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *Consumer Electronics, IEEE Transactions on*, 48(2), may. 2002.

[17] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song. Page-differential logging: an efficient and DBMS-independent approach for storing data into flash memory In *SIGMOD Conference*, 2010.

[18] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6), 2008.

[19] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), 2007.

[20] Y. Lee, J. Kim and S. Maeng. ReSSD: a software layer for resuscitating SSDs from poor small random write performance. In *SAC*, 2010.

[21] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *VLDB J.*, 19(1), 2010.

[22] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *OSDI*, 2008.

[23] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *USENIX ATC*, 2009.

[24] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6), 2001.

[25] S. W. Schlosser and G. R. Ganger. MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole? In *USENIX FAST*, 2004.

[26] S. W. Schlosser, J. Schindler, A. Ailamaki, and G. R. Ganger. Exposing and Exploiting Internal Parallelism in MEMS-based Storage. CMU Technical Report CMU-CS-03-125, 2003.

[27] E. A. M. Shriver, A. Merchant, and J. Wilkes. An Analytic Behavior Model for Disk Drives With Readahead Caches and Request Reordering. In *SIGMETRICS*, 1998.

[28] F. Shu and N. Obr. Data set management commands proposal for ATA8- ACS2. `http://www.t13.org/`, 2007.

[29] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMoN*, 2009.

[30] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7), 1981.

[31] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD Conference*, 2009.

[32] Y. Wang, K. Goda and M. Kitsuregawa. Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems. In *DEXA*, 2009.