The Necessary Death of the Block Device Interface

Matias Bjørling[1], Philippe Bonnet[1], Luc Bouganim[2,3], Niv Dayan[1]

[1] IT University of Copenhagen
Copenhagen, Denmark
{mabj,phbo,nday}@itu.dk

[2] INRIA Paris-Rocquencourt
Le Chesnay, France
Luc.Bouganim@inria.fr

[3] PRISM Laboratory
Univ. of Versailles, France
Luc.Bouganim@prism.uvsq.fr

ABSTRACT

Solid State Drives (SSDs) are replacing magnetic disks as secondary storage for database management, as they offer orders of magnitude improvement in terms of bandwidth and latency. In terms of system design, the advent of SSDs raises considerable challenges. First, the storage chips, which are the basic component of a SSD, have widely different characteristics – e.g., copy-on-write, erase-before-write and page-addressability for flash chips vs. in-place update and byte-addressability for PCM chips. Second, SSDs are no longer a bottleneck in terms of I/O latency forcing streamlined execution throughout the I/O stack. Finally, SSDs provide a high degree of parallelism that must be leveraged to reach nominal bandwidth. This evolution puts database system researchers at a crossroad. The first option is to hang on to the current architecture where secondary storage is encapsulated behind a block device interface. This is the mainstream option both in industry and academia. This leaves the storage and OS communities with the responsibility to deal with the complexity introduced by SSDs in the hope that they will provide us with a robust, yet simple, performance model. In this paper, we show that this option amounts to building on quicksand. We illustrate our point by debunking some popular myths about flash devices and by pointing out mistakes in the papers we have published throughout the years. The second option is to abandon the simple abstraction of the block device interface and reconsider how database storage managers, operating system drivers and SSD controllers interact. We give our vision of how modern database systems should interact with secondary storage. This approach requires a deep re-design of the database system architecture, which is the only viable option for database system researchers to avoid becoming irrelevant.

1. INTRODUCTION

For the last thirty years, database systems have relied on magnetic disks as secondary storage [19]. Today, the growing performance gap between processors and magnetic disk is pushing solid-state drives (SSDs) as replacements for disks [11]. SSDs are based on non-volatile memories such as flash and PCM (Phase-Change memory). They offer great performance at an ever-decreasing cost. Today, tens of flash chips wired in parallel behind a safe cache deliver hundreds of thousands accesses per second at a latency of tens of microseconds. Compared to modern hard disks, this is a hundredfold improvement in terms of bandwidth and latency, at ten-times the cost. New SSD technologies, such as PCM, promise to keep on improving performance at a fraction of the cost.

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

It has now been six years since Jim Gray pointed out the significance of flash-based SSDs. Has a new generation of database systems emerged to accommodate those profound changes? No. Is a new generation of database systems actually needed? Well, the jury is up. There are two schools of thoughts:

- The conservative approach, taken by all database constructors, and many in the research community, is to consider that the advent of SSDs does not require any significant re-design. The fact that SSDs offer the same block device interface as magnetic disks allows preserving existing database systems and running them unchanged on SSDs (slight adaptations being sold as SSD-optimizations). A fraction of radical conservatives ignore SSDs and keep on writing articles and grant proposals based on disks, as if we were in the 90s (How will they teach about database systems in five years, when none of their bachelor students has ever seen a disk?). More moderate conservatives, focusing on storage management, consider that database systems have to be redesigned on top of the block device interface, based on the new performance characteristics of SSDs. The hope is that the storage and operating system communities provide a robust, yet simple, performance model for the new generation of storage devices.
- The progressive approach is to consider that the advent of SSDs, and non-volatile memories more generally, requires a complete re-thinking of the interactions between database system, operating system and storage devices. The argument is that SSDs challenge the strict layering established between these components on the basis of a simple performance contract, e.g., sequential access is no longer orders of magnitude faster than random access, SSDs are no longer the bottleneck in terms of latency, SSDs require a high-level of parallelism, SSDs do not constitute a homogeneous class of devices (as opposed to disks). This approach, that requires a deep cross layer understanding, is mainstream in the operating system and storage research communities [7,9,13]; not yet in the database systems research community.

The premise of the conservative approach is that the block device interface should be conserved as a robust abstraction that allows the operating system to hide the complexity of I/O management without sacrificing performance. We show, in Section 2, that this assumption does not hold; neither for flash-based nor for PCM-based devices. Worse, we show that it leads to brittle research based on myths rather than sound results. We debunk a few of these myths, illustrating our points with mistakes published in the articles we have written throughout the years.

In Section 3, we present the challenges that SSDs and non-volatile memories pose in terms of system design and discuss how they impact database systems. We present our vision of the necessary collaboration between database storage manager and operating system.

Note that we do not dispute that the conservative approach is economically smart. Neither do we ignore the fact that disks still largely dominate the storage market or that the block device interface will live on as a legacy for years. Our point is that the advent of SSDs and non-volatile memories has a deep impact on system design, and that we, as database systems researchers, must re-visit some grand old design decisions and engage with the operating system and storage communities in order to remain relevant.

2. THE CASE AGAINST THE BLOCK DEVICE INTERFACE

2.1 SSD MYTHS

Even if the block device interface has been challenged for some years [18], these critics have had, so far, a limited impact. For instance, all research papers published in the database community, proposing new storage models, indexing methods or query execution strategies for flash devices still build on the premise of SSDs encapsulated behind a block device interface [5]. All of these approaches assume, more or less explicitly, a simple performance model for the underlying SSDs. The most popular assumptions are the following:

- *SSDs behave as to the non-volatile memory they contain*: Before flash-based SSDs became widely available, there was a significant confusion between flash memory and flash devices. Today, we see a similar confusion with PCM.
- *On flash-based SSDs, random writes are extremely costly and should be avoided*: This was actually always true for flash devices on the market before 2009. Moreover, this rule makes sense after a quick look at flash constraints and SSD architecture. Many thus propose to avoid random writes using buffering and log-based strategies.
- *On flash-based SSDs, reads are cheaper than writes*: Again this seems to make sense because, (1) reads on flash chips are much cheaper than writes (the so-called program operations); (2) flash chip constraints impact write operations (need for copy-on-write as in-place updates are forbidden on a flash chip). Some proposals are built on this rule, making aggressive use of random read IOs.

We will show in Section 2.3 that these assumptions about (flash-based) SSDs are plain wrong, but first, let us review the internals of a flash-based IO stack -- from flash chips to the OS block layer.

2.2 I/O STACK INTERNALS

A point that we would like to carry across is that we, as database researchers, can no longer consider storage devices as black boxes that respect a simple performance contract. We have to dig into their internals in order to understand the impact of these devices on system design. Here is a bottom up review of the IO stack with flash-based SSDs. We discuss PCM in Section 2.4.

Flash chip: A flash chip is a complex assembly of flash cells[1], organized by pages (512 to 4096 bytes per page), blocks (64 to 256 pages per block) and sometimes arranged in multiple *planes* (typically to allow parallelism across planes). Operations on flash chips are read, write (or program) and erase. Due to flash cells characteristics, these operations must respect the following constraints: (C1) reads and writes are performed at the granularity of a page; (C2) a block must be erased before any of the pages it contains can be overwritten; (C3) writes must be sequential within a block; (C4) flash chips support a limited number of erase cycles.

[1] See [5] for a discussion of flash cells internals.

The trends for flash memory is towards an increase (i) in density thanks to a smaller process (today 20nm), (ii) in the number of bits per flash cells, (iii) of page and block size, and (iv) in the number of planes. Increased density also incurs reduced cell lifetime (5000 cycles for triple-level-cell flash), and raw performance decreases. For now, this lower performance can be compensated by increased parallelism within and across chips. At some point though, it will be impossible to further reduce the size of a flash cell. At that point, PCM might be able to take over and still provide exponential growth in terms of density.

Flash SSD: A flash-based SSD contains tens of flash chips wired in parallel to the SSD controller though multiple channels. Flash chips are decomposed into logical units (LUN). LUNs are the unit of operation interleaving, i.e., operations on distinct LUNs can be executed in parallel, while operations on a same LUN are executed serially. We consider that SSD performance is *channel-bound* if channels are the bottleneck and IOs wait for a channel to be available before they can be executed. SSD performance is *chip-bound* if chip operations are the bottleneck and IOs wait for a chip operation to terminate before they can be executed. Figure 1 illustrates these notions on an example.

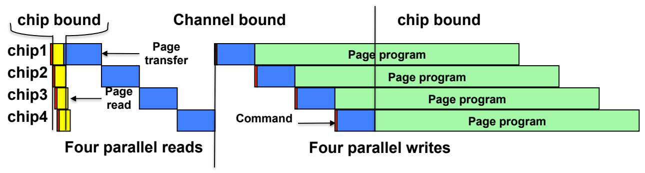

Figure 1: Example of channel transfer and chip operations on four chips (we assume 1 LUN per chip) attached to the same channel.

SSD controller: The SSD controller embeds the so-called Flash Translation Layer (FTL) that maps incoming application IOs –a read, a write or a trim[2] on a logical block address (LBA)– into flash chip operations. As illustrated on Figure 2, FTL is responsible for:

- *Scheduling & Mapping:* The FTL provides a virtualization of the physical address space into a logical address space. This mapping is done at the page (and possibly block) level. The FTL implements out-of-place updates (copy-on-write) to handle C2 and C3. It also handles chip errors and deals with parallelism across flash chips. While each read (resp. trim) operation is mapped onto a specific chip, each write operation can be scheduled on an appropriate chip.

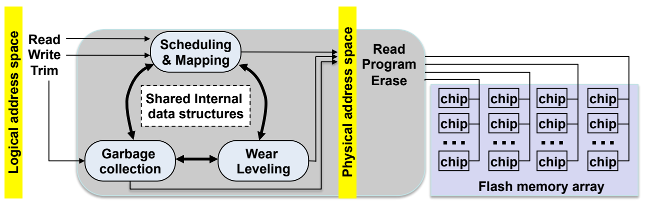

Figure 2: Internal architecture of a SSD controller

- *Garbage Collection:* Each update leaves an obsolete flash page (with a before image). Over time, obsolete flash pages accumulate, and are reclaimed through garbage collection.
- *Wear Leveling:* The FTL relies on wear-leveling to address C4--distributing the erase counts across flash blocks and masking bad blocks.

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

Note that both garbage collection and wear leveling read live pages from a victim block and write those pages (at a location picked by the scheduler), before the block is erased. The garbage collection and wear leveling operations thus interfere with the IOs submitted by the applications.

OS Driver: SSDs are not directly accessible from the CPU; the operating system provides a driver that manages communications to and from the device. Most SSDs implement a SATA interface and are accessed via the generic SATA driver. Some high-end SSDs (e.g., ioDrive from FusionIO) are directly plugged on the PCI bus. They provide a specific driver, which implements part of the SSD controller functionalities (leveraging CPU and RAM on the server to implement part of the FTL).

Block Layer: The block layer provides a simple memory abstraction. It exposes a flat address space, quantized in logical blocks of fixed size, on which I/O (read and write) requests are submitted. When an I/O request is submitted, it is associated to a completion queue. A worker thread then sends a page request to the disk scheduler. When the page request completes, an interrupt is raised (within the device driver), and the I/O request completes. In the last few years, the Linux block layer has been upgraded to accommodate SSDs and multi-cores. CPU overhead has been reduced– it was acceptable on disk to reduce seeks –, lock contention has been reduced, completions are dispatched on the core that submitted the request, and currently, the management of multiple IO queues for each device is under implementation.

Is it still reasonable to hide all this complexity behind a simple memory abstraction? Let us now revisit the performance assumptions popular in the database community.

2.3 DEBUNKING SSD MYTHS

(1) SSDs behave as to the non-volatile memory they contain.

Hopefully, the previous section will have made it very clear that this statement is not true. We pointed out this confusion in [6]. Still, two years later, we proposed a bimodal FTL that exposed to applications the constraints of a single flash chip [4]. We ignored the intrinsic parallelism of SSDs and the necessary error management that should take place within a device controller. Exposing flash chip constraints through the block layer, as we proposed, would in effect suppress the virtualization of the physical flash storage. This would limit the controller's ability to perform garbage collection and wear leveling (as it could not redirect the live pages of a victim block onto other chips) and its ability to deal with partial chip failures. It would also put a huge burden on the OS block layer if the application aimed at efficiently leveraging SSD parallelism by scheduling writes on multiple chips. Today, papers are published that attribute the characteristics of a phase-change memory chip to a SSD, thus ignoring that parallelism and error management must be managed at the SSD level.

(2*) On flash-based SSDs, random writes are very costly and should be avoided***.**

While this statement was true on early flash-based SSDs, it is no longer the case [2,3,5]. There are essentially two reasons why a flash-based SSD might provide random writes which are as fast as, or even faster than sequential writes. First, high-end SSDs now include safe RAM buffers (with batteries), which are designed for buffering write operations. Such SSDs provide a form of write-back mechanism where a write I/O request completes as soon as it hits the cache. Second, modern SSD can rely on page mapping, either because mapping is stored in the driver (without much RAM constraints), or because the controller supports some form of efficient page mapping cache [10]. With page mapping, there are no constraints on the placement of any write. Thus a controller can fully benefit from SSD parallelism when flushing the buffer regardless of the write pattern! An interesting note is that random writes have a negative impact on garbage collection, as locality is impossible to detect for the FTL. As a result, pages that are to be reclaimed together tend to be spread over many blocks (as opposed to sequential writes where locality is easy to detect). Quantifying these effects is a topic for future work. To sum up, the difference between random writes and sequential writes on flash-based SSDs is rather indirect. We completely missed that point in [4], where we ventured design hints for SSD-based system design.

(3) *On flash-based SSDs, reads are cheaper than writes*.

While at the chip level reads are much faster than writes, at the SSD level this statement is not necessarily true. First, for reads, any latency or delay in the execution leads to visible latency in the application. It is not possible to hide this latency behind a safe cache, as it is the case for writes. So if subsequent reads are directed to a same LUN and, if that LUN or the associated channel is busy, then the read operation must wait (e.g., wait 3ms for the completion of an erase operation on that LUN)! Third, reads will benefit from parallelism only if the corresponding writes have been directed to different LUNs (on different channels). As we have seen above, there is no guarantee for this. Fourth, reads tend to be channel-bound --while writes tend to be chip-bound --, and channel parallelism is much more limited than chip parallelism.

2.4 DISCUSSION

It is unlikely that the complexity of flash-based SSDs can be tamed into a simple performance model behind the block device interface. So what should we do? An option is to wait for the OS and storage communities to define such a model. In the meantime, we should stop publishing articles based on incorrect assumptions. Another option is to skip the complexity of flash-based SSDs and wait for PCM to take over, as the characteristics of PCM promise to significantly reduce complexity (in-place updates, no erases, on-chip error detection, no need for garbage collection). First, there is a large consensus that PCM chips should be directly plugged onto the memory bus (because PCM is byte addressable and exhibits low latency) [8,16]. The capacity of each PCM chip is unlikely to be much larger than RAM chips. That still leaves us with the problem of secondary storage. Second, PCM is likely to be integrated into flash-based SSDs, i.e., to expand buffer capacity and performance. As a result, flash-based SSDs are unlikely to disappear any time soon. Third, even if we contemplate pure PCM-based SSDs [1], the issues of parallelism, wear leveling and error management will likely introduce significant complexity. Also, PCM-based SSDs will not make the issues of low latency and high-parallelism disappear. More generally, PCM and flash mark a significant evolution of the nature of the interactions between CPU, memory (volatile as well as non-volatile) and secondary storage. This is an excellent opportunity to revisit how database systems interact with secondary storage.

3. SECONDARY STORAGE REVISITED

For years, we have assumed that persistence was to be achieved through secondary storage, via a memory abstraction embodied by the block device interface. The advent of flash and PCM force us to reconsider this assumption:

1. We can now achieve persistence through PCM-chips plugged on the memory bus and directly addressable by the CPU [7], in addition to secondary storage, composed of SSDs.
2. Flash-based SSDs are no longer accessed via a strict memory abstraction. The TRIM command has been added to read and write to make it possible to applications to communicate to a SSD that a range of logical addresses were no longer used and could thus be un-mapped by the FTL. SSD constructors are now proposing to expose new commands, e.g., atomic writes [17], at the driver's interface. More radically, FusionIO is now proposing direct access to its driver, entirely bypassing the block layer (ioMemory SDK). The point here is that the block device interface provides too much abstraction in the absence of a simple performance model.

This evolution forces us to re-visit the nature of persistence in database systems. We see three fundamental principles:

- *We should keep synchronous and asynchronous patterns separated*, as Mohan suggested [16]. Until now, database storage managers have implemented conservative asynchronous I/O submission policies to account for occasional synchronous I/Os [13]. Instead synchronous patterns (log writes, buffer steals under memory pressure) should be directed to PCM-based SSDs via non-volatile memory accesses from the CPU, while asynchronous patterns (lazy writes, prefetching, reads) should be directed to flash-based SSDs via I/O requests.
- *We should abandon the memory abstraction in favor of a communication abstraction to manage secondary storage*, as we suggested in [4]. The consequence is that (a) the database system is no longer the master and secondary storage a slave (they are communicating peers), and (b) the granularity of interactions is not limited to blocks. This has far reaching consequences on space allocation and naming (extent-based allocation is irrelevant, nameless writes are interesting), the management of log-structured files (which is today handled both at the database level and within the FTL), garbage collection and wear leveling. Interestingly, Jim Gray noted in [11] that RAM locality is king. An extended secondary storage interface would allow us to efficiently manage locality throughout the I/O stack.
- *We should seek inspiration in the low-latency networking literature.* Secondary storage is no longer a bottleneck in terms of latency, and it requires parallelism to reach nominal bandwidth. A similar evolution has been witnessed for some years in the networking community, where the advent of 10/40/100 GB Ethernet, forced them to tackle the problems caused by low-latency. The solutions they explored including cross-layer design, shared memory, and FPGAs are very much relevant in the context of a re-designed I/O stack.

Note that any evolution of the role of secondary storage will take place in the context of multi-core CPUs. So, the staging architecture [12], based on the assumption that all data is in-memory, should be the starting point for our reflection.

Why don't we let the OS community redefine the IO stack? Well, they are not waiting for us. Proposals are flourishing for PCM-based [1,9,7], flash-based [14] and even database [15] systems. Note that these approaches are based on actual storage hardware and complete system design. We argue that it is time for database system researchers to engage other systems communities to contribute to the on-going re-design of the I/O stack and re-think the role of persistence in database systems.

4. CONCLUSION

In this paper, we established that the database systems research community has a flash problem. We argued that the high-level of abstraction provided by the block device interface is a significant part of the problem. We joined the choir of those who preach a re-design of the architecture of (single-site) database systems. We argued that we ignore the evolution of secondary storage at our own peril. First, because some of the assumptions we are making are myths rather than sound results. Second, because the on-going re-design of the I/O stack is an opportunity for intriguing research.

5. ACKNOWLEDGEMENTS

This work is funded by the Danish Council for Independent Research CLyDE project.

6. REFERENCES

[1] A. Akel, A. Caulfield, T.Mollov, R.Gupta, S. Swanson. Onyx: A Prototype Phase Change Memory Storage Array. HotStorage 2011.

[2] M. Bjørling, P. Bonnet, L. Bouganim, and B. T. Jònsson. Understanding the energy consumption of flash devices with uFLIP. IEEE Data Eng. Bull., December, 2010.

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

[4] P. Bonnet and L. Bouganim. Flash Device Support for Database Management. CIDR, 2011.

[5] P. Bonnet, L. Bouganim, I. Koltsidas, S. Viglas. System Co-Design and Data Management for Flash Devices. VLDB 2011.

[6] L. Bouganim, B. T. Jònsson, and P. Bonnet. uFLIP: Understanding flash I/O patterns. CIDR, 2009.

[7] A. Caulfield, T. Mollov, L. Eisner, A. De, J. Coburn, S. Swanson: Providing safe, user space access to fast, solid state disks. ASPLOS 2012.

[8] S. Chen, P. Gibbons, S. Nath: Rethinking Database Algorithms for Phase Change Memory. CIDR 2011.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, D. Coetzee: Better I/O through byte-addressable, persistent memory. SOSP 2009.

[10] 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.

[11] J. Gray. Tape is dead, disk is tape, flash is disk, RAM locality is king. Pres. at the CIDR Gong Show, Asilomar, CA, USA, 2007.

[12] Stavros Harizopoulos, Anastassia Ailamaki: A Case for Staged Database Systems. CIDR 2003

[13] Christoffer Hall, Philippe Bonnet: Getting Priorities Straight: Improving Linux Support for Database I/O. VLDB 2005.

[14] H. Lim, B. Fan, D. Andersen, M. Kaminsky: SILT: a memory-efficient, high-performance key-value store. SOSP 2011

[15] M. Mammarella, S. Hovsepian, E. Kohler: Modular data storage with Anvil. SOSP 2009:147-160

[16] C.Mohan, S.Bhattacharya. Implications of Storage Class Memories on Software Architectures. HPCA Workshop on the Use of Emerging Storage and Memory Technologies. 2010

[17] X. Ouyang, D. W. Nellans, R. Wipfel, D. Flynn, D. K. Panda: Beyond block I/O: Rethinking traditional storage primitives. HPCA 2011.

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

[19] M. Stonebraker. Operating system support for database management. Commun. ACM, 24(7):412–418, 1981.