Native Storage Techniques for Data Management

Ilia Petrov *1, Andreas Koch #2, Sergey Hardock #3, Tobias Vincon *4

* *Reutlingen University, Germany*
1 ilia.petrov@reutlingen-university.de,
4 tobias.vincon@reutlingen-university.de

Technische Universität Darmstadt, Germany
2 koch@esa.tu-darmstadt.de,
3 hardock@dvs.tu-darmstadt.de

***Abstract*—In the present tutorial we perform a cross-cut analysis of database storage management from the perspective of modern storage technologies. We argue that neither the design of modern DBMS, nor the architecture of modern storage technologies are aligned with each other. Moreover, the majority of the systems rely on a complex multi-layer and compatibility-oriented storage stack. The result is needlessly suboptimal DBMS performance, inefficient utilization, or significant write amplification due to outdated abstractions and interfaces. In the present tutorial we focus on the concept of *native storage*, which is storage operated without intermediate abstraction layers over an open native storage interface and is directly controlled by the DBMS. We cover the following aspects of native storage: (i) architectural approaches and techniques; (ii) interfaces; (iii) storage abstractions; (iv) DBMS/system integration; (v) in-storage processing.**

I. Structure and Organization of The Tutorial

We propose a 1.5 hour tutorial on the above topic. The scope can be optionally extended to 3 hours to cover *In-Storage Processing*. The target audience is database researchers and practitioners with interests in storage management on modern storage hardware. The proposed tutorial is original work and as such has not been presented elsewhere. The contents and the expected duration are outlined below and described in detail in Section II.

- Brief overview of modern storage technologies and the classical I/O stack (10 min.)
- Native Storage (5 min.)
- Architectural Approaches to Native Storage (15 min.)
- Native Storage Interfaces (20 min.)
- Storage Abstractions for Native Storage (15 min.)
- System Integration (20 min.)
- In-Storage Processing (90 min. – *in case of a 3h tutorial*)
- Wrap-up (5 min.)

II. Tutorial Outline

In the present tutorial we examine the influence of *native storage* on data-intensive systems and data management. We begin with a succinct description of the concept of native storage and a brief summary of the characteristics of modern storage technologies. The main focus of this tutorial is on their influence on different aspects of data management. *Firstly*, we describe different architectural approaches and techniques for defining and using native storage. Existing native storage systems pursue different goals and result in various architectural blueprints. These range from small sensor data or graph processing to enterprise scenarios and in-storage processing. *Secondly*, we present a cross-cut analysis of different proposals for native storage *interfaces*. These are very different from the currently dominating compatibility block-device interface. *Thirdly*, novel storage abstractions are needed with open native storage interfaces in place. *Fourthly*, we analyze the different approaches for system integration of native storage and native storage interfaces. *Last but not least*, having considered all these aspects in isolation we provide a wrap-up in terms of data management techniques for native storage.

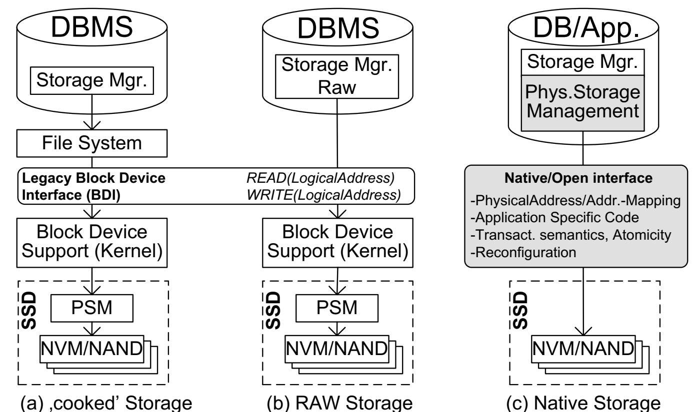

Fig. 1. DBMS storage alternatives: (a) Traditional cooked DBMS storage; (b) DBMS on RAW volumes/devices; (c) DBMS on Native Storage

III. Introduction

Over the last decade we witnessed several important breakthroughs in storage technologies: Flash and Flash SSDs have become omnipresent as database storage; Non-Volatile Memories (e.g., PCM or Intel/Micron's 3D XPoint) are gradually becoming real. Although these have very different characteristics from old-fashioned magnetic HDD storage, they are still treated as fast replacements and are embedded in a classical *"cooked"* storage stack. This typically comprises multiple layers: a low-level compatibility translation layer (typically running on-device), a block-device, OS kernel support, and a file system [1], [2]. In fact, file-systems are considered part

of the NVM I/O stack even for non-volatile main-memory settings [3], [4]. Even though a compatibility stack fosters proliferation of modern storage technologies and simplifies systems development, it has a number of disadvantages [5]–[8]: (i) performance lags and underutilization of SSD resources ([2], [8]–[10] report lags of several times); (ii) no cross-layer optimizations due to layered abstractions and information hiding as well as the rigidness of compatibility interfaces; (iii) I/O (write-/read-) amplification of several times in terms of size and count, leading to lower performance and faster wear [8], [11]; (iv) functional redundancy along the I/O stack [2], [11]; (v) inability to configure the semiconductor storage adaptively (depending on the application or the workload) [1], [12], [13]. In fact, with all of the above in place, [1] reports that only 40% of the raw Flash bandwidth is delivered to applications and that to achieve high performance only, 50%-70% of the raw capacity is effectively available to applications (the rest is reserved for write handling and error correction).

A. Architectural Approaches and Techniques

In pursuit of a better I/O stack design, the use of alternative architectural blueprints seem unavoidable [5]–[7]. Opening up storage confronts systems and applications with different physical/low-level abstractions, evolving interfaces, and the question of how to distribute physical storage management along the stack. *Physical storage management – PSM* (e.g. FTL for Flash SSDs) encompasses: (i) the logical-to-physical address mapping (L2PAM); (ii) wear-levelling; (iii) error correction (ECC) and bad-block management; (iv) physical metadata management, (v) physical garbage collection and storage optimizations; and (vi) write management. In a traditional I/O stack, *PSM* is typically performed on-device, hidden behind the compatibility block-device interface.

Various native storage architectures exist. LightNVM and open-channel SSDs envisage a host-based, shared L2PAM table and handle physical metadata management, GC and write optimizations, as well as wear-levelling, on the host as part of the LightNVM subsystem. BlueDBM [14] and NoFTL [2] likewise assume a native storage, which however is extensible. NoFTL [2] and NoFTL-KV [15] have proposed open native storage and follow the approach of deep database integration, which explores coherent integration of PSM in different modules of the DBMS and investigates algorithmic improvements and cross-layer optimizations. Architecturally, native storage can be realized as host-based storage [2], [15] or distributed storage [14], [16], [17].

Early proposals for an I/O stack redesign gravitate around the concept of *bimodal* [6] or *multi-modal storage*. Under *bimodal storage* [6] if the DBMS issues "constrained" I/O patterns, i.e. no in-place updates and no random writes, the SSD uses a minimal FTL, while for all "unconstrained" I/O patterns, the SSD switches to traditional full-scale FTL [18]. Application Managed Flash (AMF) [19] explores append-based storage management over an extended block-device interface under a novel append-based file system. AMF explores the architectural coupling of free space management to physical GC, in the same time leaving ECC, bad-block management, and wear leveling on device. [9], [11], [20] assume partial exposure of the mapping (which still resides on device) to applications. Their design goals are to handle storage virtualization [9], storage management for append-mostly systems [11], or to explore transactional atomicity [20]. Some complementary aspects of native storage architectures are *reconfigurability* and *intelligent storage/In-Storage Processing (ISP)* [21]–[28]. In-situ execution of data processing operations minimizes data movement, leverages internal storage characteristics (parallelism, bandwidth, on-device CPU/FPGA), to achieve performance improvement of several times as well as better resource and energy efficiency.

B. Interfaces

Current hardware and software interfaces to memory and storage are rudimentary. Such interfaces: (i) have limited support for parallelism and concurrency, hence they limit system bandwidth and throughput; (ii) are relatively rigid with limited extensibility mechanisms; (iii) have only a limited set of capabilities and expose outdated abstractions. Building on top of such outdated abstractions demands layers of backwards compatibility, which prevent algorithms and system architectures from efficiently using modern storage technologies as their true characteristics are masked. On the hardware level, for instance, the traditional RAS/CAS DRAM architecture offers limited parallelism, while increasing the number of DIMMs per channel typically decreases performance. Similar behavior is exhibited by SATA. Block Device Interfaces (BDI) and block I/O are ubiquitous, yet they are a major bottleneck [5], [7] as they do not match the properties of modern storage technologies and require: immutable logical addresses; fixed I/O granularity; a rigid set of operations, mainly read/write; symmetric and wear-proof storage.

The *basic native storage interface* typically comprises *READ_PAGE*, *WRITE_PAGE* and *ERASE_BLOCK* commands [2], [14], [29]. These are defined on *physical addresses*. Since overwriting is an issue on modern storage technologies (wear, erase-before-overwrite), physical addresses need to change. This is a stark contrast to the BDI commands that rely on immutable *logical addresses* (LBA), and raises the issue of logical-to-physical address mapping (L2PAM). Various extensions to the basic interface are proposed by different systems. LightNVM [29], for example, suggests that the above are vectored commands, i.e. they take sets or ranges of physical addresses as arguments, instead of a single address. NoFTL [2] suggests extensions such as *write_delta* for writes of sub-page granularity, *copyback* to reduce data transfers incurred by the garbage collector, or *get_addr_table* to speed-up L2PAM table recovery.

I/O atomicity is an open issue in the traditional I/O stack, but it becomes viable with native storage and modern storage technologies. The key is the ability to control the L2PAM table and GC so that old physical pages and their address mapping entries are retained, while new contents are being written on a *different* physical location. Only if the write sequence, are

succeeds the old address mappings completely and atomically replaced with the new ones. I/O atomicity has inspired various architectural designs regarding *copy-on-write* (CoW) storage management and logging. [7] introduced a new I/O primitive "atomic-write", however without support for concurrency. [20] suggests transaction-awareness, allowing the DBMS to notify the SSD about the beginning and end of transactions. The *SHARE* [11] interface to Flash targets atomicity and CoW and proposes the *share(*$LogicalAddr_1$*,* $LogicalAddr_2$*)* command to allow two logical page numbers to be mapped on the same same physical page. *SHARE* is defined to be variant of *TRIM* that has also been explored in ANViL [9] predecessors called *ptrim()*.

Another aspect is the management of the *logical-to-physical address mapping*. With native storage it can be: (a) host- or device resident, depending on the available resources; (b) completely exposed (and managed by the application), or partially exposed (through special commands) but managed by PSM. [2], [14], [29] assume full exposure and host-based L2PAM table management. [2] investigates full DBMS integration. [9], [11], [20] assume partial exposure of the mapping (which still resides on device) to applications. This yields new commands and abstractions. ANViL [9] considers exposing the logical-to-physical address mapping table to applications and proposes commands such as *clone()*, *move()* or *delete()*.

In-Storage Processing (ISP), targets the execution of application/system specific functionality in-situ, and is an additional factor for interface extensions. [21] defines a new session-based DBMS-SmartSSD communication protocol, comprising operations like OPEN, CLOSE, GET, and a set of APIs for on-device functionality, such as Command API, Thread API, Data API and Memory API. Willow [16] proposes similar concepts for a user-programmable SSD. IBEX [27], [28] investigates a DB-record based interface.

C. Abstractions

As an open native storage interface replaces traditional BDI, the different physical organization of native storage is exposed to the DBMS. Typically, native storage comprises chips, channels, the on-device controller, and its resources. Adapting storage management and data processing for this type of organization is non-trivial. Therefore there is a pressing need for new storage abstractions that ease the DBMS integration of native storage.

AMF [19] assumes flash *blocks* and contiguous *segments*. Segments are introduced as a unit of allocation and physical distribution to achieve better bandwidth. More importantly AMF segments need to be explicitly deallocated by *TRIM* to physically reclaim the occupied space. A segment is subdivided into *sectors* that are a unit of I/O. As AMF targets append-based storage, a sector can never be overwritten (unless the whole segment is deallocated).

NoFTL introduces the concepts of *regions* and *groups* to manage native storage [30]. A *NoFTL region* comprises a set of physical chips and data channels as well as physical storage management strategies, such as address management, garbage collection and data placement. Regions are coupled to standard DBMS logical storage structures such as segments or tablespaces. A storage device is thus viewed and maintained by the DBMS as a set of regions. Every database object is then assigned to a certain region based on its properties, while every region can hold multiple objects (i.e. one-to-many relationship). *Groups* [30] serve as means to improve hot/cold data separation and thus decrease unnecessary GC activity, reduce erases and improve performance and longevity.

ANViL [9] proposes the *snapshot* at the level of a volume or a file as a native storage abstraction. A snapshot allows to checkpoint the state of a file/volume with little space and performance overhead, as only mapping entries are cloned. ANViL [9] also introduces *deduplication* as an abstraction based on *range cloning* that identifies and collapses identical blocks.

D. System Integration

There are different approaches to integrate native storage and physical storage management (PSM) into the DBMS or other applications. We distinguish *partial integration* and *deep integration*. Many approaches offer non-intrusive *partial integration* of native storage and rely on multi-modal native storage interfaces. Such systems tend to preserve existing I/O interface, however they also incorporate new features as extensions. Systems such as AMF [19], SHARE [11], XFTL [20] represent partial integration. Some of the advantages are the high degree of reuse, proliferation, and low implementation footprint. Furthermore, some of the native storage systems such as BlueDBM [14] or ANViL [9] offer multi-modal interfaces, leaving it to application to decide which one to use.

Systems such as [2], [31] support *deep integration*, i.e. the PSM is coherently integrated into different modules of the system. The key insight is that deep integration results in a surprisingly simple and lightweight implementation. This is the case, since many DBMS sub-systems *already* implement similar functionality, which only needs to be leveraged and extended for deep integration.

E. Reconfigurability

Storage built on top of semiconductor storage technologies can be dynamically reconfigured depending on the workload. This type of *reconfigurability* has been initially explored in [12], [13] in compatibility storage settings. [1], [16] investigate reconfigurability based on native storage. NoFTL [32] offers an advisor to derive region properties from I/O access patterns to different DB-objects.

F. In-Storage Processing (in case of a 3 hour tutorial)

The ability to execute application/system specific functionality in-situ (*In-Storage Processing – ISP*) is a very relevant trend and a revival of past ActiveDisc/DatabaseMachines efforts. [21] is one of the first works to explore offloading parts of data processing on Smart SSDs, indicating the potential of significant performance improvements of up to 2.7x and

energy savings of up to 3x. [21] defines a new session-based DBMS-SmartSSD communication protocol, comprising operations like OPEN, CLOSE, GET, and a set of APIs for on-device functionality, such as Command API, Thread API, Data API and Memory API. Willow [16] proposes similar concepts for a user-programmable SSD. [21] identifies two research questions: (i) how can ISP handle the problem of on-device processing in the presence of a more recent version of the data in the buffer; and (ii) what is the efficiency of operation pushdown in the presence of large main memories.

The initial ideas of [21] have recently been extended in a complementary approach called *In-Storage Processing/Computing* [22]–[24]. [22] demonstrates a performance improvement of 5x and 47x for scans and joins on embedded CPUs. Further approaches stress the importance of in-situ analytical processing on on-device stream processors or embedded CPUs [25], [26]. Still, all of the above target *read-only ISP*, assuming that the on-device data is immutable.

IBEX investigates how data processing on FPGAs can be used as explicit co-processors, or implicitly as part of an intelligent storage system [28]. IBEX exploits reconfigurable computing and the capabilities of custom-hardware to accelerate certain database operations. Yet, operations are not performed *in-situ* as data and results need to be transferred form storage to the FPGA and vice versa.

G. Data Management on Native Storage

In this part of the tutorial we provide a brief overview on recent solutions in the industry and academia for architecting, organizing, and utilizing native storage. The two major directions here are (i) the utilization of storage-specific out-of-place update strategy, as well as (ii) usage of database semantics for optimizations in FTL (e.g., reconfigurable/reprogrammable SSDs).

IV. Biographies of the Presenters

Ilia Petrov is a professor at Reutlingen University, Germany and a head of the Data Management Lab. His research focus is on data management on modern hardware. He holds a Ph.D. from the University of Erlangen-Nürnberg.

Andreas Koch is a professor at the Technische Universität Darmstadt, Germany and head of the Embedded Systems and Applications Group (ESA). His research interests are in the area of specialized computing systems ranging from low power embedded systems up to accelerators for data-center high-performance computers.

Sergej Hardock is a Ph.D. student at the Databases and Distributed Systems Group at the Technische Universität Darmstadt, Germany. His research interests are in database systems on modern hardware, native Flash database storage, lean Flash-aware database systems.

Tobias Vincon is a Ph.D. student at the Data Management Lab at Reutlingen University, Germany and at the Technische Universität Darmstadt. His research interests are in database systems on modern hardware and Near-Data Processing.

References

[1] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, “Sdf: Software-defined flash for web-scale internet storage systems,” in *Proc. ASPLOS*, 2014.

[2] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, “Noftl: Database systems on ftl-less flash storage,” *Proc. VLDB*, vol. 6, no. 12, 2013.

[3] H. Volos, S. Nalli, S. Panneerselvam, and et al., “Aerie: Flexible file-system interfaces to storage-class memory,” in *Proc. EuroSys*, 2014.

[4] J. Arulraj, A. Pavlo, and S. R. Dulloor, “Let’s talk about storage & recovery methods for non-volatile memory database systems,” in *Proc. SIGMOD*, 2015.

[5] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan, “The necessary death of the block device interface,” in *Proc. CIDR*, 2013.

[6] P. Bonnet and et al., “Flash device support for database management,” in *Proc. CIDR*, 2011.

[7] X. Ouyang, D. W. Nellans, R. Wipfel, and D. Flynn, “Beyond block i/o: Rethinking traditional storage primitives,” in *Proc. HPCA*, 2011.

[8] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, “I/o stack optimization for smartphones,” in *Proc. USENIX/ATC*, 2013.

[9] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Anvil: Advanced virtualization for modern non-volatile memory devices,” in *Proc. USENIX/FAST*, 2015.

[10] F. Chen, B. Hou, and R. Lee, “Internal parallelism of flash memory-based solid-state drives,” *ACM ToS*, vol. 12, no. 3, 2016.

[11] G. Oh, C. Seo, S.-W. Lee, and et al., “Share interface in flash storage for relational and nosql databases,” in *Proc. SIGMOD*, 2016.

[12] J.-Y. Shin, Z.-L. Xia, and et al., “Ftl design exploration in reconfigurable high-performance ssd for server applications,” in *Proc. ICS*, 2009.

[13] A. Birrell, M. Isard, C. Thacker, and T. Wobber, “A design for high-performance flash disks,” *SIGOPS Oper. Syst. Rev.*, 2007.

[14] S.-W. Jun, M. Liu, S. Lee, Arvind, and et al., “Bluedbm: Distributed flash storage for big data analytics,” *ACM TOCS*, vol. 34, no. 3, 2016.

[15] T. Vincon, S. Hardock, C. Riegger, J. Oppermann, A. Koch, and I. Petrov, “Noftl-kv: Tackling write-amplification on kv-stores with native storage management,” in *Proc. EDBT*, 2018.

[16] S. Seshadri, M. Gahagan, and et al., “Willow: A user-programmable ssd,” in *Proc. OSDI*, 2014.

[17] Z. István, D. Sidler, and G. Alonso, “Caribou: Intelligent distributed storage,” *Proc. VLDB Endow.*, 2017.

[18] D. Ma, J. Feng, and G. Li, “A survey of address translation technologies for flash memories,” *ACM Comput. Surv.*, vol. 46, no. 3, 2014.

[19] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and A. Arvind, “Application-managed flash,” in *Proc. USENIX/FAST*, 2016.

[20] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, “X-ftl: Transactional ftl for sqlite databases,” in *Proc. SIGMOD*, 2013.

[21] J. Do, Y.-S. Kee, J. M. Patel, D. J. DeWitt, and et. al, “Query processing on smart ssds: Opportunities and challenges,” in *Proc. SIGMOD*, 2013.

[22] S. Kim, H. Oh, and et al., “In-storage processing of database scans and joins,” *Inf. Sci.*, 2016.

[23] I. Jo, D.-H. Bae, A. S. Yoon, and et al., “Yoursql: A high-performance database system leveraging in-storage computing,” *Proc. VLDB*, 2016.

[24] Samsung, “In storage computing,” 2015. [Online]. Available: ”http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150813_S301D_Ki.pdf”

[25] S. Cho, C. Park, H. Oh, and et al., “Active disk meets flash: A case for intelligent ssds,” in *Proc. ICS*, 2013.

[26] D. Tiwari, S. Boboila, and et al., “Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines,” in *Proc. FAST*, 2013.

[27] L. Woods, J. Teubner, and G. Alonso, “Less watts, more performance: An intelligent storage engine for data appliances,” in *Proc. SIGMOD*, 2013.

[28] L. Woods, Z. István, and G. Alonso, “Ibex: An intelligent storage engine with support for advanced sql offloading,” *Proc. VLDB*, 2014.

[29] M. Bjørling, J. González, and P. Bonnet, “Lightnvm: The linux open-channel ssd subsystem,” in *Proc. USENIX/FAST*, 2017.

[30] S. Hardock, I. Petrov, R. Gottstein, and A. P. Buchmann, “Revisiting DBMS space management for native flash,” in *Proc. EDBT*, 2016.

[31] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, and Arvind, “Bluecache: A scalable distributed flash-based key-value store,” in *Proc. VLDB*, 2016.

[32] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, “Selective in-place appends for real: Reducing erases on wear-prone dbms storage,” in *Proc. ICDE*, 2017.