NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management*

Tobias Vinçon
DXC Technology
DBLab, Reutlingen University
Reutlingen, Germany
tobias.vincon@dxc.com

Sergey Hardock
DVS, TU-Darmstadt
Darmstadt, Germany
hardock@dvs.tu-darmstadt.de

Christian Riegger
DBLab, Reutlingen University
Reutlingen, Germany
christian.riegger@
reutlingen-university.de

Julian Oppermann
ESA, TU Darmstadt
Darmstadt, Germany
oppermann@esa.tu-darmstadt.de

Andreas Koch
ESA, TU Darmstadt
Darmstadt, Germany
koch@esa.tu-darmstadt.de

Ilia Petrov
DBLab, Reutlingen University
Reutlingen, Germany
ilia.petrov@reutlingen-university.
de

ABSTRACT

Modern persistent Key/Value stores are designed to meet the demand for high transactional throughput and high data-ingestion rates. Still, they rely on backwards-compatible storage stack and abstractions to ease space management, foster seamless proliferation and system integration. Their dependence on the traditional I/O stack has negative impact on performance, causes unacceptably high write-amplification, and limits the storage longevity.

In the present paper we present NoFTL-KV, an approach that results in a lean I/O stack, integrating physical storage management natively in the Key/Value store. NoFTL-KV eliminates backwards compatibility, allowing the Key/Value store to directly consume the characteristics of modern storage technologies. NoFTL-KV is implemented under RocksDB. The performance evaluation under LinkBench shows that NoFTL-KV improves transactional throughput by 33%, while response times improve up to 2.3x. Furthermore, NoFTL-KV reduces write-amplification 19x and improves storage longevity by imately the same factor.

1 INTRODUCTION

Over the last decade, various specialized DBMSs have been intensively investigated to meet the demand of new workloads, applications or data models. Persistent Key/Value stores (KV-stores) are specialized for high-throughput and predominantly update-intensive, OLTP-style workloads.

KV-stores exhibit a characteristic *lightweight architecture*, simplifying the deployment and integration process for large infrastructures and lowering maintenance demand in production. *Scalability* is intrinsically supported in terms of partitioning and distribution schemes, making KV-stores an excellent choice for current data-center architectures. The *simplicity* of their interface (with *put* and *get*) as well as data model matches wide range of modern insert and update intensive applications running high-throughput OLTP-style workloads. Last but not least, the ability to *serve as DB-Engines* in traditional and modern NoSQL databases (e.g. MyRocks[13] or MongoRocks), allows for the *integration* as meta stores into applications and distributed file systems (e.g. Ceph[10]), or serve as a backend for OLTP services.

*Produces the permission block, and copyright information

Persistent KV-stores leverage the properties of modern hardware due to the lean architecture, interface and flexibility, yet native hardware support is rare. The majority of such KV-stores rely on backwards-compatible storage, to ease administration and foster proliferation. Furthermore, the use of file systems simplifies space management, support for various storage architectures and the embedding in existing data center environments. *The underlying assumptions are that: (1) files and file-based I/O are the appropriate storage abstractions, and (2) use of standard/compatibility interfaces (and abstractions) on each individual layer of the I/O stack does not harm performance.*

The traditional I/O stack was developed with the characteristics of HDDs in mind, with the block-device interface, block I/O operations and files as abstractions. New storage technologies such as Non-Volatile Memories or Flash exhibit very different characteristics. However, to utilize them, persistent KV-stores require multiple layers of backwards compatibility, having a negative impact on performance and longevity. (1) Hardware resources are not fully exploited because of the hardware-oblivious abstractions. (2) DBMS access patterns result in suboptimal physical I/O patterns due to the presence of multiple abstraction layers along the critical I/O path. (3) KV-store information about the current workload cannot be used for better physical data placement. (4) Functionality along this critical I/O path is redundant. Significant write-amplification and suboptimal performance are the inevitable consequences.

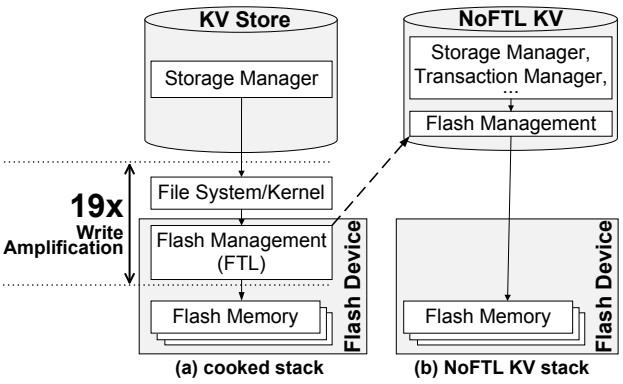


Figure 1: Write-Amplification along a traditional I/O stack in contrast to NoFTL-KV.

To verify the above claims we perform an experiment under RocksDB running LinkBench and measure the end-to-end write-amplification along a backwards-compatible, file-system based stack. The results (Fig. 2a) indicate a 19x physical write volume increase, lower performance and longevity.

In this paper we present NoFTL-KV (Fig. 1), an approach that avoids backwards compatibility and targets the above disadvantages by controlling the underlying physical storage directly. NoFTL-KV integrates physical storage (Flash) management natively in the KV-store. Subsequently, it opens up ways for workload adaptability within the storage layer and new abstractions for native storage.

The main contributions of this paper are:

(1) The extension of the concept of native storage management (NoFTL) to persistent KV-stores. We show that by coherently integrating address mapping, data placement, GC and free space management into the KV-store, storage characteristics, on-device parallelism and wear-leveling are addressed.
(2) NoFTL-KV is implemented under RocksDB.
(3) The performance evaluation under LinkBench[1] shows that NoFTL-KV improves the transactional throughput by 33%, while the response times improve up to 2.3x. Furthermore, NoFTL-KV improves physical storage management. In terms of *write efficiency*, NoFTL-KV performs 87% less physical page writes (including maintenance I/O and GC). Moreover, NoFTL-KV performs 19x less erases, improving the endurance by approximately the same factor.

The rest of the paper is structured as follows. NoFTL-KV and the integration into RocksDB are described in Section 3. Experimental results are discussed in Section 4. We conclude in Section 5.

2 RELATED WORK

Modern workloads (Social Media, Big Data or IoT) not only have become write-intensive and require high sequential throughput, but also demand low latencies [16]. *Read-* and *Write-Amplification* are major performance factors [16].

These can either be approached by utilising compression to decrease I/O in general [5] or by aligning better with the characteristics of modern storage devices. The latter is addressed in terms of either new data structures [3, 4, 19], or new software interfaces [2] as well as Flash interface extensions [9, 12, 14]. However, neither of those takes the issues with the *cooked stack* into account. [6] and [7] present a full integration of native storage support within traditional DBMS. A few lightweight KV-stores address the concept of direct native storage integration [8, 15, 17, 18] by moving the entire KV-store onto the device. Yet, physical storage management is only partially addressed.

With NoFTL-KV we address the deep integration of native storage management to tackle all issues regarding the traditional cooked stack while avoiding to overload the device controller with database functionality and maintaining a mature KV-store.

3 NOFTL-KV: NATIVE STORAGE KV-STORE

We investigate the concept of native storage management and NoFTL under persistent KV-stores to address and evaluate the above mentioned claims. RocksDB exhibits an append-only I/O pattern for various write-intensive workloads, because of its LSM-Tree-based persistent storage. LSM-trees perform regular compactions to remove old records, to ensure optimal tree structure and to perform hot-cold-separation. Compactions reorganize levels of the LSM-Tree, removing updated or deleted KV-Pairs, at regular intervals or given a certain threshold. As a consequence, frequently changing data is placed in the upper levels of the LSM-Tree, while the lower levels contain the cold data.

Under NoFTL-KV we pursue coherent integration of Flash management into existing modules of the KV-store as shown in Fig. 3. Firstly, NoFTL-KV has direct control over hardware resources through a native storage interface (NSI). NSI allows the DBMS to operate with I/O operations, in granularity and with addressing schemes supported by the the underlying storage technology. Furthermore, NSI eliminates the need to support backwards compatibility. Secondly, we revisit hardware-oblivious abstractions and propose using physical storage abstractions such as *Regions* to: (a) reduce read/write amplification along the I/O path, (b) utilize available I/O parallelism more efficiently, (c) provide better hot-cold data separation to (d) improve space management and (e) increase longevity.

Moreover, unnecessary DBMS data transfers can be reduced by pushing tasks down to the storage device. For instance, parts of garbage collection and compaction can be planned by the NoFTL-KV storage manager for certain Regions, but are executed onto the device to reduce I/O contention and data transfers. Likewise, queries, i.e scans, can be pushed down and executed on the storage device. Especially in combination with Regions, such queries can profit by the involved address mapping and level of on-device-parallelism. Also worth to mention is that processors on such storage devices usually exhibits the characteristics of common co-processor (ASIC or FPGA). These are perfectly aligned to the characteristics of modern storage technology (Flash, NVM) e.g. in respect to parallelism.

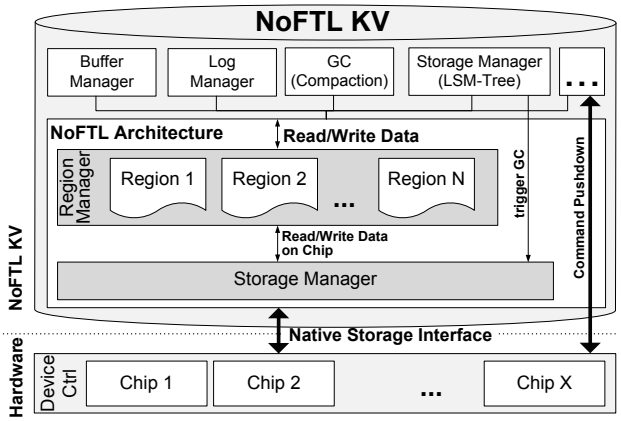
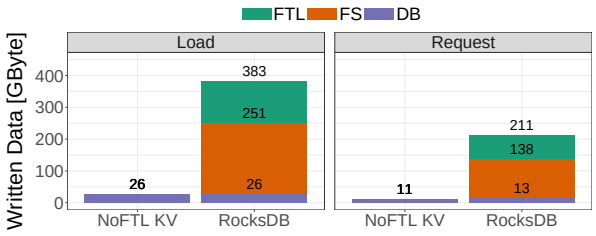


Figure 3: NoFTL-KV: Design of a deep integration of the NoFTL concept within an entire KV-store for native storage management

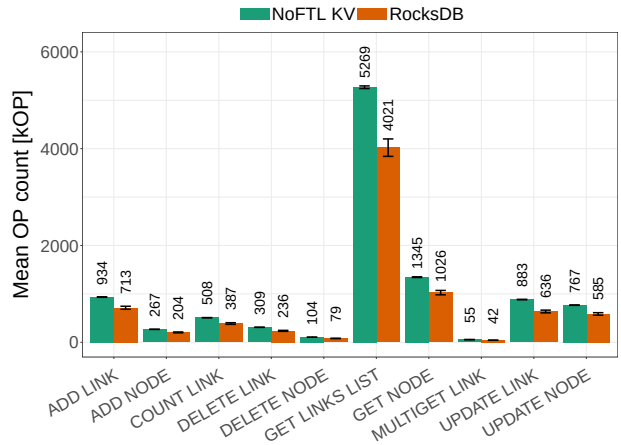
By integrating address mapping into the storage manager of the KV-store, the latter gets control over the physical data placement on Flash. Hence, the KV-store can utilize available information about data semantics, statistics and the access pattern (e.g., desired level of I/O parallelism) to perform efficient placement. Individual levels of the LSM-Tree can be physically separated on different chips to improve I/O throughput and parallelism since I/O-heavy compaction jobs do not block the entire device. Consequently, new storage abstractions can be defined besides files.



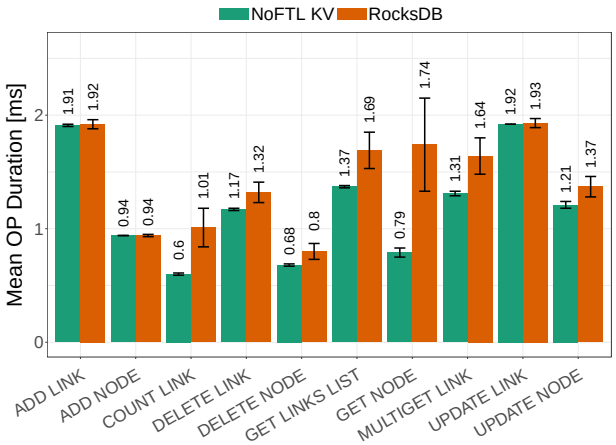
(a) Amount of data written by the DB, FS and FTL during the load and request phases of LinkBench shows the write-amplification of RocksDB in contrast to NoFTL-KV

	NoFTL-KV		RocksDB		Speedup
	Mean	StDev	Mean	StDev	
GC Calls	1	3	4769	2434	
GC Page Write	0	0	1932263	737453	
Block Erase	1127	3564	21389	7792	18.98x

(b) Number of logical page writes of the DB's compaction and physical page writes of the device over 10 request phases demonstrates write-amplification and inconsistent logical to physical page write ratio.



(c) Throughput: The average number of executed operations over the last 7 request phases demonstrates that NoFTL-KV outperforms RocksDB



(d) Response Time (the lower the better): Average operation latencies and std. dev. are better and more stable under NoFTL-KV vs. RocksDB

Figure 2: Results of the experimental evaluation of NoFTL-KV using LinkBench

We introduce *Regions* as physical storage abstractions spanning multiple chips/dies (i.e. parallel unit of the storage device). They can be effectively optimized for different access patterns (sequential, random, append) of various KV-store components (e.g. Levels of LSM-Tree and Log Manager). Regions allow for flexible physical storage management as the parameters of hot-cold data separation, garbage collection etc. are part of the definition. The level of supported I/O parallelism per Region can be defined in terms of the number of chips/dies it spans or whether these run in pseudo SLC, MLC or TLC Flash mode. Region definitions are not static, but can evolve over time to reflect properties of the workload.

```
CREATE REGION rgBlockMapping (
     MAX_CHIPS=4, MAX_CHANNELS=4,..., ADDR_MAPPING=BLOCK,
     NAND_MODE=MLC, ...);
CREATE TABLESPACE MyRocks.tblBlock (
     REGION=rgBlockMapping, UNIFORM EXTENT SIZE 128K );
CREATE TABLE MyRocks.nodetable(...)
     TABLESPACE MyRocks.tblBlock;
```

Furthermore, the functional redundancy along the *cooked I/O stack* is reduced. While, the file-system and the FTL distort the append-based access pattern and amplify the read/write data volume, NoFTL-KV simplifies the critical I/O path, exhibits a physically sequential I/O pattern, and offers better physical storage management. Consequently, write-amplification is significantly reduced. Similarly, the integration of the garbage collection within the compaction process of the LSM-Tree, allows for elimination of the time and resource-expensive merges common for traditional FTL-based SSDs. As a result, the KV-store is able to trigger the GC only when necessary and under the current workload. Higher longevity through less block erases and better throughput are the consequence.

4 EXPERIMENTAL EVALUATION

Testbed. Our testbed comprises a server equipped with an Intel Xeon E5-1620 v3 3.50 GHz CPU-core, 32GB RAM and an Intel DC 3600 SSD under Ubuntu 12.04 LTS, kernel 3.13.0.

The Flash storage device for the NoFTL-KV data is emulated by our real-time Flash Simulator[6], which is running as a kernel module. Configured with common latencies for reads, writes, and erases of current SLC NAND Flash it is able to simulate a modern enterprise SSD with either block- or char-device interfaces. For the block device, FASTer[11] is utilised as FTL with an over-provisioning area of 14%. In our setup, the simulator consumes 24 GB of memory to emulate an SSD of the same capacity with 256 pages (4KB) on 24576 blocks. The level of parallelism (emulated NAND chips/dies) is limited by the number of hardware threads. For our experiments we configured NoFTL-KV to only store RocksDB LSM-Tree files on the emulated device and the remaining files on the Intel DC 3600 formatted with an ext4 file system.

LinkBench. The experimental evaluation is performed using LinkBench[1], which is an OLTP-style workload on large updatable graphs. Under LinkBench the working data set size is an order of magnitude larger than the database buffer. The request phase of LinkBench comprises common graph queries like adding, getting, counting, deleting, updating nodes or edges on the graph. The experimental dataset is a graph of 15M nodes (initial), amounting to 15GB raw data. The number of requests and duration vary depending on the experiment. The baseline utilises the same configuration (ext3, 4KB blocksize, active journal) for MyRocks with RocksDB, hereinafter referred to as RocksDB.

Write-Amplification. To measure write amplification, hooks are placed in the storage engine of RocksDB (DB), the file system (FS), and the Flash emulator (FTL), i.e., in all layers along the I/O path. The number of requests is limited to 1M per thread with sufficient time (10h) to be executed completely. This ensures that, at the end, both variants have executed the same number of operations. The results (Fig. 2a) for NoFTL-KV and the baseline RocksDB represent average values of multiple runs.

Not surprisingly, the significant write amplification of the cooked stack becomes evident. The 26 GB of raw data, bulk-loaded during the *load phase*, swells up by more than 14 times to 383 GB. On top of that, the file system adds about 225 GB and the FTL increases this again by 132 GB. During the *request phase*, the disadvantages of the cooked I/O stack become even more visible. The average write-amplification here is more than 19x. This creates enormous I/O overhead, which is clearly reflected by the metrics to follow.

Throughput. The mean number of executed operations and their errors for every operation type is shown in Fig. 2c. NoFTL-KV outperforms RocksDB in every type of query. This is because of the smaller data volume to be written, and the better utilisation of available Flash parallelism. The workload of LinkBench has a high write-intensity over the complete duration. Consequently, the throughput increases about 31% accross all operation types. The performance stability across different runs, indicated by the error bars increases by an order of magnitude.

Response Time. To investigate the impact on response time for common operations, we perform further experiments with 1M requests per thread. Fig. 2d shows the average duration, while the error bars indicate the standard deviation.

One can clearly see that the latency is lower under NoFTL-KV. Especially reading operations like *GetNode()*, *GetLinksList()*, and *MultigetLink()* perform significantly better. This is even more relevant, since about 22% of these could not be served by database buffer (cache miss rate) and are read from the persistent device. On the other hand, inserting and updating operations like *AddNode()*, *AddLink()*, and *UpdateLink()* complete directly after pushing the data into an in-memory buffer and the WAL. This buffer is persistet only after a compaction, which is not taken into account for the operation latency. This explains the similar performance for both RocksDB and NoFTL-KV. The only exception is *UpdateNode()*, which might result in multiple gets that are slower with the traditional I/O stack.

Erases and Longevity. Write-amplification on Flash devices inevitably leads to more physical Flash erases, which has negative impact on device longevity. Table 2b captures the GC activity in terms of physical page writes and block erases during the benchmark runs of the previous experiment.

RocksDB performs almost 19 times the physical block erases than NoFTL-KV, which is primarily due to (i) the journal of the file system, which doubles Flash page writes, and (ii) FASTer's hybrid address mapping scheme. It is worth noting that the erase overhead of FASTer would also be present in other FTLs, which utilize hybrid address translation (common for current SSDs). As soon as the so-called log block area of the device runs out of space, the GC kicks in and merges the updated data with the corresponding Flash blocks in the data block area. Each of those merges requires multiple page migrations (on-device write amplification), and one or two erase operations (partial or full merges). NoFTL-KV is configured to use BLM, which matches the append-only LSM-Tree based storage management of RocksDB.

5 CONCLUSION

In the present paper we propose NoFTL-KV, an approach that results in a lean I/O stack, integrating physical storage management natively in the Key/Value store. NoFTL-KV eliminates backwards compatibility, allowing the Key/Value store to directly exploit the characteristics of modern storage technologies. NoFTL-KV is implemented under RocksDB and evaluated using LinkBench. The transactional throughput improves by 33%, while response times improve up to 2.3x. Furthermore, NoFTL-KV reduces write-amplification 19x and improves endurance. In addition, our current integration on the file-based LSM-Tree can be further improved by a deeper integration into the KV-store's data structure in future work to gain additional performance improvements.

REFERENCES

[1] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proc. SIGMOD 2013.*

[2] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. FAST 2017.*

[3] Niv Dayan, Philippe Bonnet, and Stratos Idreos. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proc. SIGMOD 2016.*

[4] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proc. SIGMOD 2011.*

[5] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing Space Amplification in RocksDB. In *Proc. CIDR 2017.*

[6] Sergej Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. NoFTL: Database Systems on FTL-less Flash Storage. In *Proc. VLDB 2013.*

[7] Sergey Hardock, Ilia Petrov, Robert Gottstein, and Alejandro P. Buchmann. NoFTL for Real: Databases on Real Native Flash Storage. In *Proc EDBT 2015.*

[8] Y. Jin, H. W. Tseng, Y. Papakonstantinou, and S. Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *In Proc. HPCA 2017.*

[9] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proc. SIGMOD 2013.*

[10] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. Understanding Write Behaviors of Storage Backends in Ceph Object Store. In *Proc. MSST 2017.*

[11] S. P. Lim, S. W. Lee, and B. Moon. FASTer FTL for Enterprise-Class Flash Memory SSDs. In *Proc. SNAPI 2010.*

[12] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-value Store. In *Proc. ATC 2015.*

[13] Yoshinori Matsunobu. InnoDB to MyRocks Migration in Main MySQL Database at Facebook. In *Proc. SREcon 2017.*

[14] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proc. SIGMOD 2016.*

[15] Samsung. http://www.samsung.com/semiconductor/global/file/insight/2017/08/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf

[16] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proc. SIGMOD 2012.*

[17] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. Proc. OSDI 2014.

[18] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A Scalable Distributed Flash-based Key-value Store. In *Proc. VLDB 2016.*

[19] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-structured File System to Exploit the Internal Parallelism of Flash Devices. In *Proc. ATC 2016.*