nKV: Near-Data Processing with KV-Stores on Native Computational Storage

Tobias Vinçon, Arthur Bernhardt, Ilia Petrov
[firstname].[surname]@reutlingen-university.de
DBlab, Reutlingen University

Lukas Weber, Andreas Koch
[surname]@esa.tu-darmstadt.de
ESA, Technische Universität Darmstadt

ABSTRACT

Massive data transfers in modern key/value stores resulting from low data-locality and data-to-code system design hurt their performance and scalability. Near-data processing (NDP) designs represent a feasible solution, which although not new, have yet to see widespread use.

In this paper we introduce nKV, which is a key/value store utilizing *native computational storage* and *near-data processing*. On the one hand, nKV can directly control the data and computation placement on the underlying storage hardware. On the other hand, nKV propagates the data formats and layouts to the storage device where, software and hardware parsers and accessors are implemented. Both allow NDP operations to execute in host-intervention-free manner, directly on physical addresses and thus better utilize the underlying hardware. Our performance evaluation is based on executing traditional KV operations (*GET*, *SCAN*) and on complex graph-processing algorithms (*Betweenness Centrality*) in-situ, with 1.4×-2.7× better performance on real hardware – the *COSMOS+* platform [22].

ACM Reference Format:
Tobias Vinçon, Arthur Bernhardt, Ilia Petrov and Lukas Weber, Andreas Koch. 2020. nKV: Near-Data Processing with KV-Stores on Native Computational Storage . In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3399666.3399934

1 INTRODUCTION

Besides substantial data ingestion, yielding an exponential increase in data volumes, modern data-intensive systems perform complex analytical tasks. To process them, systems trigger massive *data transfers* that impair performance and scalability, and hurt resource- and energy-efficiency. These are partly caused by the scarce bandwidth in combination with poor data locality, but also result from traditional (*data-to-code*) system architectures.

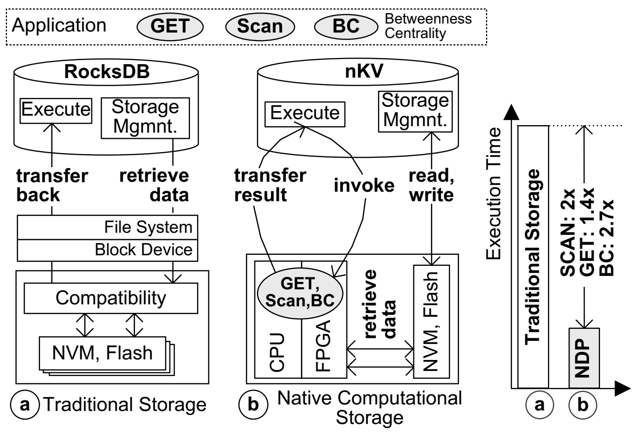

Figure 1: KV-Store transferring data along a traditional I/O stack (a); and (b) nKV executing operations in-situ on native computational storage.

Near-Data Processing (NDP) is a *code-to-data* paradigm targeting in-situ operation execution. In other words, operations are executed as close as possible to the physical data location, utilizing the much better on-device I/O performance. NDP leverages several trends. Firstly, hardware manufacturers can fabricate *combinations of storage and compute* elements economically, and package them within the same device – so called NDP-capable *computational storage*. As a result, even commodity storage devices nowadays, have compute resources that can be effectively used for NDP, but are executing compatibility firmware (to traditional storage) instead. Secondly, with semiconductor storage technologies (NVM/Flash), the *device-internal* bandwidth, parallelism, and latencies are significantly better than the external ones (device-to-host). Both lift major limitations of prior approaches like ActiveDisks [1, 24] or Database Machines [5].

Wide-spread, high-performance persistent key-value stores like LevelDB or RocksDB [9] tend to rely on a traditional layered-storage stack (Figure 1). It simplifies their architecture, allows for more flexibility and eases data management and administration. However, layers within the DBMS (e.g. Storage Manager or access methods), but also underlying the file- and operating system encapsulate information and functionality necessary for the successful utilization of NDP techniques. *Firstly*, NDP operations executed on-device require the physical address ranges of the data to be processed. In traditional storage, address information is scattered along the layers of the storage stack (DBMS, File System, OS) and hidden behind layers of abstraction (Figure 1). *Secondly*, NDP-operations need to navigate through and interpret the physical data on-device. To this end data formats and layout accessors are necessary on device. However, *data format definitions* are only available within the DBMS or sometimes within the application on top. Moreover, data layouts (page or record) and traversal methods for the data organization (files or LSM-trees) are typically hard coded in the DBMS and thus not available on device.

To address the above, in this paper, we present nKV, which is a key/value store utilizing *native computational storage* and *near-data processing* (Figure 1). nKV eliminates intermediary layers along the I/O stack (e.g. file system) and operates directly on NVM/Flash storage. nKV directly controls the physical data placement on chips and channels, which is critical for utilizing the on-device I/O properties and compute parallelism. Furthermore, nKV can execute various operations such as *GET* or *SCAN* or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM-cores and as hardware-software NDP (HW/SW-NDP) using corresponding FPGA-based accelerators. The necessary FPGA hardware is built in the form of simple processing elements that can be used to offload certain tasks from the ARM-cores. Under nKV we target *host-intervention-free* NDP-executions, i.e. the NDP-device has the complete address information, can interpret the *data format* and access the data in-situ without host interaction. To reduce data transfers nKV also employs novel *ResultSet*-transfer modes. nKV is resource efficient as it eliminates compatibility layers and utilizes freed compute resources for NDP. nKV performs 1.4×-2× better than RocksDB: *GET latency* – 1.4×; *SCAN* – 2×; *BC* execution time – 2.7×.

This paper is organized as follows. In the next section we describe the data organization of RocksDB and the challenges it poses to NDP. In Section 3 we describe the architecture of nKV and how those NDP-challenges are addressed in terms of interface extensions (Section 3.1), in-situ data processing (Section 3.2), as well as operations and algorithms (Section 3.3). The architecture of the underlying NDP hardware accelerators is described in Section 5. We discuss the experimental results in Section 6 and conclude in Section 8.

2 BACKGROUND

In contrast to traditional data organizations, where data is updated in-place, LSM-trees [21] have been proposed as an out-of-place update structure to tackle the sustained update and insertion rates of modern workloads and provide query capabilities at the same time. Classical LSM-trees [21] comprise multiple B-Tree-structured index components (C_0 to C_K, Fig. 2) that are stored in new locations and have constant size ratios $r = |C_{i+1}|/|C_i|, i \in [0, K)$. An insert or update operation hits the C_0 component that is located in memory. Once it reaches a size threshold, it is flushed to disk and is merged with the C_1 component. The merge processes gradually move data from C_0 to C_K, purge outdated KV-Pairs, reclaim space and indirectly ensure hot-cold data separation.

nKV builds on RocksDB [9], which introduces one independent LSM-Tree per column family to separate the access characteristics of different database objects. Modern LSM-Tree variants (surveyed in [19]) are multi-levelled. Modifications to an LSM-Tree are first placed in the main memory component C_0, which comprises a set of *MemTables* in RocksDB. These are realized as memory-efficient data structures such as SkipLists. Whenever a MemTable reaches a given size limit, it becomes *immutable* and a new one is created to accommodate further modifications. Later on, immutable MemTables are transformed into *Sorted String Tables (SST)* and flushed to the secondary storage (Fig. 2), whereas each LSM-tree component $C_1..C_K$ comprises multiple SSTs. Thereby, the contained key-value pairs are placed into multiple *data blocks* in sort order of the key. Furthermore, an *index block* that comprises key-offset pairs pointing to each data block (a sparse index) is prepended. Index blocks reduce the access complexity to key-value pairs within the SST.

During the flush to C_1 no merge occurs for performance reasons. Consequently, overlapping key-value ranges of SSTs can occur (consider SST_{12}-SST_{1n}, C_1, Fig.2). Merge steps to underlying layers $C_2 \dots C_K$, called compactions, take either SSTs only on the level above or combine them with SSTs on the target layer, based on the given strategy (e.g. tiered or levelled). Either way, all KV pairs of the input SSTs are sorted, out-dated entries are pruned, and the results are stored in new SSTs on the target level (see dotted box, Fig. 2). Hence, key ranges in SSTs bellow C_1 do not overlap anymore. Yet, keys may appear on multiple levels with different values (consider $Key11$ or $Key70$), *to account for the temporal distribution of updates to a given key-value record.* For instance $Key70$ has been updated multiple times: $Key70$ on C_1 is the most recent record and its existence invalidates $Key70$ on C_2 and C_3.

To *retrieve* a key-value record based on the key, the $GET(key)$ first traverses the MemTables and the immutable MemTables on C_0. If the respective key is not found, the index block of one or more SSTs in C_1 has to be read (as SSTs may overlap on C_1, but not on $C_2 \ldots C_K$). By parsing the key-offset pair, the data block, which might contain the key, can be identified and also has to be read from secondary storage. If the key is still not found, layers ($C_2 \ldots C_K$) have to be traversed similarly. Due to the data organization and the compaction process, a key can now reside only in a single SST per level. *Range scans* with or without key predicates behave similarly, but are more complex and are supported by other internal structures (like fence pointers). Consider $SCAN([Key68, Key70])$, which traverses all levels and retrieves $Key70$ from C_1, and $Key69$ and $Key68$ from C_3.

However, if a scan involves *value predicates*, e.g. $SCAN(0 \leq Val \leq 7)$, the only option is to iterate over the entire dataset, yielding a significant increase of I/O transfers, which in turn has enormous potential to be improved via NDP.

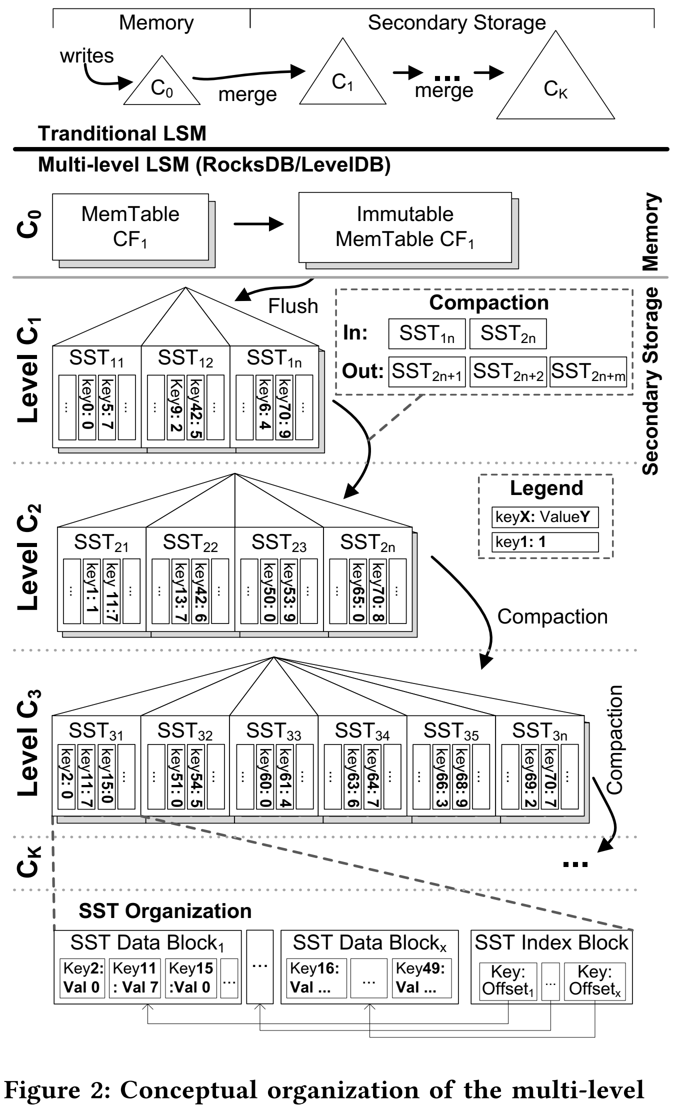

Figure 2: Conceptual organization of the multi-level LSM-Trees in RocksDB/LevelDB.

3 ARCHITECTURE OF nKV

Native computational storage. One of the underlying design principles behind nKV is that native storage enables efficient NDP (Figures 1 and 3). In this sense nKV extends [29]. *Native storage* is storage that is operated without intermediary/compatibility layers of abstraction along the critical I/O path, and is directly controlled by the database. This means that nKV can directly operate on NVM/Flash storage using physical addresses and thus can precisely control physical placement of SST data. It is this physical placement that allows utilizing the on-device I/O bandwidth and the FPGA's compute parallelism.

nKV physically places *SST data blocks* and *SST index blocks* on different *LUNs* and *Channels* (see Figures 2 and4). This allows for reaching the internal bandwidth (Table 2) by requesting index and data blocks asynchronously and utilizing processing parallelism of FPGA-based processing elements (PEs). Besides, individual levels of the LSM-Tree are physically separated on different chips and LUNs to improve I/O throughput and parallelism since I/O-heavy compaction jobs do not block the entire device, reducing demand pressure on the bus.

Furthermore, nKV operates directly with physical addresses, to access (read or write) precisely the physical pages that are needed. This, in turn, is essential for reducing read- and write-amplification. Moreover, it inherently avoids costly host round-trips for logical-to-physical address translation. Native storage eliminates these *information hiding* effects incurred through layers of abstraction and thus simplifies

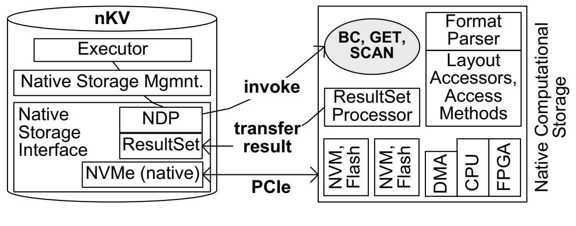

Figure 3: Architecture of nKV

the NDP-operations. Hence, native storage leads to leaner NDP-functionality.

Computation Placement. By using native computational storage, nKV can directly place computations on the heterogeneous on-device compute elements, such as ARM-cores or FPGA-based processing elements. nKV can execute various operations such *GET*, *SCAN* or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM-cores or with hardware support from the FPGA (cf. Section 5). The experimental evaluation indicates that some NDP operations such as *NDP_GET(key)* perform best on the ARM-cores, while other operations like *NDP_SCAN(value_condition)*, benefiting from parallelism, perform best on the FPGA. For its NDP-operations nKV utilizes *hardware/software co-design* to handle the proper separation of concerns and achieve best performance.

3.1 NDP Interface Extensions

NVMe support. nKV has a dedicated high-performance *user-space* and *in-DBMS* NVMe layer (Figure 3). It is very lean and tightly integrated with the rest of nKV. The *native NVMe* integration can control multiple NVMe submission and completion queues either through dedicated threads or through the transactional context. Moreover, it reduces the I/O overhead as it allows the seamless creation of I/O and NDP tasks, the precise allocation of transfer buffers for the DMA engine, and prioritizable placement within the NVMe submission queues. The deep database integration additionally avoids expensive synchronization between user- and kernel-space, and shortens the I/O paths even further as no drivers are involved along the critical access paths. Internally, the native storage command set is translated to specific NVMe I/O and NDP tasks. Although these resemble the standardized NVMe commands, they define a new category - n over NVMe. In nKV, they can be scheduled either for *synchronous* or for *asynchronous* execution.

Command set. Besides the classical native storage interface, nKV introduces NDP-Extensions [29] in terms of a generic *NDP_EXEC()* command. It takes the following parameters, among others:

(i) *Command Identifier* – Unique identifier of the NDP function;
(ii) the *SearchKey* or *SearchKeyRange(s)* – for GET or SCAN;
(iii) the *ResultsSet Size*;
(iv) *AddressMappings* – these are ranges of physical addresses, where the physical data to be processed is located;
(v) *Min/Max Keys* – RocksDB supports a type of zone map range filter that can be used on device to skip processing some index/data blocks;
(vi) *Miscellaneous* – command specific parameters such as data format definitions.

nKV composes the NVMe command based on the given parameters, current state and address information, and its transactional context. After placing it in the NVMe submission queue and DMA transferring the parameters to the device, the NDP command is then executed. The result set is handled by the ResultSet processor, which also observes the execution status. Finally the ResultSet is transferred to nKV by the DMA engine. An NDP-operation can invoke multiple low-level NDP commands synchronously or asynchronously.

3.2 In-situ Data Access and Interpretation

Under nKV, the NDP-device can interpret the *data format* and *access* the data without *host intervention* (synchronization with the host) [28]. To this end, nKV extracts definitions of the *Key- and Value-formats*, and passes them as input parameters to the NDP-commands. Moreover, the *data format* such as the *Key- and Value-formats* can be automatically extracted from the DB-catalogue (*system-defined*) or can be defined by the *application*.

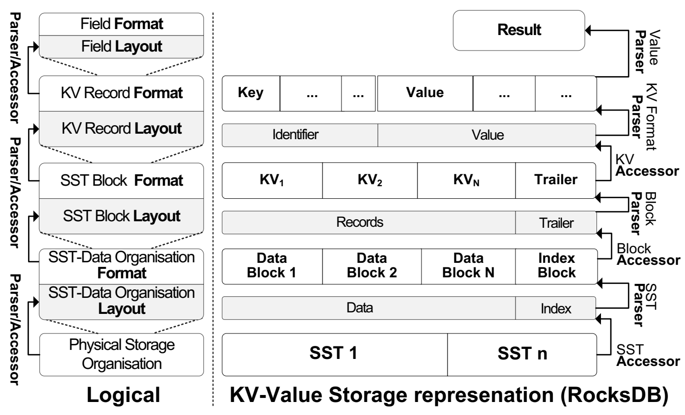

Figure 4: In-situ access and data interpretation in nKV, based on layout accessors and format parsers.

nKV employs a thin on-device *NDP-infrastructure* layer that supports the execution and simplifies the development of NDP-operations (Figures 3). It comprises *data format parsers* and *accessors* that are implemented in both *software* and *hardware* (Figure 4). The in-situ *accessors* traverse and extract the contained sub-entities of the persistent data. Whereas, the in-situ *data format parsers* process the *layout* of each persistent entity, and extract the sub-entities by invoking further accessors (Figure 4).

KV-Stores like LevelDB or RocksDB [9] organize the persistent LSM-Tree data into so called *String Sorted Tables (SST)* – see Sect. 2. To process a *GET(key)* request, for instance, nKV first identifies the respective *SST* and invokes an *NDP_GET* command with the corresponding physical address ranges, the respective *Key- and Value-formats* as well as further parameters. First, the *SST layout accessor* is invoked to access the data and the index blocks. Subsequently, the index block parser is activated to interpret the data and verify if the *key* is present and extract its offset. If this is the case, data block accessor and parser are invoked to extract the *Key/Value entry*. In case of an *NDP_SCAN(key_val_condition)* operation, the KV accessor is subsequently invoked to extract it, followed by a *field* parser to extract its value and verify the *condition*. This way, nKV extends scans in classical KV-Stores. Typically, they are only able to process filter criteria on key-embedded attributes, but not filter predicates involving the value.

3.3 Operations and Algorithms

Lookup. KV-Stores offer fast (low-latency, high parallelism) retrieval of a value, based on its key, through the *GET(key)* operation. In nKV, this operation first performs a lookup within main-memory components (MemTables, Fig. 2) regardless of execution model. If the search key is not found, the lookup will proceed scanning the deeper LSM-Tree levels by processing their indices first and eventually their associated data blocks. Both, index and data block scanning are I/Os intensive in the traditional stack (Figure 1), while with NDP, these can be performed efficiently on-device.

Scan. As mentioned in Section 3.2, Scans can be performed either on Key- or Value-embedded attributes. The index blocks of the LSM-Tree might be leveraged to navigate to necessary data blocks for key-attribute scans, similar to the Lookup operation. However, there is no auxiliary structure to accelerate scans on value-embedded attributes. Either way, multiple data blocks have to be examined depending on the selectivity of the filter-predicate. Consequently, Scans usually result in a high amount of data transfers, which NDP can significantly reduce.

UDF: Betweenness Centrality. Many applications involve more complex algorithms. Such user-defined functions (i.e. Betweenness Centrality) can also be supported by nKV. The specific algorithm implemented within nKV relies on [6] and measures the degree, to which nodes stand among each other. The logic involves shortest-path searches over the given nodes and therefore results in a variety of lookups and scans involving random and sequential I/O.

3.4 Data Consistency, Database Maintenance and NDP

In parallel to the execution of NDP functionality, nKV allows the processing of database maintenance e.g. compactions. Such parallel operations might result in new data or even changes to the LSM-Tree. Yet, as nKV's NDP-operations are executed on a snapshot of the physical data, concurrent modifications do not effect its consistency. This can be ensured by firstly, the underlying mechanism of Copy-on-Write (CoW), secondly the precise placement through the native storage interface, and thirdly, the direct control of the physical GC by nKV. Moreover, nKV requires no on-device bad-block re-mapping like other native storage management solutions [4], since bad-block management and wear-leveling are handled directly within the database engine [23]. Thus native storage management [23] leverages the the above issues by DBMS managed physical-to-logical address mapping and data placement.

3.5 Result Set Handling

Unnecessary data transfers may occur depending on how the result of an NDP-operation is transferred back. Therefore, ResultSet management additionally helps to avoid unnecessary stalls of processing resources. nKV aims to reduce the data transfer overhead caused by a Volcano-style *record-at-a-time* model. Instead it aims to bulk-transfer the *ResultSet*. The former is simpler, but leads to more frequent shorter burst transfers causing bus overhead. The latter results in fewer, but longer bursts leveraging the throughput-optimized PCIe.

Each NDP call in nKV defines a maximum ResultSet size as a parameter. The NDP ResultSet-Processor (Figure 3) allocates on-device resources for it: either in DRAM, or if the contention is high, it allocates a temporary region on Flash. As long as the actual result size does not exceed the predefined one, the ResultSet is sent back as bulk DMA-transfer, to leverage the full performance of the DMA engine. Alternatively, it may be pipelined to the next NDP-operation. Furthermore, nKV has a built-in extension mechanism that in the worst case may preemptively request more physical space from the DBMS, as it manages the logical-to-physical address mapping [23].

4 HARDWARE-ARCHITECTURE

The *COSMOS+* platform [22] is a PCIe-based extension-card. It contains all the required hardware-modules to function as a regular NVMe-based SSD. It can be fitted with up to 2 DIMM-extensions containing Flash modules. The available Toggle-NAND Flash-extensions can be configured in SLC or MLC mode. In this work, they are configured as SLC with 16 dies organized in two channels.

The main computational engine of the *COSMOS+* platform is a Xilinx Zynq-7000 SoC (XC7Z045-3FFG900) that combines two 667 MHz ARM Cortex-A9 cores with an FPGA (Figure 5). In the The *COSMOS+* platform [22], the FPGA-portion is used to implement the required SSD-infrastructure. This infrastructure is made up of two separate domains: The first one is responsible for accessing the flash memory. It comprises one or many Tiger4-controllers with corresponding low-level flash interfaces. For each channel, a distinct Tiger4-controller, as well as a low-level interface, is required.

The second domain contains primarily an NVMe-Core, which allows access from the host to the device via the NVMe interface. The NVMe-Core also wraps the actual low-level PCIe-interface.

Both of these domains are running at different clock-frequencies. While the flash domain uses a 100 MHz clock, the NVMe-Core is running at 250 MHz. When planning to extend this platform, the following aspects are relevant:

1) The amount of resources on the FPGA-portion (PL) of the SoC is limited. While the platform can be fitted with more flash-DIMMs, this also requires more flash controllers. This in turn impacts the resource requirements. In this work, one flash-DIMM is used with 2 flash controllers. While this limits the available flash memory and the corresponding parallelism, it also frees up resources for different use (i.e. computational processing elements).

2) Since the different domains are running at different clock-frequencies, the extensions should be able to run at the same clock-frequencies. In the case of the *COSMOS+* platform, this is not a huge problem, since most hardware-accelerators can run at 100 MHz and can therefore reside in the flash-domain.

A simplified view of the architecture is depicted in Figure 5. It also includes the nKV hardware *extensions* described in this work (Section 5).

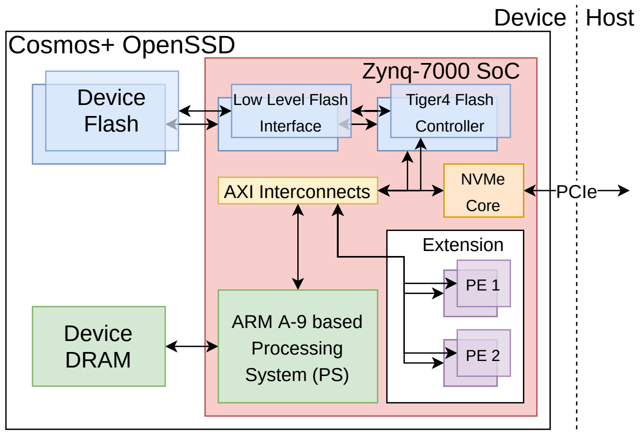

Figure 5: A simplified view of the architecture of the *COSMOS+* OpenSSD [22], including the proposed extension.

5 HARDWARE-ACCELERATION

Using the baseline architecture (Figure 5), specific processing elements (PEs) are implemented, allowing to move computation from software running on the ARM-cores to the programmable logic of the SoC, potentially improving latency, throughput, and available parallelism. The PEs are written in Chisel3 [3] using a relatively simple architecture that can be subdivided into four distinct domains (cf. Figure 6):

The control-domain consists of a register file, holding a number of control registers, which are accessible using an AXI4-Lite interface. The corresponding addresses are mapped into the address space of the processing system (PS), so that the ARM-cores can read and write these registers and thereby control and configure the PE. The control registers hold the required parameters for the functionality provided by the processing elements (e.g. the memory addresses of the input and output). In addition, the signaling to the ARM-cores is also done using these registers. One register indicates whether the PE is busy, while another can be used trigger the execution.

The memory-domain contains a load and a store unit. These are connected to the PSs DRAM-interface, allowing the PE to access the device DRAM. Both the load and the store unit perform data transfers in chunks of 32 KB, which corresponds to the size of a single data-block in our RocksDB-configuration. The transfers are performed using AXI4 bursts to maximize throughput. The data-width of the AXI4-Bus is 64 bits and the AXI burst length is 16. Generally, longer bursts allow higher throughput, due to the sequential access pattern. Unfortunately, the Zynq-7000 family only supports a maximum burst length of 16.

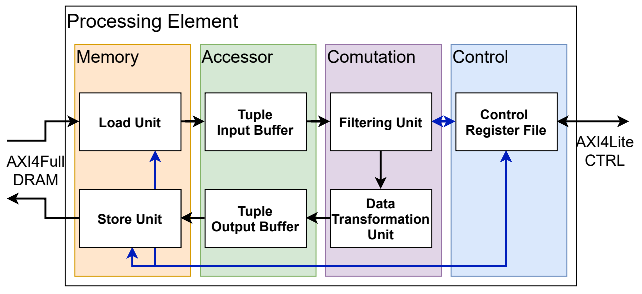

Figure 6: The overall Microarchitecture of the proposed Parser Processing Elements.

The accessor-domain is responsible for converting the different data granularities (64 bit words vs tuples) between the memory and the computational domain. The tuple input buffer will buffer incoming 64 bit data words from the load unit, until a complete tuple is available. This will then be passed to the filtering unit. Similarly, the tuple output buffer will receive a resulting tuple and split it into words of 64 bits to allow transfer to memory via the store unit. In this context, a tuple corresponds to a key-value pair (kv-pair).

The computational domain is comprised of two distinct modules. The first one is the filtering unit, which accepts single kv-pairs as input. Depending on the control registers, the filtering unit will then pass on matching kv-pairs to the data transformation, while non-matching ones are discarded. In the current implementation, the filtering unit can be configured to apply a single predicate on a kv-pair. This is done using three parameters: the column selector ($i \in [0, n-1]$, where n is the number of distinct data-fields), the compare operator ($op \in \{nop, =, \neq, >, \geq, <, \leq\}$) and a reference value (c) to compare against. Considering a kv-pair $t = (t_0, t_1, t_2, \ldots, t_{n-1})$, the following expression is evaluated: $r = t_i \ op \ c$. If r is true, the kv-pair will be passed on, else it will be discarded.

The last module transforms the data into the required output format. This corresponds to a projection of tuples and allows the automatic removal of RocksDB-metadata or unnecessary tuple elements. The transformed tuple is passed back to the accessor-domain, to be stored back to the device DRAM. The complete microarchitecture of the PEs is also depicted in Figure 6.

Building atop of the baseline architecture of COSMOS+ (Figure 5), we developed an extended architecture, which contains additional processing elements. In particular, we built two different processing elements for the specfic evaluation dataset (the database-of-research-papers): One for the data of the paper themselves (paper-PE), and another one for the data of the references (ref-PE). Initial experiments showed, that the paper-PE can process a 32 KB block of data faster than the two Tiger4s controllers are able to provide it (due to the flash latency). Thus, we employ a single paper-PE in the final architecture. For the handling of the paper references in the database, much more data has to be processed multiple times. In this case, the flash latency becomes less relevant, since the reference data is cached in the on-device DRAM and does not have to be fetched from flash memory each time. Thus, the architecture can keep multiple ref-PEs busy. To keep the interconnects balanced, we opted for seven ref-PEs, yielding a total of eight PEs (including the single paper-PE). Generally, it would be possible to increase the number of PEs, but all active PEs compete for access to the on-device DRAM. Thus, there will be a point of diminishing returns considering overall throughput as soon as the full memory bandwidth is saturated. Instead of replicating PEs for more throughput, it might be more reasonable to use multiple different PEs to increase flexibility of the hardware acceleration.

The baseline and extended hardware designs were synthesized and implemented using Xilinx Vivado v2019.1. The resulting resource utilizations are reported in Table 1, both in terms of absolute numbers, as well as relative to the resources available on the Zynq 7045 chip. The baseline results indicate that the Zynq has many spare resources. Even though small, a large fraction of its hardware resources are unused. The main reason for this lies in the flash-configuration. For a platform with two flash-DIMMs and the full parallelism, eight flash controllers and flash interfaces are needed. In our design, we only use one flash-DIMM with two controllers/interfaces, which vastly reduces the resource-requirements.

Table 1: FPGA-Resource Utilization of the Baseline and extended Architectures, including hierarchical utilizations of relevant sub-modules.

	Slices		BRAM		DSPs	
	abs.	%	abs.	%	abs	%
Baseline	14544	26.61	78	14.31	0	0
Tiger4	8174	5.81	15	2.75	0	0
NVMe-Core	4312	7.89	29	5.32	0	0
LL Flash	475	0.87	5	0.92	0	0
Extended	35667	65.26	101	18.53	0	0
paper-PE	33103	15.14	0	0.0	0	0
ref-PE	4012	1.84	0	0.0	0	0
Available	54650	100.00	545	100.00	900	100

These free resources are then leveraged by our extended architecture to offload computations from the ARM-core to the FPGA. In doing so the hardware accessors and format parsers can be instantiated multiple times. In fact, nKV uses two different kinds of parsers with up to seven instances.

Another interesting point is the vast difference in resource requirements between the paper-PE and the ref-PE. The reason for this lies in the different sizes of the parsed kv-pairs. The kv-pairs processed by the paper-PE are 136 bytes each, while the kv-pairs processed by the ref-PE are merely 36 bytes each. Apart from the data-size, the number of distinct data fields also plays an important role, due to the number of required comparators.

Finally, there is a complete lack of DSP utilization. DSPs are hard-wired special-function slices which provide fast arithmetic and logical operations, that are typically relevant in the context of digital signal processing. For our work, DSPs could be interesting for arithmetic comparisons as well as pattern recognition.

For future extensions of this work, the above could be exploited to reduce Slice-utilization, or to implement more complex functionality within the PEs.

6 EVALUATION

For the evaluation, the *COSMOS+* board [22] (see Figure 5) is attached over PCIe 2.0 x8 as an NVMe block device supported by Greedy FTL to realize traditional storage. The host server is equipped with a 3.4 GHz Intel i5, 4GB RAM and

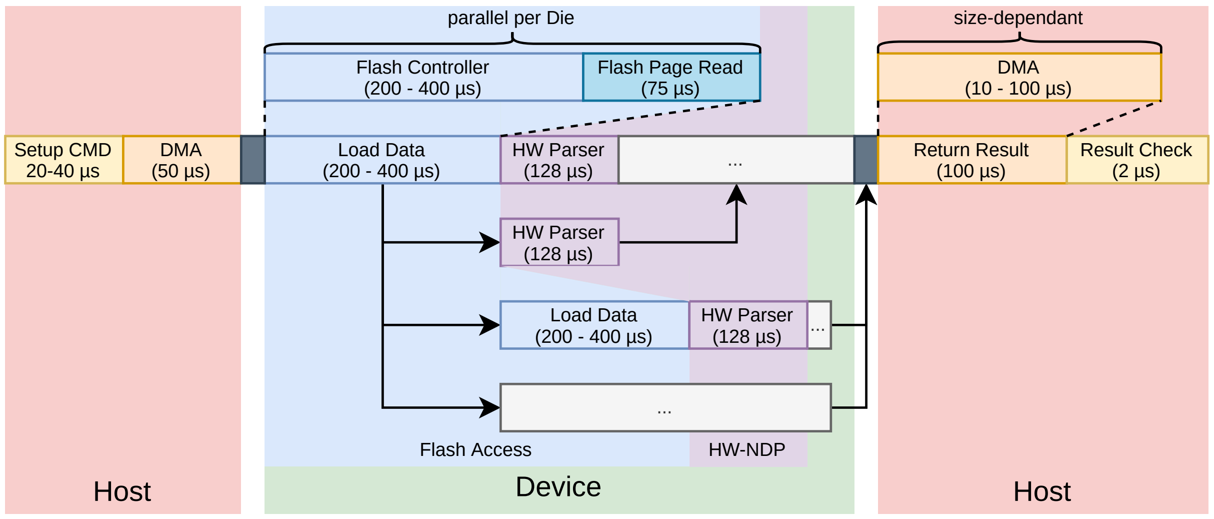

Figure 7: Break-Down of Execution Times within the NDP Stack with HW support.

runs Debian 4.9 with ext3. We configure both RocksDB [9] and COSMOS+ [22] with a 5MB cache. COSMOS+ is directly mapped into the userspace and controlled by *native NVMe*.

We evaluate nKV on a *2.4GB* research paper graph dataset from Microsoft Academic Graph [27]. It comprises approx. *48 million* Key/Value-pairs: *3.8M* papers, *40M* references, *18K* venues, and *4.2M* authors. For each experiment, we report the average execution times of three cold test runs. The baseline for our experiments is RocksDB using block-device storage (*Blk*) on top of *GreedyFTL* and *ext3*. Performance results of GET(key), SCAN(value_predicate) and BC are reported for three different stacks: *Blk* as baseline, software NDP (*NDP:SW*) on the ARM and FPGA-assisted NDP (*NDP:SW+HW*).

6.1 Low-level Flash Properties

Physical data placement and the on-device Flash characteristics play an essential role in nKV. The following Table 2 shows the on-device latency and bandwidth, measured by directly issuing page reads to the Flash Management Unit. The level of parallelism is controlled by data placement on either different Channels, LUNs or both. While a single page-read takes approx. 300μs, careful data placement on Channels and LUNs reduces latency down to 94μs with full parallelism (Table 2). However, an upper limit of around 217 MB/s can be observed for sequential and random workloads, which is due to the bus limitations of COSMOS+.

6.2 Experiment 1: Lean Native Stack

One important conceptional characteristic of NDP with nKV is the removal of traditional compatibility layers to simplify the access stack. To verify this property, we execute a

Table 2: Flash Latencies and Bandwidth (BW) of the *COSMOS+ OpenSSD* for different levels of parallelism.

	Pages	Parallelism	Duration per Page [μs]	
	1	1 Ch. 1 LUN	300.00	
	4	2 Ch. 2 LUN	113.50	
	8	2 Ch. 4 LUN	*94.12*	
Access	**Pages**	**Parallelism**	**BW [MB/s]**	**IOPS**
Random	1500	1 Ch. 1 LUN	52	3000
	1500	2 Ch. 1 LUN	102	6000
	1500	1 Ch. 8 LUN	108	6000
	1500	*2 Ch. 8 LUN*	*213*	*13000*
Seq.	*640*	*2 Ch. 8 LUN*	*217*	*13000*

GET(key) command. We compare the results of our baseline (Blk) against nKV with software NDP (NDP:SW), and nKV with Parsers-PE support (NDP:SW+HW) – Figure 8.

nKV utilizes the native data placement and **improves the round-trip time by 1.4**× due to *native NVMe* and mapping the device into its userspace. This reduces the execution time from 7.9 ms to 5.7 ms, as shown in Figure 8. Interestingly, there is no benefit from hardware PEs since the gains from concurrent executions are limited due to the sequential nature of first reading and then processing Flash data.

6.3 Experiment 2: Data Transfer Reduction

While the relatively simple GET-operation does not benefit from the hardware PEs, this changes for bigger and more complex operations like the SCAN. In addition to the vastly reduced latency, the use of NDP has additional effect on the overall system. While all three implementations read similar amounts of data from flash (492.3 MB ± 0.2 MB due to

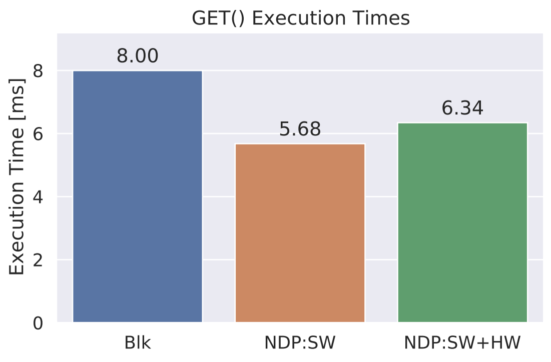

Figure 8: GET execution times for Blk, NDP:SW and NDP:SW+HW.

caching), the required DMA data transfers vary. For NDP-operations, a single DMA transfer is required to push down the additional parameters. We draw the following conclusions. Firstly, efficient ResultSet handling reduces the transfer overhead by employing large bulk DMA transfers. Secondly, due to on-device filtering the amount of data to be transferred also decreases depending on the predicate selectivity.

The execution time is reduced by **more than 2x** (from 6.95 s to 3.35 s) as shown in Figure 9.

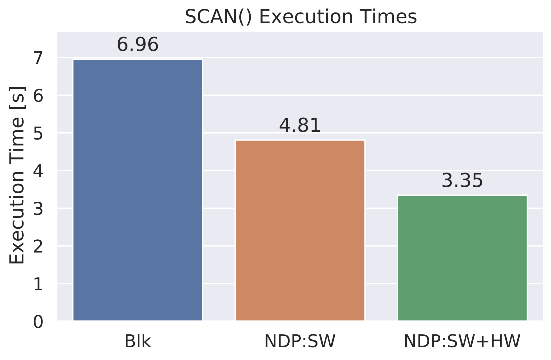

Figure 9: SCAN execution times for Blk, NDP:SW and NDP:SW+HW.

6.4 Experiment 3: Native Computational Storage

Native Computational Storage plays an essential role for nKV. Especially with complex graph analysis operations like Betweenness Centrality (BC) the concepts of native data placement, flash parallelism and computation placement can be leveraged to improve the performance. An execution on a smaller graph, benefits the software implementation. For a large number of edges, the complexity is high and multiple HW Parsers can be utilized to improve performance. With a total of 7 HW Parsers nKV achieves **2.7x speed-up** for 2.037.755 edges (Figure 10).

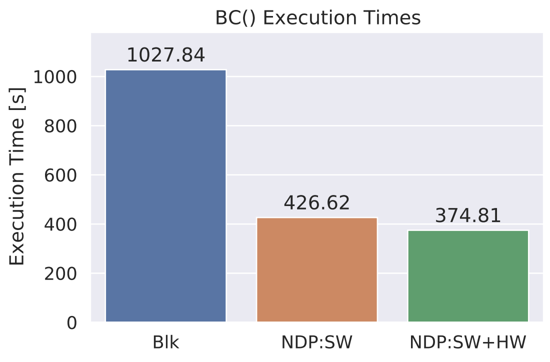

Figure 10: Betweenness centrality (BC) execution times for Blk, NDP:SW and NDP:SW+HW.

6.5 Experiment 4: Execution Parallelism

In large graph processing the number of applied HW Parsers is important to balance between FPGA utilization, memory bandwidth limitations and performance. nKV allows configure the number of HW Parsers individually for each NDP operation. In Figure 11 BC is executed with 2.037.755 edges using a different number of ref-PEs. Clearly, increasing the

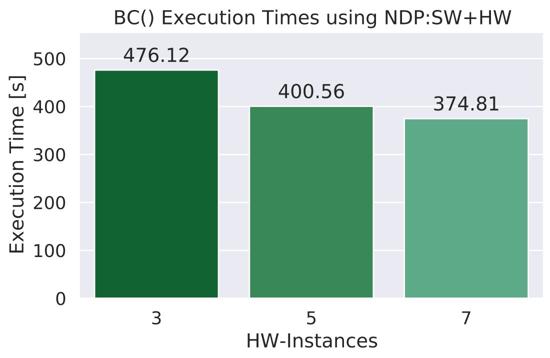

Figure 11: Betweenness centrality (BC) execution times for for NDP:SW+HW using 3, 5 and 7 instances of the ref-PE hardware parser.

number of PEs per operation, yields better speed-ups. Using seven PEs instead of three PEs results in a **speed-up of 1.25x**. While the data suggests that more parsers are better, it is important to note that all instances compete for DRAM accesses. Thus, adding more parsers will yield diminishing returns due to memory contention and increased randomness in the memory access patterns.

7 RELATED WORK

The Near-Data Processing is deeply rooted in *database machines* [5] developed in the 1970s-80s or Active Disk/IDISK [1, 16, 25] from the late 1990s. Besides dependence on proprietary and costly hardware, the I/O bandwidth and parallelism are claimed to be the limiting factors. While not surprising, given the characteristics of magnetic/mechanical storage combined with Amdahl's balanced systems law [10], this conclusion needs to be revised. Storage devices built with modern semi-conductor storage technologies (NVM, Flash) are offering high raw bandwidths with high levels of parallelism *on-device*.

With the advent of Flash technologies and reconfigurable processing elements, Smart SSDs [8, 15, 26] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [30]. Biscuit [11] is a timely proposal for a general NDP framework. JAFAR [2, 31] is one of the first systems to target NDP for DBMS (column-store) use, whereas [14, 18] target joins besides scans. The use of NDP in the realm of KV-Stores has been investigated in [7, 17]. Kanzi [12], Caribou [13] and BlueDBM [20] are RDMA-based distributed KV-Stores investigating node-local operations.

Much of the prior work on persistent KV-Stores and NDP focusses on *bandwidth* optimizations. NoFTL-KV [29] addresses the problem of *write-amplification*. The NDP extensions demonstrated by $\underline{n}$KV target the *read-amplification*, *latency improvements* and *computational storage*.

8 CONCLUSION

In this paper we introduced $\underline{n}$KV – a key-value store designed for native computational storage and near-data processing. $\underline{n}$KV controls physical data placement directly and hence the on-device I/O parallelism. Along the same lines, $\underline{n}$KV can place NDP operations on different compute elements on device (ARM or novel FPGA PEs) and also configure the hardware per operation accordingly, e.g. the number of hardware parsers used. Both placement methods impact the performance of NDP operations, GET is faster on the ARM, while SCANs are faster with hardware support. $\underline{n}$KV is based on the principle of explicit cross-layer data formats, hence hardware or software layout accessors and format parsers are deployed an can be used for different operations.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for the valuable comments, which significantly improved the quality of the paper. This work has been partially supported by *BMBF PANDAS – 01IS18081C/D*; *DFG Grant neoDBMS – 419942270*; *HAW Promotion*, *MWK, Baden-Würrtemberg, Germany.*

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active Disks: Programming Model, Algorithms and Evaluation. In *Proc. ASPLOS 1998*. 11.

[2] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. 2015. JAFAR : Near-Data Processing for Databases. *In Proc. SIGMOD 2015.*

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proc. DAC 2012.*

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

[5] Haran Boral and David J. DeWitt. 1989. Parallel Architectures for Database Systems. Chapter Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, 11–28.

[6] Ulrik Brandes. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* (2001).

[7] Arup De, Maya Gokhale, and et. al. 2013. Minerva: Accelerating Data Analysis in Next-Generation SSDs. In *Proc. FCCM 2013.*

[8] Jaeyoung Do, J. Patel, D. DeWitt, and et. al. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proc. SIGMOD 2013.*

[9] Facebook. 2020. RocksDB. https://github.com/facebook/rocksdb.

[10] Jim Gray and Prashant J. Shenoy. 2000. Rules of Thumb in Data Engineering. In *Proc. ICDE 2000.*

[11] Boncheol Gu, Andre S. Yoon, and et al. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proc. ISCA 2016.*

[12] Masoud Hemmatpour, Mohammad Sadoghi, and et al. 2016. Kanzi: A Distributed, In-memory Key-Value Store. In *Proc. Middleware 2016.*

[13] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. In *Proc. VLDB 2017.*

[14] Insoon Jo and et al. 2016. YourSQL : A High-Performance Database System Leveraging In-Storage Computing. In *Proc. VLDB 2016.*

[15] Yangwook Kang, Yang-suk Kee, and et al. 2013. Enabling cost-effective data processing with smart SSD. In *Proc MSST 2013.*

[16] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.* (1998).

[17] Jungwon Kim and et al. 2017. PapyrusKV: A High-performance Parallel Key-value Store for Distributed NVM Architectures. In *Proc. SC 2017.*

[18] Sungchan Kim, Sang-Won Lee, Bongki Moon, and et al. 2016. In-storage Processing of Database Scans and Joins. *Inf. Sci.* (2016).

[19] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[20] Sang-woo Jun Ming, Arvind, and et al. 2015. BlueDBM: An Appliance for Big Data Analytics. *Proc. ISCA* (2015).

[21] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* (1996).

[22] OpenSSD Project 2019. *COSMOS Project Documentation.* OpenSSD Project. http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Technical_Resources.

[23] Ilia Petrov, Andreas Koch, Sergey Hardock, Tobias Vincon, and Christian Riegger. 2019. Native Storage Techniques for Data Management. *Proc. ICDE* (2019).

[24] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. 2001. Active disks for large-scale data processing. *Computer (Long. Beach. Calif).* 34, 6 (2001), 68–74.

[25] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia. In *Proc. VLDB 1998.*

[26] Sudharsan Seshadri, Steven Swanson, and et al. 2014. Willow: A User-Programmable SSD. *USENIX, OSDI* (2014).

[27] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *Proc. WWW 2015*.

[28] Tobias Vincon, Arthur Bernhardt, Lukas Weber, Andreas Koch, and Ilia Petrov. 2020. On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems. In *Proc. HardBD @ ICDE 2020.*

[29] T. Vincon, S. Hardock, C Riegger, J. Oppermann, A. Koch, and I. Petrov. 2018. NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management. In *Proc. EDBT 2018*.

[30] Louis Woods, J. Teubner, and G. Alonso. 2013. Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances. In *Proc. SIGMOD 2013*.

[31] Sam Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. 2015. Beyond the Wall: Near-Data Processing for Databases. *Proc. DAMON* (2015).