Cache-Coherent Shared Locking for Transactionally Consistent Updates in Near-Data Processing DBMS on Smart Storage

Arthur Bernhardt*, Sajjad Tamimi#, Florian Stock#, Tobias Vinçon*, Andreas Koch#, Ilia Petrov*
#Embedded Systems and Applications Group, *Data Management Lab
#Technische Universität Darmstadt, *Reutlingen University

ABSTRACT

Even though near-data processing (NDP) can provably reduce data transfers and increase performance, current NDP is solely utilized in read-only settings. Slow or tedious to implement synchronization and invalidation mechanisms between host and smart storage make NDP support for data-intensive update operations difficult. In this paper, we introduce a low-latency cache-coherent shared lock table for update NDP settings in disaggregated memory environments. It utilizes the novel CCIX interconnect technology and is integrated in neoDBMS, a near-data processing DBMS for smart storage. Our evaluation indicates end-to-end lock latencies of ~80-100ns and robust performance under contention.

1 INTRODUCTION

Data-modifying operations on large datasets can impair performance, as they may request data residing on cold storage, causing significant data movement [13]. This poor data locality results in massive data transfers that are necessary to verify which data meets the update conditions. To avoid this, Near-Data Processing (NDP) can leverage the higher device-internal bandwidth, parallelism and especially faster storage latencies compared to host-to-device.

Problem 1: Read-only NDP. Even though NDP can provably reduce data transfers and increase performance [9], currently NDP is utilized solely in read-only settings [2, 4, 9, 14]. Yet, in update-intensive settings, e.g., under HTAP or OLTP workloads, *transactional consistency* becomes an issue. On the one hand, the most recent updates of OLTP-style transactions are only available in the large DBMS memory [5], likely scattered across different data structures. On the other hand, NDP operations, offloaded to smart storage, require the most recent data *in-situ*, alongside the cold persistent dataset. To this end, we recently proposed nKV[15–17] and neoDBMS[3, 15] that define a small shared-state that collects all modifications to main-memory data and DBMS state. The shared-state is regularly flushed to smart storage, whenever it reaches a pre-defined limit, but most importantly, it is propagated as part of every NDP invocation. Thus, at the point of invocation, the smart storage attains a complete and consistent *snapshot*, and a read-only OLAP NDP operation can execute with transactional consistency guarantees. Moreover, the in-situ execution is asynchronous and free from DBMS/host intervention (*intervention-free NDP*).

Problem 2: Update NDP. Another major source of data movement are large data-intensive modification operations [13], which cannot be handled by *snapshot-based*, *intervention-free* NDP. To avoid update anomalies in *update-NDP* settings, low-latency, cache-coherent invalidation or synchronization mechanisms are

© 2022 Copyright held by the owner/author(s). Published in Proceedings of the 25th International Conference on Extending Database Technology (EDBT), 29th March-1st April, 2022, ISBN 978-3-89318-085-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

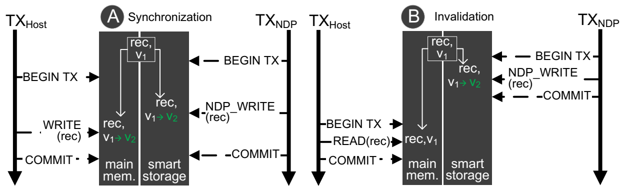

Figure 1: Concurrent TX_{host} and TX_{NDP} transactions anomalously create invalid version branches and commit.

necessary. The former *invalidates* the in-memory state (e.g., tuple versions, buffer, or address mapping entries) that an NDP operation has modified, effectively forcing the host DBMS to fetch the most recent data from storage to prevent the inconsistent use of the outdated in-memory data, in turn avoiding *write/read* conflicts (Fig. 1B). Given the latter, the smart device sends *ad hoc* lock-requests during NDP update processing, to avoid *write/write* conflicts (Fig. 1.A).

Introductory Scenario. Consider an NDP transaction (TX_{NDP}) offloading a modification operation to a record *rec* onto smart storage (Fig. 1). An initial version of *rec* is present both on device and in memory. The NDP modification of TX_{NDP} executes free of host-intervention, creating a new record version $rec.v_2$, while in the same time a host transaction TX_{HOST} does the same, creating its own $rec.v_2$, as the host DBMS is unaware of the on-device modification. As a result of the *write/write* conflict two version branches occur causing unresolvable inconsistencies. Clearly, such conflicts can be mitigated by exclusively locking the table containing *rec* ahead of TX_{NDP} execution, yet this is impractical as it would severely limit concurrency. Alternatively, Optimistic Concurrency Control (OCC) techniques are also possible, yet their *verification phase* mandates transferring the NDP read/write sets, causing additional data transfers, which likewise limits their applicability.

Shared lock-table. In this paper, we introduce an approach for a shared lock-table between smart storage devices and a host that relies on cache-coherent Shared Virtual Memory (ccSVM), which is enabled by new accelerator interfaces such as CCIX (used here) or CXL. Our solution is realized in neoDBMS, which is an NDP-DBMS, integrating ccSVM-capable smart storage. CCIX (Cache Coherent Interconnect for accelerators(X)) [10] is a novel coherent interconnect technology between general-purpose CPUs and accelerator devices, aiming at efficient heterogeneous computing. CCIX allows for low-latency ccSVM, atomics like atomic CAS or load/store, and address translation between host and (externally-attached) accelerators. Indeed, CCIX yields latencies of 80-100 ns at cacheline granularity, which are comparable to the NUMAlink latencies ≤500ns [11] in a large-scale main-memory system. To the best of our knowledge, this is the first paper to exploit CCIX for DBMS use and for NDP in disaggregated memory settings.

Our **contributions** are: (i) We present an approach for a CCIX-based cache-coherent shared lock table for synchronization in update NDP settings. Furthermore, we are investigating how well CCIX is suited for use in NDP; (ii) We compare CCIX and PCIe latencies; (iii) We show how neoDBMS utilizes CCIX to enable transactionally consistent updates on native smart storage.

The paper is organized as follows. We continue by providing background on SVM and ccSVM technologies as well as on the architecture of neoDBMS. In Sec. 3 we describe the design of our ccSVM lock table, with aspects such as logging and recovery being considered out of scope. The evaluation follows in Sec. 4.

2 BACKGROUND

2.1 Shared-Virtual Memory

Shared Virtual Memory (SVM) extends the concept of virtual memory from a single CPU to an entire group of devices. A regular CPU has access to the complete memory hierarchy and can manage every processes running on it. But in case of external hardware, such as accelerators, where either the CPU accesses that hardware, or where the hardware autonomously accesses the host memory, extra care must be taken.

From a developer standpoint, this has a number of drawbacks: working with memory addresses from the host system is cumbersome: translation is possible, but has to be done explicitly (e.g., multiple address translations when working with linked list data from the host). The same applies to cache coherence with the host system, which for PCIe has to be ensured manually. Also, SVM operations typically require just small amounts of data, especially compared to the large transfer sizes (256…512 KB) required for PCIe to reach maximum throughput. In addition, the relatively long PCIe latencies make it unsuitable for the short messages used in automated cache-coherence protocols.

2.2 Cache-Coherent SVM Interconnects

Cache-Coherent Interconnect for Accelerators (CCIX) is an advanced I/O interconnect that enables cache-coherent data sharing between various devices (e.g., CPUs and accelerators) mechanically connected via PCIe slots, but using different protocols [8]. CCIX supports signaling rates between 16-25 GT/s per link, while cache coherence is automatically maintained. CCIX supports multiple kinds of partners: Home Agents (HA) passively manage coherence and memory accesses to a specific address range; Request Agents (RA) actively initiate local and non-local read/write accesses to HAs and may perform local caching. E.g., for making host memory available to an accelerator that also has local memory, the host will be both an HA (for its own memory) and RA (for accessing accelerator memory), while the accelerator will just be an RA, assuming its own memory is not used in a cached manner. If both CCIX end points support Extended Speed Mode (ESM), they can agree on a higher link speed than PCIe (25 GT/s ESM vs. 16 GT/s PCIe 4.0).

Compute Express Link (CXL) is similar to CCIX, but was initially established by Intel. It is expected to become the industry-wide standard, but is not supported yet in commodity hardware. In contrast to CCIX, which treats host and accelerator as peers, CXL has a CPU-centric, asymmetric view. With respect to the link speed, CXL is limited by PCIe (i.e. with PCIe 4.0 it will reach 16 GT/s, with PCIe 5.0, it is 32 GT/s).

CAPI/OpenCAPI is yet another contender for a cache-coherent interconnect. Designed by IBM, it is an interconnect implemented in the CPU to access accelerators. It uses a high bandwidth, low latency interconnect that runs in POWER9 CPUs in its CAPI-2 incarnation at 32 GB/s on top of PCIe 4.0. A latency comparison of the above standards is provided in [1] (under different settings): 737ns on PCIe Gen3, <555ns on PCIe Gen4 and 378ns on OpenCAPI.

2.3 Architecture of neoDBMS

We now provide a brief overview of neoDBMS and demonstrate the need for efficient synchronization mechanisms between DBMS and smart storage to further increase the potential of NDP.

Native Storage Manager. neoDBMS is an NDP DBMS, based on PostgreSQL, designed for smart / computational storage use on non-volatile memory (NVM). Furthermore, to eliminate intermediary layers along the critical I/O path (e.g., file systems), neoDBMS relies on *native storage* [12] like NoFTL [7] and nKV [16]. neoDBMS can therefore control the physical placement of DB pages and operate directly on DBMS-controlled NVM storage.

Snapshot creation and visibility checking. Multi-versioning and MVCC are the foundation of neoDBMS, as they fit current workloads, such as HTAP, very well. However, due to multiple physically co-existing versions of tuples, every transaction needs to operate against a *snapshot* of the DB that includes all currently visible tuple versions (i.e. the *latest* committed versions visible to the transaction). The *snapshot* can include versions in the current working-set, primarily stored in the large DBMS memory, but also versions found in a cold persistent dataset already offloaded to smart storage. In order to ensure transactionally consistent NDP execution, neoDBMS collects all modifications on the host, which also include the latest changes to VID_{Map} and L2P_{Map} (described below), in a small shared state area, which is regularly flushed to smart storage. All NDP invocations also force the propagation of the current shared state, which is temporarily stored as part of a *delta buffer* (Fig. 3).

Utilizing multi-versioning and snapshots allows NDP read transactions to operate *intervention-free* on the latest committed versions visible to them without stalling. Update transactions, on the other hand, do need to consider that modifying transactions concurrently executed on both host and smart storage can simultaneously create inconsistent version branches (e.g., Fig. 1).

Version Organization and Invalidation. In-situ snapshot creation is also possible and efficient because of neoDBMSs version organization. All versions of a tuple (i.e. tuple *X*, Fig. 2) are organized in a *new-to-old* (N2O) organization [18], forming a singly-linked list [6]. Every successor version includes the reference (RecordID) to the predecessor, forming a version chain (i.e. $X.v_1$ reference to $X.v_0$ Fig. 2). Due to *native storage*, all logical *RecordIDs* can be resolved to physical persistent pointers through a logical-to-physical address mapping L2P_{Map} (Fig. 2). Additionally, neoDBMS introduces a VID_{Map} per DB-Object which serves as an entry-point to the version chain. For all tuples the *RecordID* of the latest version is stored together with a VID, which is the same for all versions of a tuple. The N2O organization enables *one-point invalidation* to determine the currently visible version of a tuple to a transaction. Each version contains the timestamp of either the creating/updating transaction (i.e. TX_C, TX_U Fig. 2).

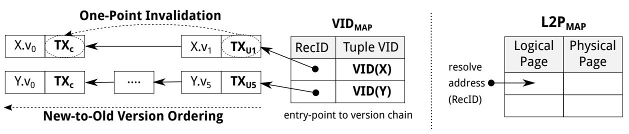

Figure 2: Version organization/invalidation of neoDBMS.

Smart storage architecture and interfaces. The type of smart storage that we use in this paper provides an array of processing elements (PEs) for NDP execution. An NDP invocation first propagates the small shared state to the device and then partitions the VID_{Map}, assigning each partition to one of the PEs. Each PE can independently start a *visibility check* task and *intervention-free* calculate the *snapshot* (i.e. traverse the version chain) with all visible tuple versions corresponding to TX_{NDP} as output. For in-situ data interpretation and page layout navigation, neoDBMS employs format parsers and layout accessors, which also support resolving *RecordIDs* (Fig. 3). The output is the complete and consistent snapshot of the entire dataset with respect to the invoking transaction, to be consumed by follow-up operations. That said, modifying NDP operations cannot guarantee consistency, due to the limitations of the existing host-smart storage synchronisation mechanisms, which are ill-suited either due to lack of performance or exceeding complexity. neoDBMS addresses this issue by sharing a small host memory area with the smart storage over latency-optimized CCIX for cache-coherent SVM (Fig. 3). In this area, we realize a light-weight and efficient shared lock table.

3 CACHE-COHERENT SHARED LOCK TABLE

We now describe the *cache-coherent shared lock table (ccSLT)* and its integration in neoDBMS and smart storage. Additionally, we demonstrate how it enables the invalidation and synchronization of individual tuple versions in update NDP settings.

Organization. Our lock table supports tuple/row-level locking. It is allocated in ccSVM and is therefore shared between DBMS and smart storage. ccSVM is provided and managed by CCIX-Agents which provide low latencies for small random accesses, while automatically maintaining cache-coherence and address translation. This is especially useful for the placement of our small lock-entries. The foundation of the lock table is a hash table which manages the access to certain tuple versions and can thus be used as a synchronization mechanism. The optimal size of the hash table varies depending on the workload characteristics and is therefore configurable. In neoDBMS, we target small hash tables sizes with just a few megabytes. To determine the bucket position, we employ a hash function using a combination of VID and tuple version number as key. Each bucket represents a small queue that stores the timestamps of transactions. The queue is designed to fit into a cache-line of 64 bytes. With a slot size of eight bytes, it can store up to eight transactions requesting a tuple lock. The first slot represents the transaction currently *holding* the lock, while the other slots are waiting to *acquire* the lock and therefore represent pending lock requests. Hash collisions can potentially reduce concurrency, but can be mitigated through careful table sizing, and utilizing the unique and monotonically increasing nature of the VIDs. Collisions do not impact the transactional consistency. If a lock is released, the entire queue is shifted by one slot to pass the lock to the next transaction. If the queue is already full, transactions must wait until a free slot becomes available. The shared lock table is controlled by the host via software interfaces. In addition we designed a *hardware tuple locking module* (Fig. 3) that accepts lock requests, handles hashing, and manages atomic CCIX transfers. The module also notifies processing elements about changes in the lock status. To ensure race-free placement and release of locks, neoDBMS relies on atomic compare and swap (CAS) operations, which are supported both by the tuple locking HW module and CCIX itself.

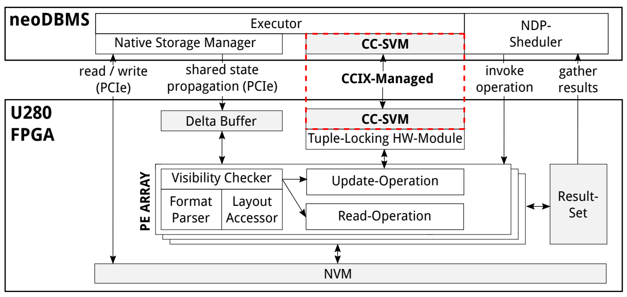

Figure 3: Architecture of neoDBMS on smart storage.

Protocol. Consider TX_{NDP} seeking to update tuple t (i.e. latest committed version t_{v1}). TX_{NDP} requests and obtains the tuple lock for t_{v1} and proceeds to create a new version t_{v2}. The request traverses the queue (single cacheline access) and inserts the TX timestamp via an atomic CAS operation at the next free slot. In the current example the queue is still empty and the timestamp can be stored in the first slot, thus acquiring the lock immediately. At the same time, another transaction TX_{HOST} starts updating the same tuple. Since TX_{NDP} is not yet committed, the tuple version visible to TX_{HOST} is still t_{v1}. TX_{HOST} can not obtain the tuple lock, as the first slot has already been acquired by TX_{NDP} and coherently synchronised with the host over CCIX. Instead, TX_{HOST} queues up into the next free slot. Execution of TX_{HOST} therefore halts, but TX_{HOST} registers to receive commit or abort events of the transaction that will pass on the lock. This avoids having to continuously check the queue status in host transactions. Device transactions, on the other hand, need to check the lock queue continuously by polling. However, CCIX can cache the lock queue on device, and as long as no changes are performed by the host, only local accesses are necessary, thus not increasing contention. In this manner, the ccSLT can serve as a lightweight synchronization mechanism to prevent the creation of inconsistent version branches.

Now consider the commit process of TX_{NDP}. TX_{HOST} is still unaware of the modifications performed by the smart storage. Yet, all transactions in neoDBMS are initiated by the DBMS. Although operations can be offloaded to *smart storage*, they still commit or abort on the *host side*. This also includes the release of any tuple locks that the transaction holds. The *result set module* on smart storage (Fig. 3) gathers all VIDs modified in the process and sends them back to the host, which can then efficiently release the tuple locks and, at the same time, fetch updates to the VID_{Map} due to the list of modified VIDs. Upon commit, TX_{HOST} is notified and can continue execution. Based on the updated VID_{Map}, TX_{HOST} is now aware of the updates and the, now outdated, in-memory version of t_{v1} is invalidated. This could be simplified even further by also employing ccSVM to perform address mappings like VID_{Map}. TX_{HOST} continues and requests a lock for t_{v2}.

4 EXPERIMENTAL EVALUATION

Experimental Setup. The experiments are conducted on an ARM Neoverse N1 System Development Platform (N1-SDP) that serves as a host. It has 4× ARM N1-CPUs operating at 2.6GHz and a total of 16GB RAM. To prototype the smart/computational storage, a CCIX-capable Xilinx Alveo U280 FPGA (AU280) board is connected via a CCIX-enabled PCIe Gen3 slot to the host. The Alveo U280 is equipped with two off-chip 16GB DDR4 DIMMs.

Both devices support CCIX, with the N1-SDP being configured as CCIX-HA (Home Agent), and the AU280 as CCIX-RA (Request Agent). This setup allows neoDBMS to allocate contiguous pinned memory on the host, which can then be accessed in a cache-coherent manner from the NDP-PEs on the AU280. Pinning the memory helps to further improve the latency of CCIX by reducing the address translation overhead. The dedicated tuple locking HW module on the AU280 is implemented in Bluespec.

Experiment 1: Shared Lock Table Latencies (Host and Device). We begin with a general experiment to investigate the end-to-end performance of the CC Shared Lock Table (Fig. 4), measuring lock request latencies on host and device, and investigating the behaviour under different degrees of contention. We instrument both the host and the smart storage device to continuously create lock requests. At the same time, we gradually increase the degree of contention on the other side. Beginning from 0% parallel requests up to 100%, where both sides request locks simultaneously (Fig. 4).

Insight: Under low contention, tuple versions can be locked on the host with low latencies of 80-100ns. Under high contention and increasing synchronization effort due to CCIX coherence events, the host still provides robust performance with latencies around 400ns, which is comparable to the latencies archieved by NUMALink [11] (380-500ns). Using non-local atomic read/write CCIX transactions, the device is able to maintain constant lock-latencies of 800ns, and is *unaffected* by the increased contention due to directly updating the memory of the host.

This experiment indicates that the placement of CCIX-HA and CCIX-RA is an important design decision. If the *host* executes a latency sensitive OLTP-style workload, it should be the one acting as CCIX-HA. Alternatively, if the workload is *NDP-update intensive* and incurs high frequency NDP-modifications, the CCIX-HA should be placed on the *smart storage device.*

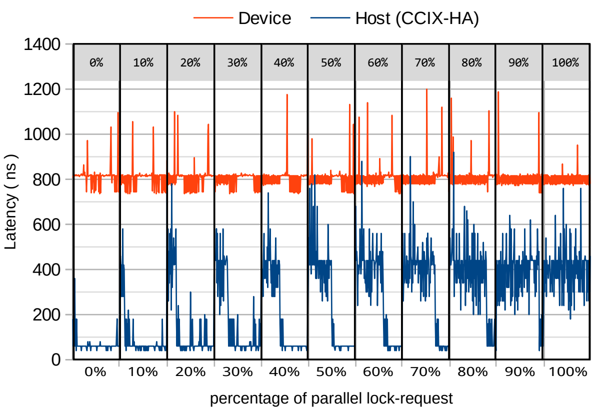

Figure 4: Shared Lock Table Latencies with increasingly parallel lock-requests.

Experiment 2: CCIX vs. PCIe. Next, we examine the data transfer latencies for CCIX compared to PCIe, and demonstrate the benefit of using the latency-optimized CCIX (Fig. 5).

We consider two transfer granularities 64B and 8KB, and measure them on device. First, cacheline-sized (64B) transfers are most relevant for our cache-coherent Shared Lock Table, as they represent the hash bucket size of the queue managing the tuple version locks. Second, we consider transfer sizes of 8KB since neoDBMS/PostgreSQL mostly operates on 8KB page granularity. CCIX-internal pre-fetching allows the device to work on local cache, achieving 80-100ns. We also report the *cache-miss* latencies for non-local atomic read/write accesses. Notably, it is consistent with the device latency of 800ns in experiment 1 (Fig. 4).

Insight. CCIX provides excellent latencies (80-100ns) for small granularity (64B) accesses. CCIX latencies are comparable with NUMAlink [11] latencies of ≤500 ns that are typical for large-scale main-memory systems.

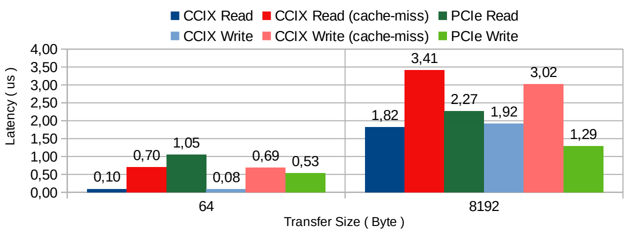

Figure 5: CCIX and PCIe Latencies on AU280 smart storage.

Experiment 3: PostgreSQL Locking in comparison to Shared Lock Table To evaluate the impact of an additional tuple locking mechanism in neoDBMS, we compare the internal PostgreSQL shared tuple locking functionality, which can only lock on host, against our CC Shared Lock Table (Fig. 6). In this experiment, tuple locking was isolated to measure the end-to-end latencies for placing a tuple lock. We execute 1M lock requests that either lock the tuples sequentially, as they were inserted, or randomly. Fig. 6 shows a histogram of the distribution of latencies for individual lock requests.

Insight. With 80-100ns for sequential and random locking, the ccSLT yields much lower latencies than PostgreSQL, which peaks at ca. 520ns. Sequential accesses profit from CCIX-internal prefetching, shown by the shorter tail of the distribution.

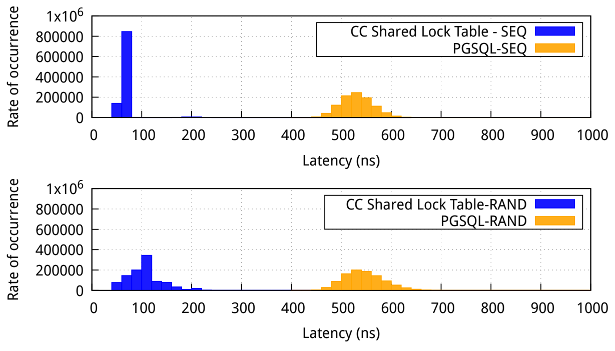

Figure 6: Shared Locks PostgreSQL vs neoDBMS.

5 CONCLUSIONS

In this paper, we introduce tuple synchronization and invalidation mechanisms on cache-coherent shared virtual memory for near-data update processing in host/smart storage environments. Our approach utilizes CCIX, a novel cache-coherent low-latency accelerator interface. Beyond the synchronization examined here, ccSVM also allows cooperative processing of large, highly linked structures (e.g., graphs) on smart storage devices and host.

Acknowledgments. We thank the anonymous reviewers for the valuable comments. This work has been partially supported by *DFG neoDBMS – 419942270* and *HAW Promotion, MWK, Baden-Würrtemberg, Germany.*

REFERENCES

[1] Brian Allison. 2018. Introduction to the OpenCAPI Interface. https://openpowerfoundation.org/wp-content/uploads/2018/10/Brian-Allison.OPF_OpenCAPI_FPGA_Overview_V1-1.pdf.

[2] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *Proc. CIDR*.

[3] Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Carsten Heinz, Christian Knoedler Tobias Vinçon, Andreas Koch, and Ilia Petrov. 2022. neoDB: In-situ Snapshots for Multi-Version DBMS on Native Computational Storage. *submitted ICDE* (2022).

[4] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *Proc. FAST*. 29–41.

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

[6] Robert Gottstein, Ilia Petrov, and et al. 2017. SIAS-Chains: Snapshot Isolation Append Storage Chains. In *ADMS@VLDB*.

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

[8] CCIX Consortium Inc. 2016. An Introduction to CCIX - White Paper. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf. (2016).

[9] Sungchan Kim, Hyunok Oh, and et al. [n.d.]. In-storage Processing of Database Scans and Joins. *Inf. Sci. 2016* ([n. d.]).

[10] David Koenen and Jeff Defilippi. 2017. CCIX: a new coherent multichip interconnect for accelerated use cases. http://www.armtechforum.com.cn/attached/article/C7_CCIX20171226161955.pdf.

[11] Timothy Prickett Morgan. 2014. SGI Scales Up HANA On UV NUMA Systems. https://www.enterpriseai.news/2014/06/03/sgi-scales-hana-uv-numa-systems/.

[12] Ilia Petrov, Andreas Koch, Sergey Hardock, Tobias Vincon, and Christian Riegger. 2019. Native Storage Techniques for Data Management. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2048–2051. https://doi.org/10.1109/ICDE.2019.00236

[13] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proc.SIGMOD*. 893–908.

[14] David Sidler, Zsolt Istvan, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. 2017. DoppioDB: A Hardware Accelerated Database. In *Proc. SIGMOD*.

[15] Tobias Vinçon, Christian Knoedler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi andLukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Update-aware Near-Data Processing for Database Systems onNative Computational Storage. *submitted VLDB* (2022).

[16] Tobias Vincon, Lukas Weber, Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2020. nKV: Near-Data Processing with KV-Stores on Native Comp. Storage. In *Proc. DaMoN*.

[17] Tobias Vincon, Lukas Weber, Arthur Bernhardt, Christian Riegger, Sergey Hardock, Christian Knoedler, Florian Stock, Leonardo Solis-Vasquez, Sajjad Tamimi, Andreas Koch, and Ilia Petrov. 2020. nKV in Action: Accelerating KV-Stores on Native Computational Storage with Near-Data Processing. *PVLDB* 12 (2020).

[18] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 12.