CINDA: Using Cache-Coherent Interconnects for Accelerating Databases by Enabling Near-Data Processing of Update Transactions

Sajjad Tamimi, Arthur Bernhardt, Florian Stock, Ilia Petrov, Andreas Koch, *Member, IEEE*

***Abstract*—Near-Data Processing (NDP) has been proven useful to accelerate Database Management Systems (DBMS) that handle infrequently accessed data stored in slow persistent storage. A key challenge for such an architecture is the synchronization of host-based and NDP operations, which require fine-grained interactions especially when the NDP device can also update (modify) the DBMS data autonomously.**

This paper introduces CINDA, the first full-stack computational storage capable of accelerating *both* read and update (write) database transactions using NDP. The proposed system relies on a hybrid host-device interface to enable the DBMS accessing persisted data, offloading computation to the storage device, and coordinating concurrent device-update operations with the host-update ones. A hybrid interface utilizes a cache-coherent interconnect such as CCIX or CXL for low-latency synchronization using a shared-lock table, and PCIe DMA for high-throughput bulk I/O. We evaluated the effectiveness of the proposed approach in a CCIX-based system by realizing an FPGA-based NDP-capable computational storage device and customizing an NDP-capable DBMS based on PostgreSQL to support update NDP operations. Our full-stack evaluation using the YCSB benchmark demonstrates that CINDA can deliver ≈4.2× end-to-end speedup when executing long-running update transactions directly on the storage device, while the host DBMS performs frequent short updates.

***Index Terms*—Near-Data Processing, Computational Storage, CCIX, CXL, FPGA, Database Management Systems.**

I. Introduction

Computational storage improves overall system performance by moving processing closer to the data and leveraging the higher device-internal bandwidth and parallelism vs. bottlenecking host-storage external interfaces.

The NDP approach is especially beneficial if large volumes of data would need to be moved to the host, such as in long-running data analytics (read-only) transactions over infrequently accessed data stored in cold storage [1].

Many of today's Database Management Systems (DBMS) workloads, however, are *hybrid* workloads, which have *both* long analytical as well as frequent but short update (write) transactions occurring in parallel, requiring highly efficient synchronization between host and NDP for consistent updates.

S. Tamimi, F. Stock, and A. Koch are with Embedded Systems and Applications Group, Technical University of Darmstadt, ([last name]@esa.tu-darmstadt.de). A. Bernhardt and I. Petrov are with Data Management Lab, Reutlingen University, ([firstname].[surname]@reutlingen-university.de).

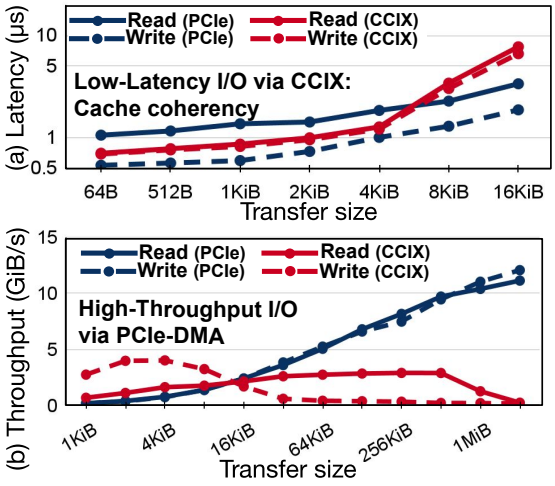


Fig. 1. Host-device (a) latency [2] and (b) throughput comparison between PCIe and CCIX traffic for different data lengths on PCIe Gen3 x16. As both CCIX and PCIe use the same PCIe physical layer [3], similar speedups are expected in later PCIe generations. Note that the CCIX latencies refer to remote (non-local) access to the host memory, which carries extra latency for write accesses due to the coherency protocols [2].

Although recent studies [4], [5] have highlighted the significance of update-intensive workloads in large-scale database systems, the use of NDP techniques to improve them has been hindered by the traditionally employed PCI Express interface (PCIe) with its focus on high-throughput bulk transfers, instead of low-latency synchronization.

Cache-Coherent Interconnects (CCI), such as Compute eXpress Link (CXL) or Cache Coherent Interconnect for Accelerators (CCIX), can avoid that bottleneck, as shown in Fig. 1. In addition to efficient fine-grained synchronization, they also enable the utilization of the cache-coherent Shared Virtual Memory (ccSVM) programming model between the host and NDP-device which can considerably simplify the use of shared pointer-based data structures, especially when using byte-addressable Non-Volatile Memory (NVM) as persistent storage.

Utilizing the NDP technique in a modern database system that works under hybrid workloads poses multiple challenges. First, executing long-running transactions on the device concurrently with update transactions on the same record on the host leads to *write/read* conflicts. To address this, [1], [6]–[9] proposed a snapshot-based mechanism to ensure transaction consistency for executing read-only transactions on the device.

Second, executing transactions including update operations on the device causes *write/write* conflicts with the concurrent transactions on the host. Addressing this issue requires low-latency cache-coherent host-device interactions. Due to its long latencies, PCIe is unsuitable for this purpose [10], [11].

To tackle this challenge, [2], [12] proposed employing a CCI, namely CCIX, as a low-latency synchronization mechanism between the host and device to coordinate the execution of concurrent transactions on the host and the device. These prior works focused mainly on the low-level synchronization mechanism to realize a lock table that could be efficiently shared between host and device. However, they focused on evaluating synchronization in isolation, without fully integrating these mechanisms into the on-device architecture or software-side DBMS, largely due to the absence of a suitable hybrid host-device interface. Thus, they did not address how to execute NDP *update* transactions near physical data in the presence of concurrent host-side update transactions. Alleviating this limitation is essential for building and evaluating an *end-to-end* updateNDP-capable system.

This paper introduces CINDA, an FPGA-based computational storage device capable of autonomously executing database operations with *both* read and update (write) operations near physical data on the NDP device. The proposed storage, along with an updateNDP-capable DBMS (a modified version of neoDBMS [1], [6]), in turn based on PostgreSQL 12, forms a complete end-to-end system architecture.

The contributions of this paper are as follows:

(i) *UpdateNDP-capable computational storage based on a novel H-NSI:* We propose a computational storage that uses our *Hybrid Native-Storage Interface* (H-NSI) to enable the DBMS to offload database operations, including read/update operations, to the device. H-NSI combines the traditional Native Storage Interface (NSI) advantages, such as the removal of legacy abstraction layers between the host and storage (e.g., file systems) [1], [6], with the capability to flexibly switch between PCIe bulk transfers and CCI synchronization messages in the host-NDP device interface.

(ii) *A novel hardware-accelerated NDP engine integrating a low-latency shared-lock handler:* We propose a new NDP engine capable of processing *both* read and update operations near physical data on the storage. The engine uses a novel shared-lock handler to guarantee transactionally consistent execution of the update operation. In addition, the engine autonomously executes NDP operations and leverages the byte-addressability of the NVM storage and shorter internal-device latency to improve overall on-device execution time.

(iii) *Full-stack end-to-end evaluation with an industrial-strength DBMS:* We perform in-depth benchmarking of our full-stack hardware-software architecture when executing update-intensive workloads, including both long-running as well as high-frequency update transactions, from the YCSB benchmark.

Note that this paper focuses on a detailed description of the *hardware* architecture and *hardware/software* interface for processing database update/modification operations near physical data within update environments. The database-side of the project cannot be discussed here in detail due to space reasons, it has thus been covered in *dedicated* publications: [1], [6], [8] focused on processing read-only operations near physical data, and [2], [12], which introduced a new stand-alone locking mechanism, but without integrating it into the NDP engine and the software-side DBMS. Both of these integration *are* addressed in the current paper.

We begin with a brief introduction to the host-device interface technologies, the DBMS, and discuss related works in Sec. II. Then, we present the CINDA system by explaining our updateNDP-capable DBMS in Sec. III-E, followed by H-NSI in Sec. III-B, and our updateNDP-capable computational storage in Sec. III-D. The end-to-end evaluation setup and results are presented in Sec. IV, before we conclude in Sec. V.

II. Background and Related Work

A. Cache-Coherent Host-Device Interface Technologies

PCI Express (short: PCIe) is a dominant interface for connecting storage devices/accelerators to the host and is optimized for transferring large chunks of data to reach maximum throughput. Even though, PCIe supports optional advanced features like Address Translation Service (ATS) and Page Request Interface (PRI), however, most available devices do not support cache coherency. This is also due to the longer PCIe latencies which makes implementing cache-coherence protocol over PCIe impractical [10], [11]. In the following, we will briefly introduce CCI technologies, i.e., CCIX and CXL, which enable PCIe-based host-device cache-coherency, before discussing our design choices.

1) Cache Coherent Interconnect for Accelerators (CCIX): CCIX [3] was the first truly multi-vendor standard that extends PCIe to support data coherency between end-point devices. This standard was introduced by the CCIX Consortium in 2016 and is supported on ARM- and x86-based CPUs. In CCIX, the host and end-point devices interact with one another in a *non-CPU-centric* manner, allowing different heterogeneous devices (e.g., CPUs and accelerators) to share data coherently. The CCIX interconnect enables multiple devices to share data in a cache-coherent manner and to use a common virtual address space. This requires that the host/device owning the physical memory can act as a Home Agent (HA) to provide cache-coherent accesses to it. In turn, a remote device employs a Request Agent (RA), along with a local system cache, to communicate with an HA for coherent access to the memory managed by the HA. In this work, we rely on the simple direct-attached CCIX topology [3], where the FPGA implements an RA and is directly connected to the host providing the HA.

2) Compute eXpress Link (CXL): CXL [13] is the newer multi-vendor standard which was introduced in 2019 by CXL Consortium. The CXL standard defines CXL.io, CXL.cache, and CXL.mem sub-protocols for device enumeration and configurations, sharing host memory in a cache-coherent manner, and accessing on-device memory, respectively. The CXL protocol defines three device types. Type-1 devices use CXL.io and CXL.cache for cache-coherent host-device accesses and PCIe-DMA transfers (e.g., smart NICs) but cannot expose

device memory to the host. Type-2 devices (e.g., GPGPUs, FPGAs) use CXL.io, CXL.cache, and CXL.mem, allowing cache-coherent accesses, PCIe-DMA transfers, and mapping device memory to the cacheable system memory. Type-3 devices use just CXL.io and CXL.mem for memory expansion without cache-coherency.

In contrast to CCIX, CXL's end-point devices are connected to the host in a *CPU-centric* manner, where the CPU "primarily" manages coherency when using its HA. The end-point Type-2 device, equipped with a Device Coherency Engine (DCOH) and system cache, allows its local operations to perform non-local cache-coherent access to the remote memory owned by the host.

3) Choice of CCI: CCIX vs. CXL: The hardware reported here employs CCIX as a CCI between the NDP device (i.e., FPGA) and host. The CCI enables both parties to share data in a coherent manner, and use a common virtual address space. In both CCIX and CXL (i.e., CXL.cache), the interconnects and system caches automatically maintain the coherency of accessed data with the remote memory without the software driver being involved in data management.

The more modern CXL standard defines a CCI capability similar to CCIX, in so-called Type-2 & 1 devices that employ the sub-protocol CXL.cache. However, even at this point in time (November 2024), only few CXL.cache-capable devices and hosts have become commercially available.

Prior studies that report on CXL either employ simulation or discuss only Type-3 devices [14], [15]. These devices use CXL.mem sub-protocol and do not support data coherency and virtual address translation that is equivalent to CCI-capabilities provided in CCIX. In addition, latencies, throughput, or similar characteristics reported in these studies [14], [15] are in fact *not* comparable to the CCI results presented here, as they only consider CXL.mem for memory expansion *without* utilizing CCI capabilities.

Since we were determined to benchmark our approach on *actual* hardware, we were forced to employ CCIX for our evaluation. The ARM Neoverse N1 System Development Platform (N1-SDP), which has been used in this study as a host, is the *only* host that is actually qualified to provide CCI over a CCIX-enabled PCIe slot. However, looking ahead, from our understanding of the CXL specifications, we expect that our results will also fully apply to current systems using CXL as CCI, once CXL.cache-capable NDP devices and hosts become more common in 2024/2025.

B. DBMS Background

1) Workloads: A transaction is a sequence of database operations that must be executed in a single logical unit of work. A database workload refers to a predefined and parameterized collection of transactions that operate on a dataset including multiple database objects, each of which represents a database table. Modern DBMSs operate under two types of workloads that can be combined in hybrid workloads [16] and weighted arbitrarily: long-running analytical transactions and frequent but short transactions. The former primarily involves queries that necessitate significant data transfer from persistent storage and require access to large portions of cold data. The latter primarily involves simple but frequent and short combinations of read, write, or update operations. Prior work (Sec. II-C) has shown that executing read-only long-running transactions on the device, while handling frequent but short transactions on the host, improves system performance by reducing data movement and memory pollution. CINDA takes this approach further by targeting the execution of long-running update/write transactions on the device, alongside frequent but short transactions executing on the host. A key limitation that hinders execution on the device is the traditional PCIe, which is optimized for high-throughput bulk transfers.

2) MVCC and its Record Version Organization: Concurrency control techniques enable DBMSs to guarantee the transactional consistency of concurrent operations while executing concurrent transactions that read/write to the shared data entity, which might cause inconsistencies or incorrect results. A Multi-Version Concurrency Control (MVCC) [17], [18] is a well-known scheme that leverages the properties of modern hardware and is suited for hybrid workloads. In an MVCC-based system, every modification to a record (e.g., *insert*/*update*) results in a new version of that record, so-called *record version*, which is stored out-of-place.

In MVCC-based DBMSs, each record version is an independent entity with a creation, an invalidation timestamp, and a pointer to its predecessor or successor. Each record version is identified by these timestamps, the transaction identifier (txID) from the calling transaction which increases monotonically, and a logical pointer, forming a *version chain*. The process of determining record versions for the called txID, known as *version visibility checking*, is vital for query executors, whether operating in a host-side DBMS or within the NDP engines in the proposed architecture (as detailed in Sec. III-E).

There are two primary methods of organizing version chains [19]: Oldest-to-Newest (O2N) and Newest-to-Oldest (N2O). In the O2N, the head of the version chain points to the oldest record version, facilitating long analytical transactions, while in the N2O, favoring frequent but short transactions, it points to the newest version. In addition, the N2O facilitates record updates by eliminating the need for in-place invalidation of the previous record version, which reduces memory access and computational load. The core DBMS used in this study relies on the N2O organization and so-called VID-mapping (as explained in Sec. II-B4) to facilitate synchronization, record version invalidation, and shared-state handling.

3) DBMS Record and Page Format: To perform visibility checking, query executors require each record to include visibility information stored in its header (Fig. 2-a). Upon creation, records are placed within a 'page'. Among different page formats, the N-ary Slotted Model (NSM) page format is a widely used page format in relational databases and is considered optimal for handling modifications. This format, as it stores all attributes of a single record contiguously within a page, is well suited for update-intensive workloads that often require access to all record attributes. The core DBMS in this study (as detailed in Sec. II-B4) employs an NSM page format, as shown in Fig. 2-c. A page consists of a header, slot pointers, and records. Slot pointers, indicating each record's

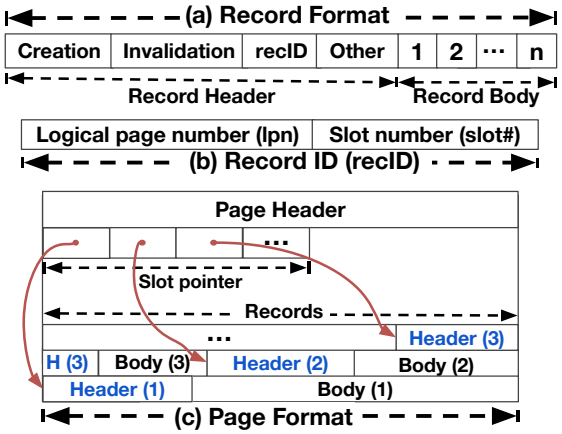


Fig. 2. (a) Record format, (b) recID format, and (c) N-ary slotted Model (NSM) page format used in this study.

offset and length inside the page, are placed after the page header and filled from the top, while records are filled from the bottom. Page sizes, determined by DBMS configuration, generally range from 2KiB to 32KiB.

4) Update-Aware NDP DBMS: neoDBMS [1], [6], [8] is an update-*aware* NDP DBMS (c.f. Sec. II-C) capable of offloading just *read-only* NDP queries to the storage and executing them in the presence of the concurrent read and write (update) operations on the DBMS software-side on the host. In this work, we extended the core neoDBMS further, as explained in Sec. III-A, now enabling it to execute a query with *write (update)* operations directly on the storage device, alongside ongoing write (update) operations on the host.

In the following, we first explain how the DBMS manages persisted data. Then, we describe how the DBMS uses the shared-state buffer to guarantee transactional consistency in the execution of read-only NDP operations.

a) Data Management: In neoDBMS, all persisted data, including records, VID-mapping tables, and L2P-mapping tables, are organized into *pages*. To provide an entry point for each version chain [1], [6] a Virtual ID-mapping (VID-mapping) table is utilized. This table contains unique VIDs for all record versions in the version chain, with each VID linking to the latest record version in its chain. The recID for each record contains a logical page number (lpn) and slot number of the record on the page (slot#), as shown in Fig. 2-b, enabling the executor to look-up and calculate the physical address of a record.

The VID-mapping table is hierarchically structured in three-page layers: vidBase $\rightarrow$ vidVector $\rightarrow$ vidPage. The vidPage pages contain the actual VID information, and the vidVector pages hold the physical addresses of these pages. The vidBase pages hold the physical addresses of vidVector. Therefore, the NDP engine must read through the first two-page layers (vidBase and vidVector) before reaching the vidPage layer. The L2P-mapping table is stored similarly in two-page layers: l2pBase $\rightarrow$ l2pVector. The l2pVector pages contain the actual VID information, while l2pBase pages hold the physical addresses of these pages.

b) Shared-State Buffer: The shared-state buffer is a small ($\approx$100KiB) memory region that tracks all differences between the persisted data already moved to persistent storage and the most recent data from ongoing processes on the DBMS. The DBMS propagates pages from this buffer to the actual storage in two modes. In the first mode, called *flush & append*, the buffer regularly flushes full pages to the persistent storage as flushed pages. In the second mode, called *pass-along & cache*, even not-yet-full pages, containing data of still-in-progress host transaction, are flushed to the storage before each NDP invocation, allowing the read NDP operations to proceed independently without requiring coordination with the host. Note that executing update operations as NDP on the device requires a synchronization mechanism, detailed in Sec. III-C.

C. Related Work on Using NDP

Previous studies on the utilization of NDP techniques to improve the overall performance of database applications can be categorized into three categories.

Studies in the first category primarily focused on scenarios involving read-only workloads. [20]–[22] as a pioneer in leveraging the higher I/O-bandwidth of smart storage, proposed offloading a portion of data processing tasks to these storage devices. Later, JAFAR [23], Biscuit [24], PolarDB [25] also explored the use of the NDP technique for accelerating size-reducing database operations. While these studies improved DBMS performance in read-only settings, it's important to note that write/update operations have become increasingly important in large-scale systems, such as Twitter/X [5] and Facebook/Meta [4]. Similarly, DBMS operating under hybrid workloads must handle not only read but also write operations.

Studies within the second category have suggested the use of NDP techniques for DBMS systems operating in a hybrid workload setting. In [1], [6], [8], the authors proposed offloading read operations (form a transaction) to the storage device while allowing concurrent transactions to update data on the host. The system proposed uses a snapshot-based mechanism to ensure transaction consistency (update-aware NDP). This approach has been adopted later on by [9], which requires a host-side pre-processing step for snapshot construction, yielding increased data movement and write amplification. Even though [9] processes read operations relatively close to the persistent storage, all data access requests still pass through PCIe. This latency remains a limiting factor, preventing utilization of low latency on-device access. Although these approaches improve overall system performance by processing long-running read transactions on the device, it lacks support for executing transactions with update operations near the physical data.

Studies in the third category suggested techniques for processing transactions with update/write operations close to the data. [2], [12] introduced to use of a low-latency synchronization mechanism through CCIX between the host and storage device to prevent write/write conflicts during the execution of update operations on the device. While that method has demonstrated success in delivering comparable latency to large-scale NUMA systems, it primarily focuses on

the synchronization mechanism and does not integrate it into the NDP DBMS stack or address storage management.

In this study, the CINDA system ensures transaction consistency even for NDP updates in the presence of the concurrent updates on the software-side DBMS.

III. Proposed Architecture

This section describes our new CINDA architecture. It enables a DBMS to execute queries with *both* read and write (update) operations on the computational storage system, while guaranteeing transactionally consistent execution with concurrent transactions on the host. Fig. 3 shows the overall overview of the CINDA system architecture, comprising an updateNDP-capable DBMS (at the top) and computational storage (at the bottom) that are interacting via H-NSI.

The H-NSI (in the middle) is a key element in the proposed system, as it enables the exchange of fine-grained, low-latency coherency messages between the host and the computational storage device. The combination of this low-latency interface with the conventional PCIe interface forms the essential infrastructure for the H-NSI implementation. The flexibility offered by H-NSI, i.e., switching between low-latency and high-throughput traffic, enables the implementation of a synchronization mechanism between host and device executors, used when executing update operations to prevent unexpected write/write conflicts. The integration of the synchronization mechanism into the hardware-side (i.e., NDP engine) and DBMS software-side of the system, along with the novel H-NSI, forms an end-to-end system capable of executing the queries with not only read but now also write operations.

This section presents different elements of the proposed system, beginning with introducing an updated NDP-capable DBMS. Then, we continue with the H-NSI and shared-lock mechanism. Finally, we provide detailed information on the micro-architecture of the proposed computational storage and NDP engine, and discuss the integration of the components.

A. UpdateNDP-Capable DBMS

Fig. 3-a shows the logical overview structure of the updateNDP-capable DBMS that enables offloading update operations to the NDP engine within the storage device through the H-NSI. This DBMS is based on neoDBMS (c.f., Sec. II-B4). The system comprises several key components as follows. The *query executor* is interacting with the user interface and determines the distribution of operations between the host and the device executors. The *host executor* is a query engine on the host and manages the execution of operations on the software-side of DBMS. The *NDP scheduler* is responsible for pushing down NDP operations to the NDP engine(s) on the computational storage. The *lock manager* provides a synchronization mechanism between host executor(s) and NDP engine(s), see Sec. III-C for further details.

Enabling the host-side DBMS to fully support update operations performed as NDP requires only few, highly targeted changes: The shaded modules in Fig. 3 highlight the host-side shared-lock table support with 200 Lines Of Code (LoC), the NDP scheduler with an additional 150 LoC, and the host

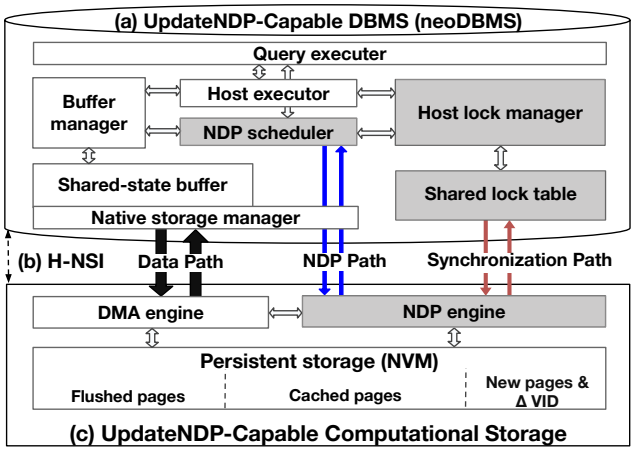


Fig. 3. Logical overview of the CINDA system architecture including (a) UpdateNDP-capable DBMS at the top, (b) H-NSI in the middle, and (c) Computational storage designed to manage both read and update NDP operations at the bottom. Note that black, blue, and red arrows indicate data path, NDP path, and synchronization path within the H-NSI, respectively. Besides H-NSI, shaded areas indicate extended/modified modules compared to [6], [8].

lock manager, which extends DBMS-tuple locks to support the locking mechanism with an extra 50-100 LoC, relative to the update-aware neoDBMS. In the following, we explain how the native storage manager partitions the persistent storage. Then, we discuss what functionality the NDP scheduler must provide to the NDP engine to enable the interpretation of database pages and records in situ.

1) Persistent Storage Partitioning: The native storage manager logically divides the persistent storage into multiple partitions as shown in Fig. 3-c. The first partition stores flushed pages of all types, including NSM, VID-mapping, and L2P-mapping pages. The second partition holds cached pages which are temporarily transferred to the persistent storage (i.e., pass-along & cache) before starting an NDP operation, are only used by the NDP engine, and will be discarded after execution. The third partition is dedicated to pre-allocated memory for the NDP engine to store new pages, e.g., result records of an NDP operation, or holding logging information, which will be transferred to the host after NDP engine execution (specified by ΔVID in Fig. 3-c).

2) In Situ Data Interpretation: To facilitate transaction execution as NDP on the device, the NDP scheduler must provide all required information to the engine. This information is required for navigating and interpreting records and pages. It is forwarded to the device during the NDP invocation and includes: MVCC-related information such as txID, physical address of vidBase, address translation information (physical address of L2PBase), NDP operation type and addresses to the pre-allocated memories, shared-lock handler configuration (more details in Sec. III-C), and NDP engine optimization parameters such as the number of active engines.

B. Hybrid Native Storage Interface (H-NSI)

The native storage interface eliminates legacy abstraction layers and block-device interfaces between the storage and the

host, enabling the DBMS to directly control the persisted data and the device hardware [1], e.g., by invoking the NDP engine. While this interface is well-suited for read-only transactions [1], [8], executing transactions that involve *update* operation requires fine-grained, low-latency synchronization between the host and storage executors to ensure transactional consistency [2], [12]. To address this challenge, we introduce the Hybrid Native Storage Interface (H-NSI) that enables DBMS to seamlessly transition between high-throughput PCIe transfers and low-latency CCI messages. H-NSI enables the DBMS to manage the persisted data directly on the device using the *data path*, offload NDP operations via the *NDP path*, and handle low-latency synchronization with a *synchronization path*. In the following, we first outline these paths and then describe the steps the software-side DBMS takes to invoke and validate an offloaded NDP operation.

1) Data Path: In a traditional system architecture, the DBMS employs READ/WRITE_PAGE commands via the file system (for instance, ext4) and block device to read/write data from/to the persistent storage device. However, with the removal of these intermediate layers in the CINDA system, these commands have been replaced with READ_PAGE and WRITE_PAGE commands for *directly* reading from and writing to the physical persistent storage device (here: non-volatile memories, NVM), respectively, as shown in Fig. 3-b.

2) NDP Path: To enable control of the NDP engines on the device by the DBMS, we introduced NDP command and NDP response operations. The NDP command performs NDP and carries out in-situ data interpretation, as detailed in Sec. III-A2. The latter returns the status of the NDP engine to the host. This is performed by raising an interrupt from each engine upon completion of execution, and then accessing the engine control registers to read the actual status updates.

3) Synchronization Path: In the CINDA system, we have enhanced the native storage interface by integrating a synchronization path as shown in Fig. 3-b. This addition allows for seamless switching between bulk PCIe-DMA transfers and synchronization traffic via the low-latency CCI. The CCI enables the CCI-capable computational storage device to access the host's memory and virtual address space while automatically maintaining coherency (c.f. Sec. II-A1). Such functionality enables the updateNDP-capable DBMS to efficiently manage concurrent update transactions through the newly introduced lock handler mechanism (see Sec. III-C).

4) NDP Invocation & Validation: Offloading and executing an operation on the NDP device involves a four-step process. In the first step, the NDP scheduler requests the buffer manager to *pass-along & cache* (c.f. Sec. II-B4b) pages, even not-yet-full pages, from the shared-state buffer to the persistent storage using the data path commands. It also requests the native storage manager to allocate space in the persistent storage for the new pages. In case of the NDP operation runs out of space, it halts the execution and notifies the software-side DBMS to allocate more storage. Then, it requests the lock manager to provide the virtual address of the shared-lock table located in host memory that is *shared* between the host and the device through the synchronization path. Note that the DBMS allocates this shared-lock table during DBMS initialization.

In the second step, the NDP scheduler invokes the engine(s) by using the NDP path commands. This involves configuring the NDP engine by writing its control registers. In the third step, once the operation execution is complete, the NDP engine notifies the NDP scheduler of the completion via the NDP path. In the fourth step, after completing the NDP transaction, the software-side of DBMS verifies the validity of the offloaded transaction's execution. To this end, it reads ΔVID, which contains the recIDs of the update records, from persistent storage and checks for write/write conflicts with concurrent transactions on the host that might occur during the execution of the offloaded operation. If the validation is successful, the DBMS updates the VID-mapping table with the new record's recID and releases the locks associated with these records. If the validation fails, the DBMS does not update ΔVID, and the newly created records are disregarded.

C. Shared-Lock Handler Mechanism

A write/write conflict occurs when concurrent transactions from both the host and the device attempt to update the same record. Two approaches can be considered to address this conflict: pessimistically locking the entire table before the NDP operation, or using an optimistic approach. While locking the table eliminates the conflicts, it severely limits system concurrency by blocking host-side updates. In contrast, for the optimistic approach, transactions proceed under the assumption of "no conflicts" and validate results afterwards upon completion of the NDP operation. Even though this approach enables concurrent host updates, it faces challenges of its own, such as potential starvation of read-intensive transactions [26], wasted resources if validation fails [27], and overhead from transferring NDP operation sets from storage for the validation.

To address this quandary, we employ a fine-grained locking mechanism called *shared-lock table* as a synchronization mechanism between the host and device transactions. This approach allows the host executor or NDP engine to lock records individually before updating them without blocking the entire table. In this manner, we increase system concurrency by enabling concurrent host updates and reduce the likelihood of host or NDP transactions being aborted (see Sec. III-B4). The shared-lock table, realized as a hash table, assigns the *right-to-modify-record* permission to a transaction, either on the host or device. This permission is granted to a transaction that placed a *lock* in the table if no other transactions are modifying the same record. This prevents simultaneous modifications of the same record by different transactions.

Using a hash table for synchronization and locking, a widely used data structure in DBMS for locking tables, provides fast lookup and allows high parallelization compared to other structures, e.g., B-trees. In addition, the lock table can be configured (as explained in Sec. III-C2c) to perform a suitable trade-off between CCI-shared memory size and collision frequency. Note that this table is allocated in the host memory and only the virtual address of the hash table is passed to the hardware-side module during NDP invocation. In addition, any modifications made to the table are automatically synchronized

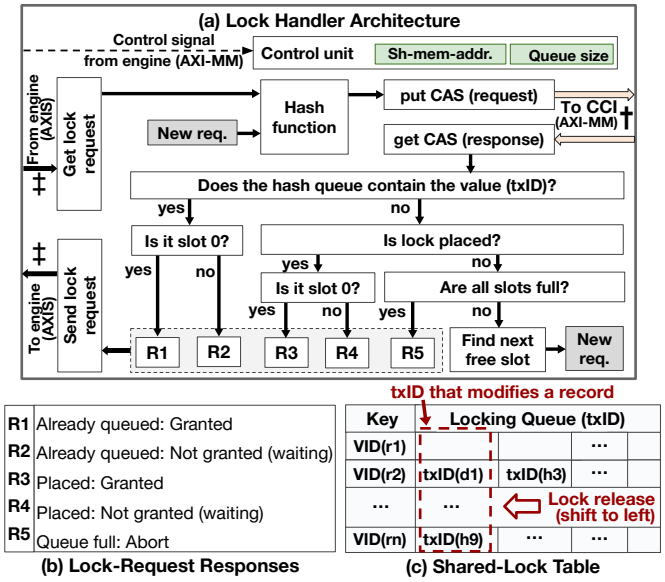


Fig. 4. (a) The architecture of hardware support for the shared-lock mechanism in the CINDA computational storage system and its responses (b), along with a logical overview of the shared-lock table (c). ‡: Interface to the NDP engine. †: Interface to the CCI bus.

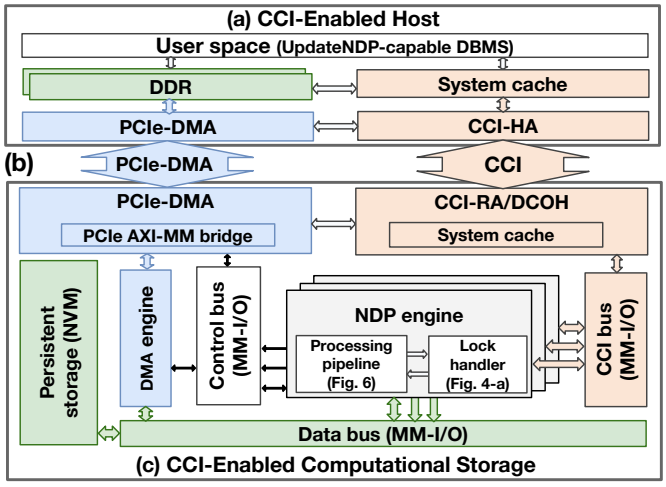


Fig. 5. General overview of the CINDA including (a) CCI-enabled host, (b) CCI-enabled physical interfaces, and (c) CCI-enabled computational storage.

using atomic operations via the H-NSI's synchronization path, which in turn employs CCI atomic updates, as explained in the following.

The proposed mechanism requires support from both the software-side of the DBMS and the hardware-side of the computational storage.

1) DBMS Software-Side Support: The lock manager on the DBMS is responsible for allocating the table during DBMS initialization and managing lock requests from the host executor. As shown in Fig. 4-c, each hash table entry consists of the key and a small queue as value. The key is the VID of the record, which is unique for all record versions, and the queue contains the txID(s) of the transaction(s) wanting to update that record. The head position in the queue gains the lock for updating the record. Upon completion of an update, the queue shifts to the left, granting the lock to the next requesting transaction. For example, in the table's second row from Fig. 4-c, the VID(r2) is the key of Record Number 2. Currently, Transaction 1 from the device (txID(d1)) is updating this record as its lock holds the head position, while Transaction 3 from the host (txID(h3)) is waiting for permission to update the record.

2) Computational Storage Hardware-Side Support: The *lock handler* module, as shown in Fig. 4-a, implements the mechanism as hardware within the computational storage. It is integrated into the NDP engine, managing lock requests to the shared-lock table through the synchronization path of the H-NSI. The details of this integration are explained in Sec. III-D. Here, we explain how the lock handler processes lock requests from the NDP engine. Then, we describe its CCI and NDP interfaces and configuration registers.

a) Lock Request & Response: Incoming lock requests to the lock handler include the VID as the hash key and the NDP transaction txID as the hash value. The NDP engine receives different types of responses based on the lock's status and checks against concurrent transactions. These responses are labeled as R1, ..., R5 as shown in Fig. 4-b.

b) Interfaces: The lock handler receives the lock request from the NDP engine and responds to it via a stream interface, indicated by black arrows and ‡ in Fig. 4-a. The interface connected to the CCI is highlighted in red arrows and †. It allows the lock handler to interact with the synchronization path of the H-NSI by executing CCI atomic compare-and-swap (CAS) operations to access the host-side memory, which holds the actual locks. Using atomic CAS operations ensures locking without race conditions.

c) Configurations Registers: A control interface enables the NDP engine to configure control registers during initialization. Shown by green boxes in the figure, the sh-mem-addr register specifies the virtual address of the start of the shared-lock table between host and device. This address is dynamically generated following the initialization of the DBMS and passed to the NDP device. The queue size register specifies the maximum number of slots in each hash table entry, corresponding to the maximum number of concurrent transactions that wait to update the *same* record. If concurrent transactions exceed the queue size, the engine triggers **`abort()`** as detailed in Sec. III-E.

D. UpdateNDP-Capable Computational Storage

Fig. 5 shows the underlying infrastructure of the CINDA system architecture, highlighting the computational storage, the host, and the interconnect with CCI capabilities, such as CCIX or CXL. The host includes a "CCI-HA" (Home Agent), which extends its virtual address space to offer a coherent address space over its physical memory. Alternatively, the device is equipped with a "CCI-RA/DCOH" (Request Agent in CCIX devices and Device Coherency Engine (DCOH) in CXL type 2 devices) that enables the internal module to perform read/write operations to the coherent address space owned by the HA. The coherency of data within these address spaces is

ensured by the CCI-enabled interconnect along with the CCI-HA and CCI-RA/DCOH.

Note that the CCI is beneficial for small transfers. However, accessing a record directly from the persistent storage via CCI requires multiple small transfers (c.f., Sec. II-B3) and involves extra buffer management overhead, as DBMS buffering strategies typically use 8 KiB page sizes. Therefore, the resulting access latencies and throughput quickly exceed those of page-sized PCIe-DMA transfers. In addition, using CCI for reading and writing data to persistent storage is sub-optimal due to the virtual-to-physical address translation overhead associated with CCI accesses [2]. Our proposed architecture leverages CCI for low-latency access to the shared lock table while using PCIe-DMA transfers for high-throughput data movement.

Fig. 5-c shows the proposed computational storage architecture, highlighting components for CCI and PCIe-DMA traffic in pink and blue, respectively. This architecture integrates multiple modules: host-device interconnect, persistent storage, arrays of NDP engines, and various peripheral components. The DMA engine is responsible for transferring data from the host's physical memory to the persistent storage. The PCIe AXI-MM bridge translates PCIe-DMA traffic to AXI-MM format. The storage includes three distinct AXI-MM buses. The data bus provides a means of connecting NDP engines, DMA engine, and persistent storage. The control bus provides the host with the ability to control internal hardware. Lastly, the CCI bus allows the NDP engines to exchange CCI messages/requests with the host via the system cache in the CCI-RA/DCOH component, through the CCI interface.

E. NDP Engine Processing Pipeline

To enable NDP execution of both read and update operations on the computational storage device, we employ an array of the *NDP engines* designed with two aims: (i) Database operations should be executed without causing any write/write conflict; (ii) Database operations should be executed as an intervention-free process on the device. To achieve the aim (i), we employ the shared-lock table to synchronize concurrent database operations that could modify a record. To achieve the aim (ii), the engine starts processing the latest created records that have not been flushed to the storage, then continues processing already-flushed records residing on the persistent storage. As NDP engines are fully cognizant of details of the DBMS data formats, they can perform in-situ data interpretation, and execute autonomously from the host.

In the following, we will explain the processing pipeline of an NDP engine in six stages. Then, we will discuss the preloader and offloader modules, which improve the performance of the NDP engine using scratchpad memories. Note that the implementation of the processing pipeline using hardwired modules and soft-core is detailed in Sec. IV-A1.

1) VID-Processing: The initial stage in the execution of an NDP transaction is to extract the recID from the VID-mapping table (Fig. 6-S1). This is achieved through a three-page layers structure (c.f. Sec. II-B4a). First, the engine loads the vidBase page. Then, it loads the vidVector page using the extracted value from the previous step. Finally, it loads vidPage from storage and forwards each recID to the visibility checker stage.

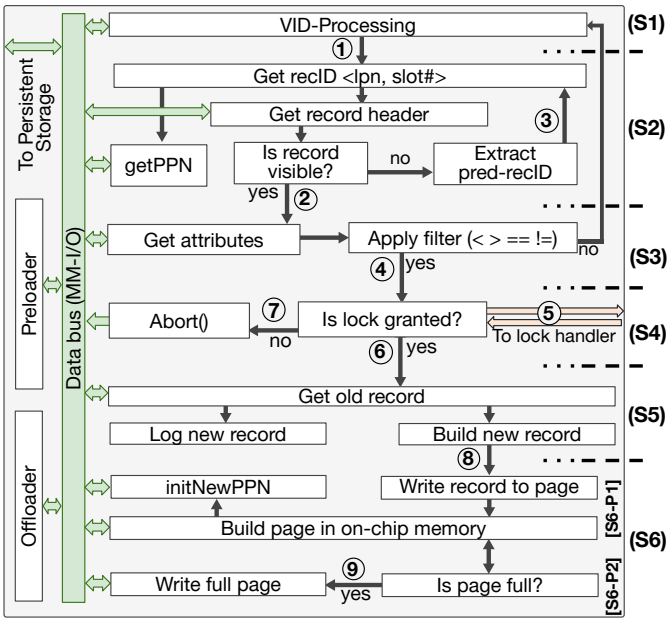


Fig. 6. The NDP engine processing pipeline with six stages: (S1) VID-Processing, (S2) Visibility checker, (S3) Filter, (S4) Lock-handler, (S5) Record-builder, and (S6) Page-builder.

2) Visibility Checker: This stage is responsible for finding a record version that is visible to the NDP operations txID. As shown in Fig. 6-S2, after receiving recID (①), which includes the logical page number (lpn) and slot number (slot#) of the record in the page, the visibility checker function uses **`getPPN(lpn)`** function to determine the physical address of the page containing the record. After resolving the page's physical address, it reads the slot pointer from the page header to determine the offset and length of the record. Then, the engine calculates the physical address of the record in the persistent storage. Finally, it accesses the record to check whether the record belongs to the transaction call time stamp by checking the visibility information on the header of the record and txID. In case of a successful comparison (②), it then forwards the physical record address to the filter stage. In case of unsuccessful comparison (③), the visibility checker function must retrieve the previous version of the record, known as the predecessor record, and check whether that is visible to the currently running transaction. The function achieves this by extracting the recID of the predecessor record from its own header. The function continues to follow the version chain until it finds a record version that is visible to the currently running transaction to the NDP operation. Note that the first record version on the version chain has *no* predecessor record, thus indicating the end of the version chain. The **`getPPN(lpn)`** module is responsible for translating an lpn to a physical page number (ppn). It uses the L2P-mapping table that resides in the persistent storage and is shared between all NDP engines.

Note that the byte-addressability access offered by emerging NVM storage allows the engine to read exactly the required amount of data, unlike page-addressed storage systems such as NAND Flash, which read the entire data block. This feature

of the underlying NVM storage improves engine performance and reduces read-amplification, where the processing element retrieves more data than needed. The latter would lead to both increased transfer times, as well as the need for larger on-chip buffers.

3) Filter: This stage is responsible for applying filter criteria to the visible record as shown in Fig. 6-S3. To this end, first, it reads the visible record and extracts selected attribute(s) from the records. Then, it checks whether the record fulfills the filter condition indicated in the NDP command operation. If the condition is true, the record is forwarded to the shared lock-handler stage (④). If not, it is discarded and the stage notifies the VID processing stage to provide a new recID.

4) Shared Lock-Handler: This stage is responsible for checking whether the current record could be modified by other concurrent operations on the software-side DBMS as shown in Fig. 6-S4. To this end, after receiving a lock request, it forwards it to the *Lock handler* (more details in Sec. III-C) module (⑤). If the lock handler grants the lock, the record information (e.g., physical record address) will be forwarded to the *record-builder* stage (⑥). If the lock is not granted and the shared-lock table's queue is not full, the engine continuously checks the lock's status until it is granted. If the queue is full, the NDP engine terminates the transaction by calling the **`abort()`** function (⑦). This function triggers an abort signal, which is then recorded in the shared status memory on persistent storage. This memory is accessible to all concurrent NDP engines. These engines regularly check for the abort signal before processing new recIDs. If they detect the abort signal, their processing is halted, and they notify the DBMS software of their status. Consequently, the software marks all new pages created by the NDP engines as discarded.

5) Record-Builder: This stage is responsible for building a new record reflecting the modifications of the update. After receiving a request for modifying attribute(s) of the record (indicated in the NDP command operation) it makes a copy of the current record version from the persistent storage into the local memory to be used as a new record version. Then, it starts to build a new record version by preparing a record header of the copied record with updated visibility information and a pointer to the predecessor record holding the current data. Finally, it modifies the targeted attributes in the new record versions and forwards it to the *page-builder* module (⑧). Note that in MVCC, all updates lead to new record versions being created, no updates are performed on the current data. The engine logs the recIDs of the new record version in the persistent storage for logging as ΔVID that allows the software-side of DBMS to validate the transaction's execution to maintain transactional consistency (c.f. Sec. III-B4).

6) Page-Builder: This stage is responsible for building a new NSM page and storing it to the NVM persistent storage, as shown in Fig. 6-S6. This stage is split into two parts. In part one (S6-P1), it allocates a new page by reserving a new ppn by **`initNewPPN`** during NDP engine initialization. The ppn specifies the physical address of the page (calculated as ppn multiplied by the page size) in the pre-allocated space within the persistent storage, as indicated by "new pages" in Fig. 3-c. The ppn, specified by the software-side of DBMS, as indicated in the NDP command operation, increments with each new page allocation. The **`initNewPPN(lpn, ppn)`** module is responsible for updating L2P-mapping in the persistent storage. To this end, it first uses lpn to calculate the physical address of the ppn entry in the L2P-mapping table. Then, it writes the new ppn value to that address, to be later used by the other software-side DBMS.

After a successful allocation and update of the L2P-mapping, S6 builds the new page (e.g., page header) in *on-chip memory*, and waits for receiving a new record to write to the page (⑧). In part two (S6-P2), the stage checks if the page is full or not. If the page is full, the stage writes the page in the persistent storage (⑨); now the engine repeats the process of allocating a new empty physical page. Using an offloader enables the engine to simultaneously flush the old page to storage while the first part S6-P1 is allocating a new page.

7) Preloader & Offloader: These units are designed to improve the performance of the NDP engine by hiding NVM latency. The preloader prefetches/pre-loads data from persistent storage to on-chip scratchpads in the background while the engine is still busy operating on other data. The offloader moves processed data from the on-chip scratchpad memory back to storage. It coalesces many smaller updates together into a single full-page write (8…64 KiB), which can be written in an efficient long burst.

IV. Experimental Setup and Evaluation

A. Experimental Setup

To evaluate the performance of the proposed system, we built the CINDA updateNDP-capable computational storage device on a Xilinx/AMD Alveo U280 FPGA (AU280), and connected it via a CCIX-capable PCIe slot to an ARM Neoverse N1-SDP host [28]. The platform is equipped with 16GB RAM and four ARM Neoverse N1 cores, similar to those used in Microsoft's Azure cloud and Google's TAU T2a cloud platforms. For the necessary infrastructure on the FPGA, we rely on the TaPaSCo framework [29] and integrate CCIX interface modules into the System-on-Chip (SoC) architecture on the AU820. Additionally, we realized H-NSI as TaPaSCo's software layer to enable seamless transitioning between PCIe-DMA transfers and CCI traffic. Since our computational storage device uses byte-addressable NVM, which currently is not widely commercially available in hardware, we *emulate* three different speed grades of NVM (modeled after Intel Optane devices) using DDR4-SDRAM [30].

1) NDP Engine Implementation: The proposed NDP engine is designed to efficiently qualify records (based on visibility checking and filtering), synchronize modifications with the concurrent transactions on the host, and build new pages. Since the data structures and algorithms on the software-side of the NDP update-capable DBMS are still in active development, we rely on a programmable CPU soft-core for selected data processing operations (c.f. Fig. 6-S1, S2, S3, S5, and S6) and a hardware accelerated modules such as a *hardwired* lock handler to synchronize the modification with the host executor (c.f. Fig. 6-S4 and Fig. 4-a) in the hardware-side DBMS interface. This approach, while less performant

than an all-hardware solution, offers far greater flexibility and ease of deployment to keep up with DBMS interface changes. To mitigate the performance loss, multiple instances of the NDP engine are utilized, allowing for distributing the NDP workload. In addition, by efficiently utilizing the application-specific data preloading and offloading through the preloader and offloader modules, data transfers and computations on the soft cores can be effectively interleaved, further improving the overall performance of the NDP engine.

Integration of the soft-core into an engine is achieved by wrapping the actual core with a control unit, hardwired lock handler (c.f. Fig. 4-a), local memory, preloader, and offloader units, as shown in Fig. 7. The *control unit* manages core initialization and signals the host upon operation completion. It interfaces with both the soft-core and host systems for efficient operation management. The DBMS software accesses and controls this unit through a control interface, i.e., AXI-MM, and is responsible for loading the pre-built firmware. The *local memory* comprises instruction memory, data memory, and scratchpad memory, all mapped to the core's address space. The *preloader* and the *offloader* are custom byte-addressable DMA engines capable of transferring data to and from scratchpad memory and persistent storage.

2) Benchmark Setup: To evaluate the NDP update functionality and involved components during high-contention settings, we utilize the YCSB benchmark [31] to generate update-intensive hybrid workloads, which we then vary in their read-to-update ratios. During the entire evaluation, the host-side DBMS is continuously and *frequently* updating the ycsb_table with *short-running* update transactions (referred to a F-S-update) affecting *individual* records:

```
UPDATE ycsb_table
SET FIELD1 = newValue1, ..., FIELD10 = newValue10
WHERE ycsb_key == randomKey;
```

Then, to stress the system and increase the contention on the shared lock table, specifically the locking mechanism, we inject a full table update as a single long-running update transaction (referred to as L-update) as shown below, either on the host-side DBMS, or on the NDP device. During this L-update execution, *every* record undergoes reading, visibility checking, and filtering processes. We vary the percentage of records that pass filtering (so-called selectivity) to be updated.

```
UPDATE ycsb_table SET FIELD1 = newValue
WHERE ycsb_key >= minKey AND ycsb_key <= maxKey;
```

Note that the on-device execution of the L-update on the NDP engine might not be validated after completion. This can occur due to potentially newer record versions having been created by concurrent F-S-update on the host-side DBMS (c.f. Sec. III-B4). To minimize abort overhead, and the need to *re-execute* the costly L-update transaction in the first place, we use the SELECT-FOR-UPDATE technique where the NDP engine locks the qualified records (visible and filtered) *early* in the processing pipeline by executing stages S1-S4 (Fig. 6). Once qualified records are successfully locked, the engine runs stages S5, S6.

3) Non-NDP Software Baseline: For the non-NDP software baseline, we use PostgreSQL 12 (pgSQL). pgSQL uses a dedicated fast Samsung 980 Pro NVMe SSD as storage, operated with a traditional storage stack, including file-system (ext4) and block device storage. Remember that existing update-aware NDP architectures, such as DANSEN in neoDBMS [8], *only* allow the execution of *read-only* NDP operations (c.f. Sec. II-B4) and do not support update NDP operations. This limitation results from lacking H-NSI with CCI capabilities and a locking mechanism to coordinate NDP operations with concurrent host updates. Both DANSEN and our work here employ PostgreSQL as host DBMS (c.f. Sec. III-A).

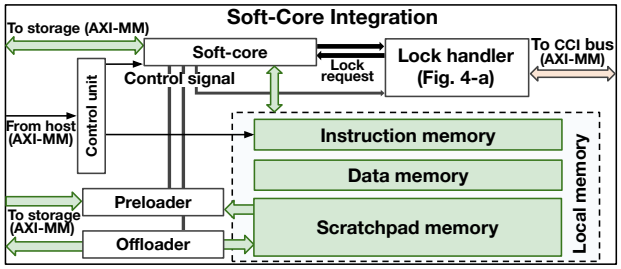


Fig. 7. Programmable soft-core in an NDP engine with supporting modules.

TABLE I
FPGA-RESOURCE AVAILABLE AND UTILIZATION.

Module name	LUTs	Registers	BRAM	DSPs
Available resources	1303680	2607360	2016	9024
Storage SoC	50.85%	21.61%	45.86%	1.23%

4) Computational Storage Configurations: Each CINDA NDP engine is based on a lock handler, preloader, offloader, and a MicroBlaze core from AMD/Xilinx (c.f. Fig. 7). The soft-core is set to 64-bit addressing mode to effectively manage data sets exceeding 4 GiB. It is optimized for "maximum performance" by enabling and setting the data cache to 16 KiB, which is the optimal configuration for processing NDP database operation [8].

The assessment of the proposed system is conducted on the ARM platform supporting 16-byte atomic CAS operations for fast synchronization via CCIX. Thus, the queue size register in the lock handler is set to 2 bytes, enabling 8 concurrent transactions per record as explained in Sec. III-C. To mitigate ATS (address translation) overhead latency, Linux huge pages were employed to allocate the shared-lock table.

We evaluate the performance of our computational storage using emulated byte-addressable NVM, based the memory timings on published latency data for Intel Optane DC from various sources [32]–[35]. To widen the scope of our experiments beyond just the Optane characteristics, we have defined three NVM latency categories: fast-NVM (r/w: 305ns/100ns), mid-NVM (r/w: 350ns/170ns), slow-NVM (r/w: 350ns/800ns). Additionally, we also observe a case where we set read and write latencies to zero without using the emulator (case no-NVM = case DRAM]), thus using the DDR4-SDRAM directly.

The proposed storage engine's flexible design allows for conducting design space exploration to identify the optimal balance between the number of NDP engines and the highest achievable clock frequency to maximize performance. The outcome of these explorations resulted in our use of 8 engines

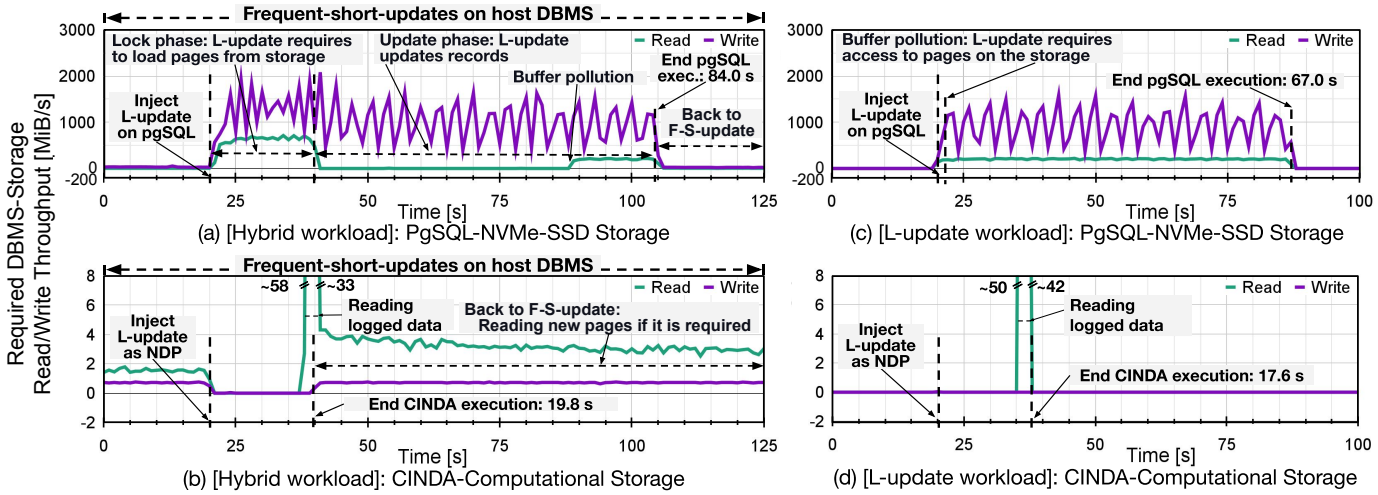


Fig. 8. A comparison of read/write throughput for (b,d) CINDA using computational storage and (a,c) PostgreSQL using NVMe storage. Note that figures (a),(b) show this comparison while the DBMS is operating under a [hybrid workload], including L-update and F-S-update transactions, while figures (c),(d) show this comparison while DBMS *only* processing long-running update transaction ([L-update]). Be sure to note the very different Y-axis scale for the PgSQL and CINDA cases, including the brief I/O spikes in CINDA around the 35s mark.

operating at 180 MHz. For the remaining modules, we used the maximum clock frequencies: the PCIe block and system cache at 250 MHz, and the emulated NVM (memory controller and NVMulator) at 300 MHz. Additionally, the resource usage of the proposed computational storage is shown in Table I.

Note that, compared to the earlier DANSEN non-cache coherent computational storage [8], the cache-coherent CINDA computational storage requires *extra* logic and routing resources to enable coherency messages through the CCI (here: CCIX). This leads to severe congestion in the chip's routing. In addition, the CINDA SoC now includes *three* different buses (see Fig. 5) that connect NDP engines to system cache, memory controller, and PCIe block, which further increases the congestion. These two factors prevent us from enabling the second memory controller on the FPGA, and achieving good timing closure with the larger number of soft-core CPUs that were realized in DANSEN.

B. Evaluations

We begin the experiments with a birds-eye-view, namely by evaluating the impact of CINDA on the overall full-stack DBMS performance. Then, we proceed to investigate individual NDP engine performance characteristics in greater detail.

*1) **Experiment 1 (DBMS-Storage Read/Write Throughput on Hybrid Workload)**:* The objective of this experiment is to evaluate the impact of the proposed system on the data movement between DBMS on the host and storage device, while executing a hybrid workload, including *concurrent* L-update and F-S-update transactions (please see Sec. IV-A2 for their descriptions). To this end, we measured the overall DBMS read/write throughput to and from the storage device while processing six million records using SELECT-FOR-UPDATE technique (also discussed in Sec. IV-A2). As shown in Fig. 8-a and -b, the host-side DBMS is executing frequent short F-S-update transactions. Then, we stress it by injecting a single long-running L-update transaction at the 20-second mark, on pgSQL in (a), and using CINDA NDP in (b).

On the non-NDP baseline in Fig. 8-a, using pgSQL and NVMe, the required read and write throughputs increase significantly after the injection of the L-update transaction. This increase is due to the intense transfers of database pages between the storage device and the host. As indicated in the figure, the DBMS first locks qualified records (lock phase) in the first 20 seconds. Then, it starts the update phase. Note the drastically raised I/O buffer contention in the update phase, necessitating the eviction of old pages to make room for new ones, which results in extra read requests to the storage, as indicated by buffer pollution in the figure. Once the L-update I/O operations are completed after 84.0 seconds (at the 104-second mark), the required throughput returns to normal levels as the system resumes processing F-S-update transactions.

In NDP using CINDA, the L-update transaction is offloaded to eight NDP engines, with both the preloader and the offloader enabled to optimize data transfers. Once the L-update is injected, these NDP engines lock the records *early* in the execution pipeline, which *halts* the host-DBMS processing of F-S-update. Thus, the host-initiated I/O drops to zero, as the computational storage is busy executing the injected update in a "sprint-to-completion" fashion. After the NDP-side L-update transaction is complete, the host-DBMS resumes processing its F-S-update transactions by fetching new records from the storage and the log information (c.f. Sec. III-B4). Note that in the CINDA system, the host-DBMS is stalled for a *much shorter* time compared to the non-NDP baseline. More detailed experiments on L-update execution times in NDP and non-NDP modes are presented later in **Experiment 4**.

*2) **Experiment 2 (DBMS-Storage Read/Write Throughput on Long-Running Update)**:* The aim of this experiment is to evaluate the data movement between DBMS on the host and storage device when *not* using SELECT-FOR-UPDATE technique. When not using this technique, there is a high risk of conflict-induced aborts during execution or invalidation of the NDP transaction, as discussed in Sec. III-B4. To ensure forward-progress, we did *not* execute F-S-update transactions

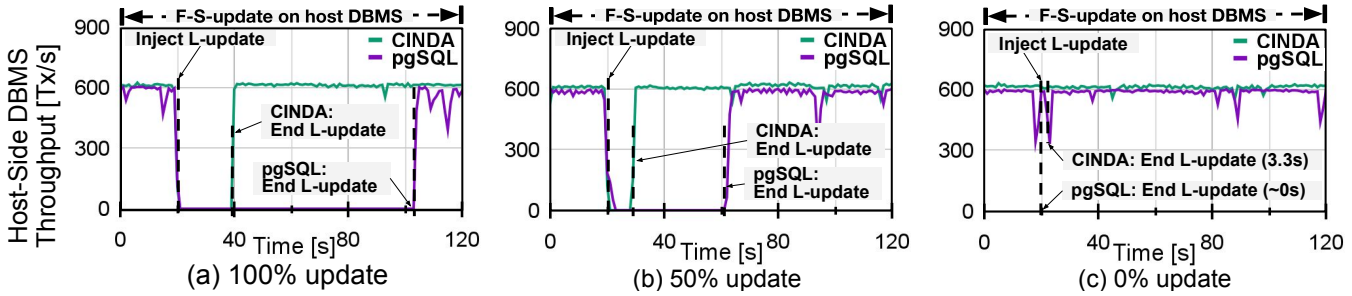


Fig. 9. A comparison of the host-side DBMS transaction throughput between CINDA using computational storage vs. PostgreSQL using NVMe storage, working under a hybrid workload for a selectivity of (a) 100%, (b) 50%, and (c) 0% of records updated in the L-update transaction.

and *only* inject the L-update transaction at the 20-second mark. The results of this comparison are shown in Fig. 8-c,d.

As expected in the non-NDP baseline, Fig. 8-c, the host DBMS starts to load pages from storage and, in parallel, updates them. After 67 seconds (at the 87-second mark), pgSQL finishes executing the transaction, which is faster than **Experiment 1**, as this time, there is no F-S-update workload in the background. In the CINDA system, as shown in Fig. 8-d, the L-update transaction is offloaded to the eight NDP engines and finishes execution in 17.6 seconds (at the 37.6-second mark). After the NDP completion, the software-side DBMS reads the logged data (ΔVID) to validate the NDP execution (c.f. Sec. III-B4), even though there are no concurrent transactions on the host in this experiment.

*3) **Experiment 3 (Host-Side DBMS Transaction Throughput):*** The objective of this experiment is to examine the impact of scaling the number of records subjected to the update operation on the performance of the DBMS under the hybrid workload. To this end, we extend the previous experiment and shrink the number of update operations in the long-running transaction from 100% selectivity (as seen in **Experiment 1**) first to 50% and then to 0%. This is achieved by altering the WHERE clause in the L-update transaction to select the records for updating. Note that even in the 0% mode, the executor still has to perform the visibility checking and filtering operations for each record.

Fig. 9 shows a host-side DBMS transaction throughput comparison between CINDA and pgSQL in terms of transactions processed per second. After injecting the L-update, CINDA's host-side DBMS transaction throughput drops to zero, as the transaction (L-update) executes on the NDP engines on the device and blocks the host-side concurrent update operations from F-S-update transactions. Upon completion of L-update on the device, the host-DBMS resumes executing F-S-update transactions. In contrast, pgSQL's throughput reduces to a *single* transaction and remains low for a longer period as the L-update transaction blocks all other concurrent F-S-update transactions on the host. Note that in traditional non-NDP mode, the DBMS is required to transfer the *complete* table from dumb storage to the host. However, in NDP mode, the NDP engines lock *only* the specific selection of records for updates, rather than locking the entire table. Using the fine-grained shared-lock table, rather than locking the whole table, enables the execution of concurrent host-side update transactions, which leads to higher concurrent host-side throughput and reduces the likelihood of NDP transactions being aborted (c.f. Sec. III-B4). While this is the same scenario that was used in **Experiment 1**, Fig. 8-a,b describes required DBMS-Storage read/write *I/O* throughput, whereas Fig. 9 details host-side DBMS *transaction* throughput.

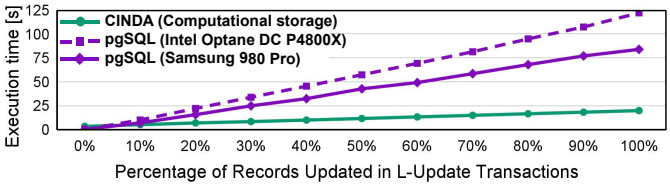


Fig. 10. A comparison of the overall execution times between CINDA and baseline (PostgreSQL) while executing L-update with an increasing fraction of records updated (higher selectivity).

Fig. 9-b and -c show the throughput comparison for 50% and 0% scenarios. In the 50% case, the L-update transaction on CINDA is shorter due to fewer records needing to be updated, and wins over the pgSQL execution due to the avoided data transfers. However, in the 0% scenario, the CINDA L-update transaction takes $\approx$3 seconds on the device. This is due to the visibility checking and filtering of each record that still needs to be performed NDP-side. pgSQL, in the 0% case, completes the execution of the L-update transaction in $\approx$0 seconds due to it using index structures to quickly determine if any records require updating at all. In the 0% case, it finds none and stops execution immediately.

Note that, in the CINDA's system, in the high contention setting, where both L-update transaction on the NDP engine and F-S-update transactions on the host attempt to update the entire table, the locking mechanism *halts* the host-DBMS processing of F-S-update. In contrast, in the low contention setting, the locking mechanism allows the host-side concurrent update operations (F-S-update) to update the qualified records without locking the entire table, which would otherwise halt the host-side workload. In addition, the CINDA host-DBMS transaction throughput achieves similar throughput as pgSQL for F-S-update. In this mode, CINDA's computational storage is used as a "dumb" (non-computational) storage device. This comparable throughput between CINDA (with N2O) and pgSQL (with O2N) shows that the chosen scheme for version change organization (c.f. Sec. II-B2) does not impact the system performance, as the version-chain-length remains small.

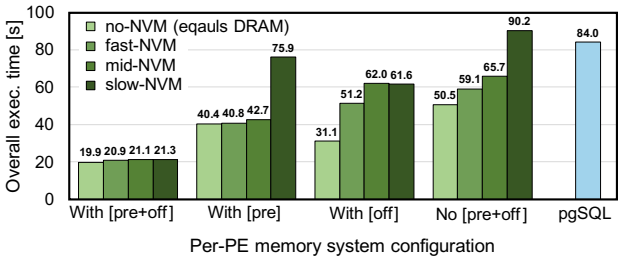


Fig. 11. Overall execution time of L-update with different per-PE memory system configurations (preloader/offloader enabled/disabled) at different NVM latencies when processing 6 million records.

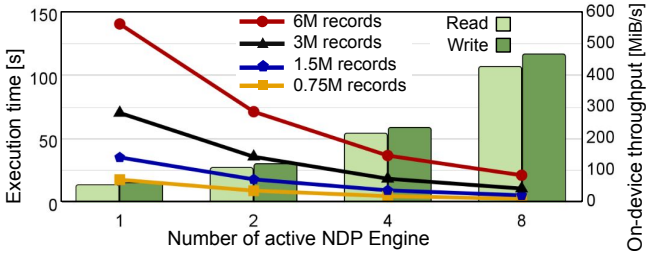


Fig. 12. The effect of increasing the number of active NDP engines while executing L-update on four different dataset sizes on the execution time (left y-axis, lines) and on-device throughput utilization (right y-axis, bars).

*4) **Experiment 4 (End-to-End Execution Time)**:* In this experiment, we compare the total execution time of our CINDA system and PostgreSQL, while processing a hybrid workload with different numbers of records subject to update. We extend the previous Experiment 3, which considered Transaction Rates, by more finely stepping the fraction of records updated (selectivity) up from 0% in 10% increments, and monitor the resulting execution times. Note that throughout the experiment, *all* of the records pass S1-S3 stages in the processing pipeline (Fig. 6), regardless of selectivity. However, only those records meeting the filter condition proceed to request a record-level lock from the shared-lock table at stage S4. Records that have been granted a lock can then be updated in stages S5 and S6. Our findings, as shown in Fig. 10, highlight a major difference between the baseline and CINDA: With the data transfers reduced by NDP, CINDA is not only considerably faster than pgSQL, it is also less sensitive to updating larger numbers of records (at higher selectivities). This is indicated by the steeper slope of the pgSQL lines in the plot. We run this experiment twice, once on a modern Flash-based Samsung 980 Pro NVMe SSD, then on a *non-Flash* Intel Optane SSD DC P4800X, which employs non-volatile memory, similar to what we emulate on our computational storage. As can be seen, for the traditional page-based DBMS workload, just using NVM instead of Flash does *not* gain benefits, as the performance on the Optane SSD is slower than that of the Flash SSD. Only when *combining* NDP and exploiting the byte-addressability of NVM, as done by CINDA, can the benefits of NVM actually be leveraged.

*5) **Experiment 5 (Effects of Memory Architecture)**:* The aim of this experiment is to assess the NDP engine performance across different configurations of the CINDA per-PE memory system (Sec. III-E7) and NVM latencies (Sec. IV-A4). To this end, we show the end-to-end execution times of processing the L-update transaction in four per-PE memory system and three different NVM latencies.

In the first setup, which is used in the previous experiments, the NDP engines operate with PEs having both the preloader and offloader (with [pre+off]), achieving the best execution time (Fig. 11). This is due to these techniques being able to *hide* even longer NVM latencies in the data transfer paths between the persistent storage and local scratchpad memory, leading to a $\approx$4.2$\times$ faster execution than on pgSQL. In the second setup, we have only enabled the preloader (with [pre]). Here, the PEs in the NDP engines incur longer execution times due to increased latency in *writing* data to storage.

Next, we enable only the offloader (with [off]), and thus cause the NDP engines to incur delays caused by the no-longer hidden extended *read* times to storage. In the fourth setup, we disable both pre- and offloading (no [pre+off]), which leads to the lowest performance.

*6) **Experiment 6 (System Scalability)**:* We further extend the previous experiment and execute the L-update transaction on different dataset sizes, measuring NDP execution times and on-device throughput. We vary the number of NDP engines, always keeping both the preloader and offloader enabled. The left y-axis of Fig. 12 shows the overall execution times, while the right y-axis shows the read and write on-device throughput used by the active engines. As expected, increasing the number of engines results in faster execution times and higher on-device throughput utilization, reaching $\approx$0.9 GiB/s read/write throughput. Our flexible hardware architecture would allow even higher levels of parallelism. However, these designs can only be implemented at lower clock frequencies and do not provide performance gains (c.f. Sec. IV-A4). Note that a more congested concurrent update operation over CCI, due to an increasing level of parallelism, does not increase the device-side lock latency [2], [12].

V. Conclusion

This paper introduced CINDA, a full-stack computational storage system for accelerating hybrid DBMS workloads. CINDA achieves this by executing both read and write database operations "close" to the stored data following the Near-Data Processing paradigm. CINDA's Hybrid-Native Storage Interface (H-NSI) provides to the DBMS both low-latency messaging via a Cache-Coherent Interconnect (CCI), and high-throughput via PCIe-DMA. This flexibility ensures efficient data access, supports database transaction offloading, and facilitates the introduction of a fast shared-lock mechanism that maintains transactional consistency between DBMS and NDP during concurrent modifications.

Our end-to-end speedup of $\approx$4.2$\times$, relative to the traditional non-NDP PostgreSQL, stems from reducing data movement and employing the shared-lock mechanism for low-latency fine-grained concurrency control. The evaluation also highlights the effectiveness of NDP engines in leveraging the byte-addressable features of NVM devices to boost performance, while hiding memory access latencies with the preloader and offloader facilities.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for valuable comments and AMD/Xilinx Inc. for hardware donation. This research was funded by DFG project #419942270 *neoDBMS*.

REFERENCES

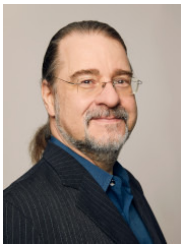
[1] T. Vinçon, C. Knödler, L. Solis-Vasquez *et al.*, "Near-data processing in database systems on native computational storage under htap workloads," *Proc. VLDB Endow.*, vol. 15, no. 10, p. 1991–2004, jun 2022.
[2] S. Tamimi, F. Stock, A. Bernhardt *et al.*, "An evaluation of using CCIX for cache-coherent host-FPGA interfacing," in *FCCM*, 2022, pp. 1–9.
[3] C. C. Inc., "An introduction to CCIX - white paper," 2016.
[4] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook," in *FAST*, 2020, pp. 209–223.
[5] J. Yang, Y. Yue, and K. Rashmi, "A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter," *ACM Transactions on Storage (TOS)*, vol. 17, no. 3, pp. 1–35, 2021.
[6] A. Bernhardt, S. Tamimi, F. Stock *et al.*, "neoDBMS: In-situ snapshots for multi-version dbms on native computational storage," *ICDE*, 2022.
[7] T. Vincon, L. Weber, A. Bernhardt, C. Riegger, S. Hardock, C. Knoedler, F. Stock, L. Solis-Vasquez, S. Tamimi, A. Koch, and I. Petrov, "nKV in Action: Accelerating kv-stores on native computational storage with near-data processing," *PVLDB*, vol. 12, 2020.
[8] S. Tamimi, A. Bernhardt, F. Stock *et al.*, "DANSEN: Database acceleration on native computational storage by exploiting ndp," *ACM Trans. Reconfigurable Technol. Syst.*, apr 2024.
[9] K. Lee, I. Jo, J. Ahn *et al.*, "Deploying computational storage for HTAP DBMSs takes more than just computation offloading," *Proc. VLDB Endow.*, vol. 16, no. 6, 2023.
[10] V. García-Flores, E. Ayguade, and A. J. Peña, "Efficient data sharing on heterogeneous systems," in *ICPP*, 2017, pp. 121–130.
[11] C. Lutz, S. Breß, S. Zeuch *et al.*, "Pump up the volume: Processing large data on gpus with fast interconnects," in *SIGMOD*, 2020, p. 1633–1649.
[12] A. Bernhardt, S. Tamimi, F. Stock *et al.*, "Cache-coherent shared locking for transactionally consistent updates in near-data processing dbms on smart storage." in *EDBT*, 2022, pp. 2–424.
[13] D. D. Sharma, "Compute express link (CXL): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy," *IEEE Micro*, vol. 43, no. 2, pp. 99–109, 2023.
[14] Y. Sun, Y. Yuan, Z. Yu *et al.*, "Demystifying CXL memory with genuine CXL-ready systems and devices," in *Proc. MICRO*, 2023, p. 105–121.
[15] H. Li, D. S. Berger, L. Hsu *et al.*, "Pond: CXL-based memory pooling systems for cloud platforms," in *ASPLOS*, 2023, p. 574–587.
[16] F. Özcan, Y. Tian, and P. Tözün, "Hybrid transactional/analytical processing: A survey," in *Proc. SIGMOD 2017*, 2017, pp. 1771–1775.
[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[18] H. Berenson, P. Bernstein, J. Gray *et al.*, "A critique of ANSI SQL isolation levels," *SIGMOD Rec.*, vol. 24, no. 2, pp. 1–10, May 1995.
[19] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," *Proc. VLDB*, 2017.
[20] J. Do, Y.-S. Kee, J. M. Patel *et al.*, "Query processing on smart SSDs: Opportunities and challenges," *SIGMOD*, p. 1221, 2013.
[21] Y. S. Ki *et al.*, "In-storage compute: an ultimate solution for accelerating i/o-intensive applications," *Flash Memory Summit*, pp. 1–30, 2015.
[22] S. Kim, H. Oh, C. Park *et al.*, "In-storage processing of database scans and joins," *Inf. Sci.*, vol. 327, pp. 183–200, jan 2016.
[23] S. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the Wall: Near-Data Processing for Databases," *Proc. DAMON*, 2015.
[24] B. Gu, A. S. Yoon, D.-H. Bae *et al.*, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *Proc. ISCA*, jun 2016.
[25] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang, "POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database," in *Proc. FAST*, 2020, pp. 29–41.
[26] X. Yu, G. Bezerra, A. Pavlo *et al.*, "Staring into the abyss: an evaluation of concurrency control with one thousand cores," *Proc. VLDB*, 2014.
[27] K. Kim, T. Wang, R. Johnson, and I. Pandis, "ERMIA: Fast memory-optimized database system for heterogeneous workloads," in *SIGMOD*, 2016.
[28] ARM, "Neoverse N1 system development platform (SDP) - documentation and support," https://developer.arm.com/ToolsandSoftware/NeoverseN1SDP, accessed October, 2023.
[29] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The TaPaSCo open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems," in *ARC*, 2019.
[30] S. Tamimi, A. Bernhardt, F. Stock *et al.*, "NVMulator: A configurable open-source non-volatile memory emulator for FPGAs," in *ARC*, 2023.
[31] B. F. Cooper, A. Silberstein, E. Tam *et al.*, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.
[32] A. van Renen, L. Vogel, V. Leis *et al.*, "Persistent memory I/O primitives," in *proc. DaMoN*, 2019.
[33] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable NVM," in *MEMSYS*, 2019, p. 304–315.
[34] J. Izraelevitz, J. Yang, L. Zhang *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," 2019.
[35] S. Eilert, M. Leinwander, and G. Crisenza, "Phase change memory: A new memory enables new memory usage models," in *2009 IEEE International Memory Workshop*, 2009, pp. 1–2.

Sajjad Tamimi is a Ph.D. candidate at TU Darmstadt (Germany) since 2019. He received his M.Sc. degree in Computer Engineering from the Sharif University of Technology (Iran) in 2016. His current research interests focus on computer architecture, modern hardware, reconfigurable computing, and embedded system design.

Arthur Bernhardt is a Ph.D. student and a member of the Data Management Lab at Reutlingen University (Germany) since 2019. He received his M.Sc. degree in Computer Science from the University of Ulm (Germany) in 2017. His current research focuses on exploring novel architectures, abstractions, and algorithms within data-intensive systems on hardware and emerging technologies.

Florian Stock worked for three years in industry as a Compiler Engineer at PACT XPP, then, he joined the Embedded Systems and Applications Group at the Technical University Darmstadt (Germany), where he now works as a Postdoctoral Researcher. His current research interests include compiler engineering, near-data processing, and geometric algebra computing.

Ilia Petrov is a Professor at Reutlingen University (Germany), where he heads the Data Management Lab. His research focus is on high-performance data management and analytics and database systems on modern hardware technologies. He holds a Ph.D. from the University of Erlangen-Nuernberg (Germany).



Andreas Koch received his Diploma in Informatics from TU Braunschweig (Germany) in 1992, where he also graduated with a Doctorate in Engineering in 1997. After working as a Post-Doc at UC Berkeley (CA, USA) from 1997 to 1999, he returned to Darmstadt to obtain his Habilitation in 2005. He joined the Department of Computer Science of TU Darmstadt (Germany) as Professor in 2005. His core research interests are domain-specific computing and the required programming tools.