Report from Dagstuhl Seminar 21283

Data Structures for Modern Memory and Storage Hierarchies

Edited by

Stratos Idreos[1], Viktor Leis[2], Kai-Uwe Sattler[3], and Margo Seltzer[4]

1 **Harvard University – Cambridge, US,** `stratos@seas.harvard.edu`
2 **Universität Erlangen-Nürnberg, DE,** `viktor.leis@fau.de`
3 **TU Ilmenau, DE,** `kus@tu-ilmenau.de`
4 **University of British Columbia – Vancouver, CA,** `mseltzer@cs.ubc.ca`

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 21283 "Data Structures for Modern Memory and Storage Hierarchies". For decades, computers consisted of a CPU, volatile main memory, and persistent disk. Today, modern storage technologies such as flash and persistent memory as well as the seemingly inevitable migration into virtualized cloud instances, connected through high-speed networks, have radically changed the hardware landscape. These technologies have major implications on how to design data structures and high-performance systems software. The seminar discussed how to adapt data structures and software systems to this new hardware landscape.

Seminar July 11–16, 2021 – http://www.dagstuhl.de/21283

2012 ACM Subject Classification Information systems → Data management systems

Keywords and phrases Cloud, Data Structures, Database Systems, Flash, Near-Data Processing, Persistent Memory

Digital Object Identifier 10.4230/DagRep.11.6.38

1 Executive Summary

Viktor Leis (Universität Erlangen-Nürnberg, DE)

The seminar brought together researchers and practitioners from the data management and systems/storage communities to discuss the implications of the modern hardware landscape on high-performance systems. Due to the pandemic, the seminar was organized as a hybrid event: Virtual participation was limited to one session per day that featured invited talks. The in-person component consisted of free-flowing plenary discussions and several smaller, focused working groups. Some key takeaways from the discussion are:

- **OS/DBMS co-design**: Traditional POSIX-style OS abstractions do not work well for data-intensive systems, leading to complex workarounds and suboptimal performance. While some of these issues could in principle be fixed by optimizing OS implementations, others require new APIs. For example, it is very difficult to implement crash-consistent data structures on top of the mmap system call.
- **Cloud**: The cloud is taking over and cloud-native data processing systems often have a a very different architecture from traditional data management systems. For example, many systems strive to separate storage from compute. This trend is enabled by ever faster networks.

- **Near-data processing**: Separating storage from compute leads to costly data movement, which may be mitigated by pushing down (parts of) the computation close to the data. Major public cloud vendors already to optimize their internal services towards this goal. The challenges is how to program such distributed and specialized hardware components.
- **Persistent Memory**: One major question discussed at the seminar was the role of byte-addressable persistent memory in future systems and whether what the "kill app" for this technology is. While there are several promising applications (e.g., graph processing or systems that require fast recovery times), it is not clear whether wide adoption will occur. Currently, the technology is quite expensive (prices per byte are similar to DRAM) and very hard to program in a crash-consistent way (e.g., writes must be carefully ordered similar to lock-free-style programming).

2 Table of Contents

3 Overview of Talks

3.1 An update on the Enzian system

Gustavo Alonso (ETH Zürich, CH)

Joint work of Gustavo Alonso, Timothy Roscoe
Main reference Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, Zeke Wang: "Tackling Hardware/Software co-design from a database perspective", in Proc. of the 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings, www.cidrdb.org, 2020.
URL http://cidrdb.org/cidr2020/papers/p30-alonso-cidr20.pdf

This talk presents the status of Enzian, a research computer being developed at ETHZ. Enzian has been designed to enable research in a wide range of topics related to how systems architecture, from both the hardware and the software perspective, need to evolve in view of developments such as accelerators and cloud computing architectures.

The talk links to several of the topics discussed during the seminar: disaggregated memory, memory hierarchies, distributed systems architecture, etc.

3.2 Deep Memory and Storage Hierachies for Scalable and Efficicient DBMSs

Carsten Binnig (TU Darmstadt, DE)

In recent years, memory and storage hierarchies have become increasingly deeper. On a single machine, additional memory and storage technologies such as PMem or NWMe SSDs have been introduced, allowing DBMSs to scale beyond the data sizes that pure in-memory systems can handle. Moreover, thanks to technologies such as remote direct memory access (RDMA) and NVMe over Fabrics, memory and storage can even scale beyond the capcities of a single machine without sacrificing too much of performance. In this talk, I have been discussing new opportunities (e.g., how to lay out data in an optimal manner across layers) and challenges (e.g., how to keep data copies consistent across layers) that arise for building scalable and efficient DBMSs when exploiting all these different layers of memory and storage on local and remote machines.

3.3 Reasoning about cloud-native data-structures

Jana Giceva (TU München, DE)

The design of data structures should no longer be driven solely by the data layout, the algorithm's access patterns and the properties of the underlying hardware. The premise is that any future data structure must also consider the impact of the relevant cloud metrics, a list that is longer than just performance and cost. This talk is a teaser into what designing a cloud data structure means in the context of scale, resource disaggregation, and novel cloud-native data system architectures. This entails reasoning in terms of cloud service components, understanding the tradeoffs where must we ensure no-data loss as opposed to good quality of service, as well as considering the impact of the whole system stack when using the underlying network-attached resources.

3.4 The data systems grammar

Stratos Idreos (Harvard University – Cambridge, US)

Data structures are everywhere. They define the behavior of modern data systems and data-driven algorithms. For example, with data systems that utilize the correct data structure design for the problem at hand, we can reduce the monthly bill of large-scale data systems applications on the cloud by hundreds of thousands of dollars. We can accelerate data science tasks by being able to dramatically speed up the computation of statistics over large amounts of data. We can train drastically more neural networks within a given time budget, improving accuracy.

However, knowing the right data structure and data system design for any given scenario is a notoriously hard problem; there is a massive space of possible designs while there is no single design that is perfect across all data, queries, and hardware scenarios. We will discuss our quest for the first principles of data structures and data system design. We will show signs that it is possible to reason about this massive design space, and we will show early results from a prototype self-designing data system which can take drastically different shapes to optimize for the workload, hardware, and available cloud budget using machine learning and what we call machine knowing. These shapes include data structure and system designs which are discovered automatically and do not exist in the literature or industry.

3.5 A Taxonomy of Database and SSDs Co-designs

Alberto Lerner (University of Fribourg, CH)

Joint work of Alberto Lerner, Philippe Bonnet
Main reference Alberto Lerner, Philippe Bonnet: "Not your Grandpa's SSD: The Era of Co-Designed Storage Devices", in Proc. of the SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pp. 2852–2858, ACM, 2021.
URL https://doi.org/10.1145/3448016.3457540

This talk discussed the numerous advantages of Co-designing Databases and SSDs. Most notably, co-designing allows a system to offload some database tasks onto storage hardware and thus obtain better performance or resource utilization. These tasks can range from simple behavioral changes in the device, such as scheduling IO operations from a latency-sensitive transaction log with high priority, to moving entire computations into the device, such as executing a portion of a query plan or transforming a log segment into a partial checkpoint. A taxonomy of offload-capable devices was presented, which organizes the devices according to the type of interface they offer. Two of these classes can benefit from further research: devices with computational features and database-storage co-designed devices. This talk summarizes a join work tutorial with Philippe Bonnet presented at SIGMOD'21 about Databases and SSDs co-design [1].

References

1 Alberto Lerner and Philippe Bonnet. *Not Your Grandpa's SSD: The Era of Co-designed Storage Devices. SIGMOD'21*, June, 2021.

3.6 NVM: Bubble Memory all over Again?

Margo Seltzer (University of British Columbia – Vancouver, CA)

Non-volatile, byte-addressable memory (NVM) has been touted as the next big revolution in persistent storage. With the the load/store access model and performance of DRAM with the persistence of flash, what could be better as a foundation for high-performance data management? In fact, the research community has been prolific in publications touting the amazing systems we'll see; yet, commercial impact has been minimal. Why?

Both the technology and hype harken back to the 1970s and the introduction of a different non-volatile technology: Bubble Memory. Like NVM, pundits predicted that bubble memory would be a game changer in our system stack. It wasn't. This talk explores the lessons we should take away from the bubble memory mania. Our after talk discussion will focus on identifying the Killer Apps that will make NVM a true game changer in the 21st century.

4 Working groups

4.1 Future databases

Carsten Binnig (TU Darmstadt, DE), Gustavo Alonso (ETH Zürich, CH), and Alberto Lerner (University of Fribourg, CH)

This work group discussed the future architecture of database systems from the perspective of modern hardware and cloud computing. The group outlined a novel system running on serverless that takes advantage of all the services the cloud provides without giving up the advantages of an actual database engine. To certain extent, what we designed is a disaggregated data processing engine.

4.2 Interface Challenges between Databases and Operating Systems

Christian Dietrich (TU Hamburg-Harburg, DE), André Brinkmann (Universität Mainz, DE), Viktor Leis (Universität Erlangen-Nürnberg, DE), and Thomas Neumann (TU München, DE)

Databases and operating systems (OS) have in common that they have to serve applications that only reveal their wishes within concrete service request. While database management systems consider users with SQL queries as "applications", they themselves are considered as "applications" by the operating system. Rooted in this dual role, the interaction between the OS and databases suffers from an expectation mismatch: Although database developers want to use general-purpose OS interfaces, they are often disappointed by the supplied performance and the given guarantees (e.g., atomicity). On the other hand, OS developers argue that databases should just use the (right) OS interface (correctly) and give more hints about the intended use of the requested resources. But, since databases are in a similar position as the OS and cannot predict the next application request, this criticism often bears no fruit.

Instead, database developers bypass central OS infrastructure (e.g., by performing direct block-device accesses) and re-implement parts of the OS functionality in user space. While these private re-implementations have the benefit of being more controllable, they make the database a problematic citizen from the OS perspective. One example of this implementation pattern is the *buffer manager* in the DBMS, which has its OS equivalent in the *page cache*; both are caches for the secondary storage, get filled on demand, and evict pages with or without write-back. In contrast to the page cache, the buffer manager allows for fine-grained control about eviction and eviction order, which is necessary for atomicity guarantees on data updates. However, such a process-local buffer manager occupies resident memory, which the OS, in contrast to page-cache pages, cannot easily reclaim when the memory pressure rises.

In our working group, we discussed the `mmap()` OS interface, which would allow the database to rely on the page cache as its buffer manager. However, even if leaving eviction control aside and focusing only on read-only workloads, the current Linux implementation is problematic: When reading random pages from a high-speed NVMe SSD, the OS fills up its page cache and makes the data available in the processes' virtual address space, whereby it nearly reaches the bandwidth limits of the underlying device (around 3 GiB/s). At some point, the page cache reaches its limit, memory becomes scares, and the OS starts to evict data pages. For this, the OS removes the evicted pages from the virtual address space, which requires the OS to ensure mapping consistency on all thread-executing cores. With a TLB shootdown, which is sent as an inter-processor interrupt (IPI), the OS requests TLB flushes on the other CPU cores. In our benchmark, these shootdowns dominate the benchmark performance after second 25 and, after second 40, each 4K read provokes on average 4 IPIs (see Fig. 1).

To tackle this issue, the Linux system call `madvise()` already provides two flags (`MADV_DONTNEED` and `MADV_COLD`), whereby the application can hint that certain pages are not needed in the near future, which would allow for a lazy unmapping of pages without TLB shootdown. However, with the current implementation, it seems that shootdowns are still performed eagerly. Also the vectorized `madvise()` variant `process_madvise()`, which would also eliminate system-call overheads, currently performs one TLB shootdown per unmapped page instead of batching them after the system call.

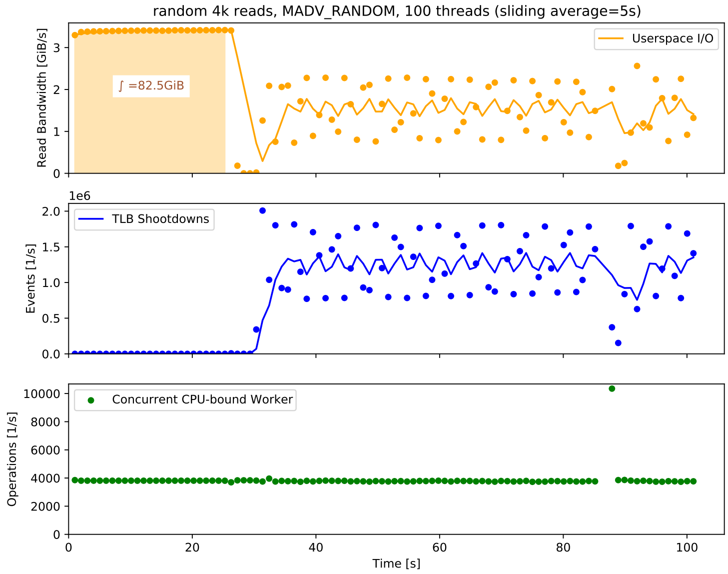

Figure 1 Random 4K Reads from Memory-Mapped Device. Benchmark was executed on a AMD EPYC 7713 64-Core Processor with 512 GiB RAM

A more general idea to improve `mmap()` for database systems could be to introduce process-local (or even mapping-local) page-cache partitions. While these would be under control of the kernel, the user-space application should be able to finely control page eviction and consistency/atomicity requirements. Preferably, this control could be exercised through asynchronous low-overhead kernel interfaces (e.g., `io_uring`) in order to keep up with the bandwidth of NVMe SSDs RAIDs.

4.3 Out-of-Memory Data Structures

Viktor Leis (Universität Erlangen-Nürnberg, DE) and Thomas Neumann (TU München, DE)

Joint work of Alfons Kemper, Viktor Leis, Alberto Lerner, Ulrich Meyer, Thomas Neumann, Alexander van Renen, Kai-Uwe Sattler, Jens Teubner

B-trees are still the most common out-of-memory data structure and perform well when the data is larger than main memory. Specialized in-memory data structures (e.g., radix trees), on the other hand, are often faster for pure in-memory workloads. In this working group, we discussed how to close this gap by designing a data structure that combines ideas from both data structures. The goal is to be as fast as pure in-memory data structures on workloads where the working set fits into main memory, while transparently supporting larger than main memory data sets as well. The key idea behind our data structure, which we code-named Dagstuhl-Tree, is to cache individual keys in a fast in-memory data structure (e.g., a radix tree). The on-disk representation is still similar to a traditional B-tree, but caching and eviction occur at a key granularity. Thus, the proposed data structure can not only speed up in-memory workloads (due to the fast in-memory data structure) but also out-of-memory workloads (due to more fine-grained cache utilization).

4.4 A Preview of Upcoming Cache Coherency Technologies

Alberto Lerner (University of Fribourg, CH), Gustavo Alonso (ETH Zürich, CH), Kai-Uwe Sattler (TU Ilmenau, DE), and Jens Teubner (TU Dortmund, DE)

This working group discussed the Compute Express Link (CXL) protocol [2], an emerging memory coherence standard that allows peripheral devices to manipulate their host's memory seamlessly. The protocol, backed by a large consortium of companies, is expected to soon appear in a new generation of commercial CPUs, GPUs, NICs, accelerators, and potentially SSDs. The group found that CXL, along with other coherence protocols such as CCIX [1] and ETH's Enzian machine's [3], have the potential to open the design space for database systems as follows. Peripheral devices can now read and update data without explicitly moving it first, thanks to the coherent hardware support. New database systems can deploy distributed, *exo-CPU* computations while still benefiting from a shared memory abstraction. The group presented its preliminary research questions and discussed the availability of academic and commercial platforms to support such efforts.

References

1 CCIX Consortium. *Cache Coherent Interconnect for Accelerators – Base Specification Revision 1.a Version 1.0.* `https://www.ccixconsortium.com`, July, 2019.

2 CXL Consortium. *Compute Express Link Specification 2.0.* `https://www.computeexpresslink.org/download-the-specification`, October, 2020.

3 Abishek Ramdas, David Cock, Timothy Roscoe, and Gustavo Alonso. *The Enzian Coherent Interconnect (ECI): opening a coherence protocol to research and applications. LATTE '21*, April, 2021.

4.5 Near-data processing – State-of-the-art and open problems

Marcus Paradies (German Aerospace Center – Jena, DE)

Introduction

The seminal idea of offloading computations close to the data to reduce unnecessary data movement dates back more than three decades. Although the general concept of near-data processing has been around for quite a while already, it only recently got enough momentum to foster an increasing research demand from academia and a more widespread development of near-data processing solutions from the industry. The growing interest in near-data processing is mainly driven by two factors: (1) ever-growing data volumes & an increasing demand for advanced analytics, and (2) increasing heterogeneity & specialization of hardware components (processing, memory & storage, and network) to data-intensive workloads. Growing data volumes pose tremendous challenges to data systems and demand more complex and scalable system architectures, including complex network topologies and deep I/O hierarchies, potentially spanning local storage, remote storage, and cloud storage resources [3]. Therefore, near-data processing can be found today across all hardware components of distributed data systems, in particular network and memory & storage infrastructures. In summary, near-data processing provides the following advantages:

- **Reduction of data movement.** Deep I/O hierarchies in distributed execution environments increase data movement since data has to be moved potentially across multiple storage tiers and the network before it is processed. This can lead to potential bandwidth bottlenecks on the network and storage stacks. Besides contributing to a waste of bandwidth resources, data movement also consumes a considerable amount of energy. Near-data processing reduces the amount of data that needs to be transferred and thus saves bandwidth resources and potentially reduces the overall energy consumption of the system infrastructure.
- **Reduction of access latency.** Offloading computations can also reduce data access latency by avoiding unnecessary data transfers across the network and storage stack.
- **Reduction of load on the host CPU.** Near-data processing enables freeing up scarce host CPU cycles by offloading parts of the computation into the network or the storage stacks. Hardware specialization in the network or in storage might even lead to faster computations compared to running the same operation on general-purpose CPUs.
- **Increase of data privacy & security.** The reduction of data movement increases data privacy and security since only data that is relevant for the processing is moved out of the storage system and across the network.

Types of Near-Data Processing

Near-data processing comes in different flavors and is typically considered in the context of offloading computations into *memory* (e.g., *Processing-in-Memory (PIM)* or *Near-Memory Processing*), *storage* (e.g., *Computational Storage*), or the *network* (e.g., *In-Network Processing (INP)*). Along the entire data path, all components, such as disks, network cards, switches, and memory modules become *active*, i.e., can perform certain operations on the data they handle.

Processing-in-Memory (PIM) and Near-Memory Processing

PIM addresses the *memory wall problem*, i.e., the growing discrepancy between microprocessor performance and DRAM memory speed [8]. By placing a lightweight processing unit in/near memory, PIM helps to alleviate the memory bandwidth limitations of traditional von-Neumann architectures. Recent developments, such as Samsung's HBM-PIM and AxDIMM and UPMEM's PIM solution based on a DRAM processing unit (DPU), demonstrate the increasing interest and momentum from industry to commercialize and embrace PIM-based hardware components for memory-bound applications [9]. While PIM is a promising technology which is gaining more traction lately, challenges for a widespread adoption arise from a limited set of supported operations and the lack of tools and programmability features.

Computational Storage

Computational storage devices (CSD) allow offloading computations into or near the storage device. While in-storage processing has been advertised since the early 90's, only recently commercial CSD products (e.g., Samsung SmartSSD, NGDSystems Newport, and Scaleflux) based on SSDs became available [5, 4]. More broadly, computational storage refers to a family of different technologies, which provide computational resources close to the storage devices. Computational resources might reside within the storage device itself (e.g., some embedded ARM cores) or are connected via a peripheral interconnect (e.g., a CSD based on reconfigurable hardware (FPGA)). Despite the availability of CSD hardware, there is currently no standard interface mechanism available, which uniformly describes the interaction between the host software and the CSD device. Standardization efforts in the NVMe working group *Computational Storage* discuss extensions of the NVMe protocol for offloading computations (e.g., orchestrated through eBPF).

In-Network Processing

Modern programmable networks create the opportunity for in-network processing, i.e., offloading computations from end hosts into network devices such as programmable switches and smart NICs [1, 7]. Programmable switches, such as Barefoot Networks' Tofino, have a flexible parser and a customizable match-action engine. To process packets at high speed, this architecture has a multi-stage pipeline where packets flow at line rate. Each stage has a fixed amount of time to process every packet, allowing for lookups in memory, manipulating packet metadata and stateful registers, and performing boolean and arithmetic operations [1]. While programmable switches offer impressive performance, they only provide a limited memory size, a limited set of supported actions (e.g., simple arithmetic, data manipulation, and hashing operations), and few operations per packet to guarantee execution at line rate.

Abstractions and Primitives

Near-data processing can be employed for specific usage scenarios, i.e., to offload a well-defined, fixed operation (e.g., SQL filtering & aggregation, regex searches, compression & encryption, etc.) or user-defined operations (e.g., UDF-like operations or kernels). Depending on the offered programming model (e.g., match-action, data-flow, etc.) and offloading mechanism (e.g.,OS/container/VM, bitstream, or eBPF), the expressiveness and composability of operations can vary dramatically. For example, initial PIM solutions only offered simple arithmetic operations to be offloaded, while recent computational storage products allow the execution of arbitrary user code in a containerized manner directly inside the SSD. It remains an open problem, how future programming models for near-data processing will look like. Even promising offloading mechanisms, such as using eBPF in the context of computational storage, struggle to allow general (and potentially complex) offloads of operations into storage devices. In cloud deployments, near-data processing opportunities are usually not directly exposed, but shall be used through well-defined service interfaces (e.g., AWS S3 SELECT), which poses the question of how much control future data systems (DBMSs and data-intensive systems like Spark, Flink, etc.) will have over such abstracted service interfaces. As of today, the most common usage of near-data processing is to offload pre-selected tasks (i.e., operator-level) into memory, storage, or the network. Few examples allow offloading arbitrarily complex operations (i.e., query-level / pipeline-level) or even run the entire DBMS inside the storage device [2].

Open Research Problems

Given the recent excitement about programmable hardware components (memory, storage, and network), there is a large number of open research (and technical) questions that will have to be addressed. The following provides an (incomplete) list of open problems:

- **Programming models and offloading mechanisms:** Programming models are currently mostly kernel-based in some supported programming language (e.g., p4 or eBPF). It is unclear how multiple near-data processing compute units (e.g., programmable switches and programmable SSDs) can be programmed under a unified programming model. The offloading mechanisms are device-specific and usually coupled to a specific protocol (e.g., NVMe in the context of computational storage).
- **Capabilities of near-data processing compute units:** Seminal works on near-data processing already prove the suitability of computation offloading for bandwidth-bound operations, where offloading an operation would lead to a significant reduction in data volume to be transferred. Nevertheless, new hardware devices (e.g., FPGAs, low-energy CPUs, ASICs) for near-data processing with drastically different performance characteristics will have to be evaluated for relevant data-intensive use cases. Besides purely non-functional requirements, also limitations that stem from the programming model have to be taken into account (e.g., p4 and eBPF pose certain restrictions on the types of operations that can be offloaded).
- **Offloading granularity:** It is an open question, at which granularity offloading tasks should be pushed into near-data processing compute units (e.g., sub-operator [6], operator, pipeline, query, or entire DBMS or data system).
- **Scheduling and Offload placement:** Given complex and deep I/O hierarchies with potentially multiple offloading opportunities, offload scheduling and placement become challenging research problems. Imagine a complex three-tier storage hierarchy with programmable SSDs & HDDs and as cold data archive a cloud-based storage service with UDF-like operator offloading. An interesting aspect is the (potentially negative) impact of near-data processing on the usefulness of caches and buffer managers.
- **Security & Performance isolation:** Near-data processing compute units are usually less powerful than full-fledged server CPUs (in particular for low-energy processors in storage devices). Since such resources will be shared by potentially many applications, performance isolation is of utmost importance. Further, unauthorized data access outside of the own local execution context must be prevented (imagine the potential danger of ransomware attacks that could be enabled through computation offloading into storage devices).
- **Cost models:** Cost models can provide a means to steer the scheduling and offloading placement depending on a generic cost metric, which allows pushing the operation to be offloaded to the optimal near-data processing compute unit. Developing such cost models is an open research problem.
- **Dealing with heterogeneous hardware and execution environments:** Data systems (e.g., DBMSs) run in different execution environments (e.g. on-premise or in the cloud), which determines also the opportunities for detecting offloading capabilities in a potentially complex system landscape. In a cloud setting, the entire storage stack (and therefore explicit control over offloading decisions) might be hidden behind an abstract service API. Generic offloading mechanisms and programming models have to be developed in order to allow data systems to leverage potentially diverse near-data processing opportunities in different execution environments.

References

1 Sapio, Amedeo and Abdelaziz, Ibrahim and Aldilaijan, Abdulla and Canini, Marco and Kalnis, Panos. *In-Network Computation is a Dumb Idea Whose Time Has Come.* Proceedings of the 16th ACM Workshop on Hot Topics in Networks, 2017

2 Jong Hyeok Park and Soyee Choi and Gihwan Oh and Sang Won Lee. *SaS: SSD as SQL Database System*, Proc. VLDB Endow., 2021
3 Marcus Paradies. *CryoDrill: Near-Data Processing in Deep and Cold Storage Hierarchies*, 2019, 9th Biennial Conference on Innovative Data Systems Research, 2019.
4 Antonio Barbalace and Jaeyoung Do. *Computational Storage: Where Are We Today?*. 11th Conference on Innovative Data Systems Research, 2021.
5 Gu, Boncheol and Yoon, Andre S. and Bae, Duck-Ho and Jo, Insoon and Lee, Jinyoung and Yoon, Jonghyun and Kang, Jeong-Uk and Kwon, Moonsang and Yoon, Chanho and Cho, Sangyeun and Jeong, Jaeheon and Chang, Duckhyun. *Biscuit: A Framework for Near-Data Processing of Big Data Workloads*, ISCA, 2016.
6 Maximilian Bandle and Jana Giceva. *Database Technology for the Masses: Sub-Operators as First-Class Entities*, Proc. VLDB Endow., 2021
7 Blöcher, Marcel and Ziegler, Tobias and Binnig, Carsten and Eugster, Patrick, *Boosting Scalable Data Analytics with Modern Programmable Networks*, Proceedings of the 14th International Workshop on Data Management on New Hardware. 2018
8 Onur Mutlu and Saugata Ghose and Juan Gomez-Luna and Rachata Ausavarungnirun, *A Modern Primer on Processing in Memory*, CoRR, 2020
9 Juan Gomez-Luna and Izzat El Hajj and Ivan Fernandez and Christina Giannoula and Geraldo F. Oliveira and Onur Mutlu. *Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture*, CoRR, 2021

4.6 Non-volatile Memory in Database Systems

Kai-Uwe Sattler (TU Ilmenau, DE), Alexander Baumstark (TU Ilmenau, DE), and Muhammad Attahir Jibril (TU Ilmenau, DE)

Non-volatile memory (NVM) started as a concept aiming at combining the properties of DRAM (low latency, byte-addressability) with those of storage (persistence, capacity, price). The idea of NVM goes back to the sixties when bubble memory was discussed. Since then, several memory technologies have been proposed, among them Re-RAM, STT-RAM, and PCM – see [1, 2] for surveys. However, only Intel together with Micron have shipped NVM products sitting in existing DRAM slots: Intel Optane DCPMM based on the 3D XPoint technology. Among the the major advantages of its Byte-addressability over SSDs are that it

- allows to use identical data structures both for persistent and transient data and, therefore,
- eliminates the need to transfer data between persistent storage and memory.

For this purpose, NVM permits direct access (in terms of cache lines) to data via standard CPU instructions such as `STORE` and `LOAD`.

Intel Optane DCPMM supports two operating modes: the *memory mode* as a large, but volatile memory pool where DRAM can act as a cache layer on top of NVM, and the *app direct mode* where NVM is used as persistent memory. Operating systems like Linux and Windows on the most recent Intel platforms integrate Optane DCPMM via Direct Access (DAX) interface, i.e., persistent memory is memory-mapped into the address space and, thus, allows to load/store from/to memory directly.

Special cache line flushing instructions are used to guarantee that store operations are persistent because the path to the *persistence domain*, where a store to persistent memory becomes durable, consists of volatile layers like the CPU caches. `CLFLUSHOPT` is an optimized form of `CLFLUSH`. `CLWB` (cache line write back) is similar to `CLFLUSHOPT` but does not evict the cache line after flushing. These are followed by memory barrier instructions like `SFENCE` to enforce ordering and ensure the cache lines reach the *persistence domain*.

Recently, Intel has announced and shipped the second generation 200 series with increased bandwidth and Enhanced Asynchronous DRAM Refresh (eADR) support. eADR extends ADR to include CPU caches in the *persistence domain*, alongside the persistent memory and the memory controller's write pending queues. This further makes NVM programming easier and eliminates cache line flushes, thereby enhancing performance.

On top of this, additional APIs and development kits such as Intel's PMDK[1] simplify the software development [3].

Over the last few years, researchers have analyzed and benchmarked Intel's NVM Optane technology. The main findings are [4, 5]:

1. Asymmetry between load and store latency.
2. Asymmetry between load and store bandwidth.
3. Sequential IO faster than random IO.
4. Access granularity based on an internal 256-byte buffer.
5. Load bandwidth scale with thread count while store bandwidth does not.
6. Higher latency and lower bandwidth compared to DRAM.

In the database context, main research fields and use cases are:

- instant recovery and logging, e.g. write-behind logging [6], log-free recovery [7] and query recovery [8, 9].
- NVM-optimized data structures including hash tables [10], radix trees [11, 12] and B^+-tree variants [13].
- I/O primitives [14], parallel programming models [15], efficient algorithms [16] etc.

However, NVM is not (yet) the promised game changer for several reasons:

- Access latency is still higher than that of DRAM.
- Despite the availability of PMDK, NVM programming is still challenging.
- Finally, the still high costs per GB have hindered the wide adoption.

Overall, NVM has promising prospects as yet another tier of modern memory and storage hierarchies, as it opens up unprecedented opportunities for database systems on future hardware.

References

1 Margo I. Seltzer, Virendra J. Marathe, and Steve Byan. An NVM carol: Visions of NVM past, present, and future. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 15–23. IEEE Computer Society, 2018.

2 Haikun Liu, Di Chen, Hai Jin, Xiaofei Liao, Binsheng He, Kan Hu, and Yu Zhang. A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions. *J. Comput. Sci. Technol.*, 36(1):4–32, 2021.

3 Steve Scargall. *Programming Persistent Memory.* Apress, Berkeley, CA, 2020.

4 Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.*, 14(4):626–639, 2020.

5 Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.

[1] `https://pmem.io`

6 Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, 2016.

7 Gang Liu, Leying Chen, and Shimin Chen. Zen: a high-throughput log-free OLTP engine for non-volatile main memory. *Proc. VLDB Endow.*, 14(5):835–848, 2021.

8 Soklong Lim, Tyler Coy, Zaixin Lu, Bin Ren, and Xuechen Zhang. Nvgraph: Enforcing crash consistency of evolving network analytics in NVMM systems. *IEEE Trans. Parallel Distributed Syst.*, 31(6):1255–1269, 2020.

9 Alexander Baumstark, Philipp Götze, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Instant graph query recovery on persistent memory. In Danica Porobic and Spyros Blanas, editors, *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, pages 10:1–10:4. ACM, 2021.

10 Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent memory hash indexes: An experimental evaluation. *Proc. VLDB Endow.*, 14(5):785–798, 2021.

11 Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: write optimal radix tree for persistent memory storage systems. In Geoff Kuenning and Carl A. Waldspurger, editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 – March 2, 2017*, pages 257–270. USENIX Association, 2017.

12 Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: range-query optimized persistent ART. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 1–16. USENIX Association, 2021.

13 Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proc. VLDB Endow.*, 13(4):574–587, 2019.

14 Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. In Thomas Neumann and Ken Salem, editors, *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, pages 12:1–12:7. ACM, 2019.

15 Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 247–258. ACM, 2018.

16 Pedro Ramalhete, Andreia Correia, and Pascal Felber. Efficient algorithms for persistent transactional memory. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 1–15. ACM, 2021.

Participants

- Gustavo Alonso
ETH Zürich, CH
- Alexander Baumstark
TU Ilmenau, DE
- Carsten Binnig
TU Darmstadt, DE
- André Brinkmann
Universität Mainz, DE
participant Christian Dietrich
TU Hamburg-Harburg, DE
- Muhammad Attahir Jibril
TU Ilmenau, DE
- Alfons Kemper
TU München, DE
- Viktor Leis
Universität Erlangen-Nürnberg, DE
- Alberto Lerner
University of Fribourg, CH
- Ulrich Carsten Meyer
Goethe-Universität – Frankfurt am Main, DE
- Thomas Neumann
TU München, DE
- Ismail Oukid
Snowflake – Berlin, DE
- Marcus Paradies
German Aerospace Center – Jena, DE
- Kai-Uwe Sattler
TU Ilmenau, DE
- Jens Teubner
TU Dortmund, DE
- Alexander van Renen
Universität Erlangen-Nürnberg, DE

Remote Participants

- Marcos K. Aguilera
VMware – Palo Alto, US
- Raja Appuswamy
EURECOM – Biot, FR
- Manos Athanassoulis
Boston University, US
- Alexander Böhm
SAP SE – Walldorf, DE
- Peter A. Boncz
CWI – Amsterdam, NL
- Mark Callaghan
Rockset – Bend, US
- Khuzaima Daudjee
University of Waterloo, CA
- Jana Giceva
TU München, DE
- Goetz Graefe
Google – Madison, US
- Gabriel Haas
Universität Erlangen-Nürnberg, DE
- Stratos Idreos
Harvard University – Cambridge, US
- Wolfgang Lehner
TU Dresden, DE
- Danica Porobic
Oracle Labs – Redwood Shores, US
- Ken Salem
University of Waterloo, CA
- Wolfgang Schröder-Preikschat
Universität Erlangen-Nürnberg, DE
- Margo Seltzer
University of British Columbia – Vancouver, CA
- Tianzheng Wang
Simon Fraser University – Burnaby, CA
- William Wang
ARM Ltd. – Cambridge, GB