CXL Memory Performance for In-Memory Data Processing

Marcel Weisgut
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
marcel.weisgut@hpi.de

Daniel Ritter
SAP
Walldorf, Germany
daniel.ritter@sap.com

Pınar Tözün
IT University of Copenhagen
Copenhagen, Denmark
pito@itu.dk

Lawrence Benson
Technische Universität Munich
Munich, Germany
lawrence.benson@tum.de

Tilmann Rabl
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
tilmann.rabl@hpi.de

ABSTRACT

The Compute Express Link (CXL) standard enables new forms of memory management and access across devices and servers. Based on PCIe, it enables cache-coherent access to remote memory. This widens the design space for database systems by expanding the available memory beyond memory local to the CPU. Efficiently utilizing CXL-attached memory requires conscious decisions by data systems about data placement and management. In this paper, we provide an in-depth analysis of database operation performance with data interleaved across multiple CXL memory devices. We experimentally evaluate the memory access performance for basic access patterns, the performance impact of placing data across multiple CXL memory devices for in-memory column scans and in-memory B+tree operations, and the performance impact of placing data in CXL memory for an in-memory database system when running the analytical TPC-H workload. Our experiments show that access to CXL-attached memory does not have to penalize performance over local access, but careful workload-aware data management is required. Our TPC-H evaluation shows that placing table columns based on access frequencies allows storing over 80% of the table data in CXL memory with a performance of 85% of a local-memory-only solution.

PVLDB Reference Format:
Marcel Weisgut, Daniel Ritter, Pınar Tözün, Lawrence Benson, and Tilmann Rabl. CXL Memory Performance for In-Memory Data Processing . PVLDB, 18(9): 3119 - 3133, 2025.
doi:10.14778/3746405.3746432

PVLDB Artifact Availability:
The source code have been made available at https://github.com/hpides/cxlbench/tree/paper/vldb25 (microbenchmarks) and https://github.com/hyrise/hyrise/tree/paper/vldb25 (Hyrise).

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746432

1 INTRODUCTION

Database systems have widely adopted the separation of compute and storage resources to benefit from elastic scaling of individual hardware resources on demand [22, 58, 66, 82]. The individual scaling capabilities avoid resource over-provisioning, which results in significant total cost of ownership (TCO) reduction [12, 58]. In contrast, compute and memory resources are usually tightly coupled, leading to stranded memory [47, 48, 53, 75].

Both the database research community and industry propose memory-disaggregated database systems to separate memory from compute resources [2, 3, 28, 41, 44, 47, 54, 75–79, 85, 86]. These efforts are mainly based on fast remote direct memory access (RDMA) network technology. The new Compute Express Link (CXL) technology is an alternative for implementing memory-disaggregated database systems [2, 3, 14, 28, 32, 44, 47].

CXL allows adding memory to a server's unified memory address space using PCI Express (PCIe) [16]. The additional memory and memory controller(s) are located on CXL devices. Traditional CPU-local memory is attached via the double data rate (DDR) interface and in the form of DRAM dual inline memory modules (DIMMs). CXL memory is — like CPU-local memory — cache coherent [68].

Memory expansion via CXL-attached memory devices has received attention in research and industry [2, 3, 29, 31, 44, 62]. A reason for using CXL memory expansion is to reduce the cost of a server's memory hardware. As main memory has become a major cost driver of server hardware [14, 48, 56, 80], reusing DIMMs of decommissioned servers can significantly reduce the TCO of a server [8, 14, 74, 87]. Memory attached via CXL does not need to match the CPU's DDR version and can be of any media type [68]. In a pooled memory scenario, where CXL memory is connected to multiple servers, the TCO can be reduced as administrators can populate servers with smaller CPU-local DRAM capacities, while larger capacities of cheaper memory can be attached via CXL [14, 53].

Incorporating CXL memory into database systems invalidates the assumption that memory-resident data is close to the CPU [47]. It requires extensive experimentation to identify what part of a database engine can be stored in CXL memory and what part should reside in CPU memory [47].

Our analysis provides insights into the use of CXL memory expansion devices. Our work is an in-depth performance study

investigating the impact of placing data across multiple real CXL memory devices for database operations.

Contributions. Our main contributions are the following:

(1) A detailed CXL memory access performance evaluation on a server with four CXL memory devices.
(2) A performance analysis of vectorized column scans and B+tree operations with data interleaved across up to four CXL devices.
(3) A performance evaluation of two data placement strategies using the TPC-H workload on an in-memory database system.

The rest of this paper is structured as follows: In Section 2, we discuss relevant background on memory interconnects and CXL. Section 3 gives an overview of our benchmarking framework. We present results of our microbenchmarks in Section 4, the result of different isolated DBMS operations in Section 5, and end-to-end experiments using the in-memory database management system Hyrise in Section 6. We analyze the economic viability of CXL setups in Section 7 and summarize our insights in Section 8. We survey related work in Section 9, before concluding in Section 10.

2 BACKGROUND

We introduce technical knowledge relevant to the remainder of this paper, including memory interconnects and CXL.

2.1 Memory Interconnects

Table 1 details the memory interconnects of the CPU we use in our evaluation. We classify them as *Socket-Local* (CPU-to-memory at one socket), *Inter-Socket* (CPU-to-CPU), and *Inter-Device* (CPU-to-device). The channel width determines how much data passes through an interconnect channel with a single transfer. An interconnect's theoretical maximum bandwidth is:

$$\text{Bandwidth} = \#\,\text{Channels} \times \text{Transfer rate} \times \text{Channel width}\ .$$

Socket-Local. Memory local to a CPU is usually attached as DDR DRAM DIMMs. DDR interfaces have a width of 64 bits [49]. The latest generation supported by CPUs is DDR5, with data rates of up to 8400 MT/s [70]. A modern CPU deploys multiple memory channels. Current Intel Emerald Rapids CPUs support eight channels, leading to a theoretical peak bandwidth of ~333 GB/s with a maximum supported memory speed of 5200 MT/s.

Inter-Socket. If a core accesses memory whose memory controller is located on a remote socket, data needs to be transferred via an inter-socket interconnect. Ultra Path Interconnect (UPI) is Intel's cache-coherent interconnect for socket-to-socket communication between CPUs [36]. UPI's transfer rate and number of ports per CPU can vary across CPU generations and models [34]. Current 5th Gen Intel Xeon Scalable CPUs (Emerald Rapids) have up to four UPI ports with a transfer rate of up to 20 GT/s [35], leading to a cumulative theoretical bandwidth of 160 GB/s.

Inter-Device. Several cache-coherent interconnects have been specified, such as GenZ [30], OpenCAPI [71], Cache Coherent Interconnect for Accelerators (CCIX) [15], AMD's Infinity Fabric CPU-GPU links [1, 65], NVIDIA's NVLink-C2C [18], and CXL [16]. The consortiums of GenZ, OpenCAPI, and CCIX have joined forces with the CXL consortium. AMD's and NVIDIA's inter-device links focus on connecting CPUs and GPUs and their memory rather than connecting memory expansion devices to server CPUs.

Table 1: Interconnect characteristics per link (channel) of the CPU and CXL setup used in our evaluation.

Type	Width [Bits]	Transfer rate [GT/s]	Data rate [GB/s]
DDR	64	5.2	41.6
UPI	16 [64]	20 [37]	40
CXL	16 [16]	32	64

2.2 Compute Express Link (CXL)

CXL [16] is a standard for interconnects between CPUs and devices. It allows them to cache data stored in each other's memory.

Protocols. CXL specifies three protocols that CPUs and devices communicate over the PCIe physical layer (PCIe PHY). *CXL.io* is the mandatory base protocol containing PCIe transactions. It is used for, e.g., device discovery, status reporting, address translation, and direct memory access. Devices can cache data stored in CPU memory via *CXL.cache*. *CXL.mem* allows CPUs to access and cache data stored in CXL device memory.

Device Types. The standard specifies three device types. Type 1 devices (supporting CXL.cache) can access and cache data stored in the CPU's local memory. Type 1 devices do not have cache-coherent memory exposed to the CPU. One example is a smart network interface card with coherent access to host (CPU) memory [68]. Type 2 devices (supporting CXL.cache and CXL.mem) have on-device memory exposed to a CPU and full coherent access both to their own memory and CPU memory. A CPU can access and cache device memory. Examples are accelerators with attached memory, such as GPUs and FPGAs [68]. Type 3 devices (supporting CXL.mem) have memory attached and allow CPUs to coherently cache data stored in the device memory. Type 3 devices cannot request data via CXL.cache and are used for memory expansion.

Memory Access Anatomy. Memory of Type 2 and Type 3 devices exposed to the host system is called host-managed device memory (HDM). A CPU's caching agent interacts with HDM via CXL.mem. This memory is integrated into the coherence domain of the host. Data transfers via CXL.mem occur at 64 B cache line granularity. If the HDM is only accessible by the host, the CPU manages the coherence exclusively. Figure 1 illustrates the integration of CXL memory into the CPU's coherence domain. CPU-local memory is directly connected to the CPU's integrated memory controller via DDR, and CXL memory is located in one or more CXL Type 3 devices. The CPU's home agent manages cache coherence and resolves conflicts across other caching agents, such as local cores,

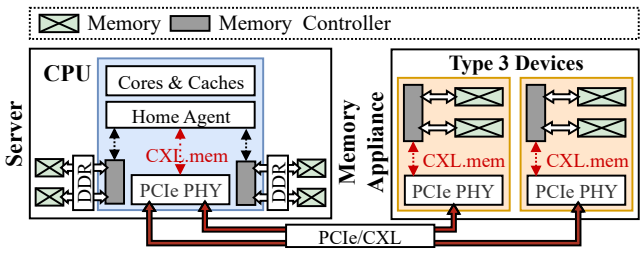


Figure 1: Memory expansion with multiple CXL devices.

other CPU sockets, and CXL Type 1 or 2 devices [67]. When the CPU performs memory access to the CXL device, its home agent communicates via CXL.mem over the PCIe PHY.

Memory Allocation. CXL memory can be configured as a memory-only NUMA node [2, 72]. It allows utilizing NUMA-related system calls to interact with the memory, such as `mbind` for setting a memory policy for a given virtual memory region and `move_pages` for moving operating system (OS) pages between nodes [62].

3 MICROBENCHMARK FRAMEWORK

We introduce CXL-Bench, a framework for benchmarking access to heterogeneous memory. It is a successor of the persistent memory benchmark framework PerMA-Bench [7]. CXL-Bench allows benchmarking access to memory exposed as a non-uniform memory access (NUMA) node, including CXL memory. CXL-Bench supports basic memory access patterns, including sequential and random reads and writes, and chains of custom operations. Custom operations allow users to model complex access patterns for database workloads. In this work, we use CXL-Bench for quantifying CXL memory access characteristics for basic access patterns. CXL-Bench allows users to specify different access sizes. It performs 8 B accesses as scalar loads and stores, 16 B and 32 B accesses as vector load and store instructions with corresponding vector sizes, and 64 B and larger accesses as 64 B vector loads and stores.

Benchmark Workflow. For each benchmark task, a number of threads perform memory accesses to a memory region based on a user-defined configuration. Parallel benchmark tasks allow simulating memory accesses of parallel database tasks. One parallel task can benefit or penalize the other by loading or evicting data into or from the cache that the other task will access. For each benchmark run, CXL-Bench prepares the memory regions to be accessed, creates batches of access operations for the pre-defined set of threads, performs the access operations, verifies memory page locations to ensure that no pages were moved during the access execution, and generates the results containing throughput and latency metrics.

Memory Region Preparation. The memory preparation step includes allocating virtual memory, binding virtual memory regions to user-defined NUMA nodes, and backing a memory region's pages by physical memory (by writing to each page, which forces the OS to allocate physical memory). Pages can be allocated in any kind of memory that is configured as a NUMA node. CXL-Bench allocates pages in memory either in a *non-partitioned* or a *partitioned* mode. The non-partitioned mode uses the entire memory region and pins its pages to the user-defined NUMA nodes. The partitioned mode allows users to split the memory region into two partitions with different user-defined sizes relative to the region's total size. It then pins the partitions to different NUMA nodes. In both modes, pages of a memory (sub-)region are pinned to the corresponding NUMA nodes via the `mbind` system call with Linux's interleaved allocation policy. If the corresponding task's memory operations include read operations, the memory region is filled with data in advance.

Task Execution. CXL-Bench generates a user-defined number of worker threads executing the memory access operations. CXL-Bench pins a thread to a set of cores (via the `pthread_setaffinity_np` GNU C library function). Users can specify per task to which cores the thread pool is pinned. This allows thread pools of parallel tasks to be pinned to different cores. Worker threads continuously fetch memory access batches from a shared queue and execute the corresponding memory access operations. This represents a common execution model where workers operate on small work packages [7]. By default, the total number of accessed bytes per batch is 64 MiB. Such batches are short-running and, thus, avoid the skew of large, long-running batches [7].

4 CXL MEMORY ACCESS PERFORMANCE

Using CXL-Bench, we first investigate the memory access behavior of CXL memory on a server with four CXL memory devices. After introducing the evaluation server (Section 4.1), we quantify the maximum throughput (Section 4.2) and the throughput scaling with multiple devices (Section 4.3) for sequential and random reads and writes. The results serve as upper bounds for the database operations in the remaining sections. We then investigate the memory access latency (Section 4.4) and the performance impact of placing an increasing number of pages in CXL memory (Section 4.5).

4.1 Hardware Setup

Table 2 details the evaluation server. Figure 2 shows the CPU memory and CXL device setup. The server is attached to four *Seagate Composable Memory Appliance (CMA) Blade* prototypes [23, 55]. Each CMA blade is an FPGA-based memory expansion solution. The FPGA-based design is not performance-optimized for production. A CMA blade supports PCIe Gen5 x16 CXL 1.1 specification connectivity. Each blade has four DDR4 channels with two DIMMs connected per channel. We refer to a blade as a CXL device.

4.2 Maximum Sustained Throughput

Setup. CXL-Bench performs either memory reads or writes with different thread counts and memory access sizes. The access patterns

Table 2: Specifications of the evaluation server.

Server	Supermicro SYS-741GE-TNRT
CPUs	2× Intel 5th Gen Xeon Scalable Gold 6542Y with 24 cores
Caches	L1i: 32 KiB, L1d: 48 KiB, L2: 2 MiB, L3: 60 MiB
Memory	8× 32 GB DDR5 with a speed of 4800 MT/s
OS	Ubuntu 24.04, Kernel 6.13

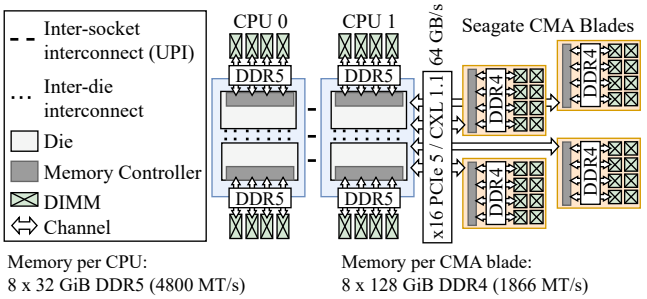


Figure 2: CPU memory and CXL device setup.

are either sequential or uniform random. The executed benchmark configurations start with one and four threads and incrementally increase in steps of four. We use powers of two for the access sizes, starting with 8 B and incrementally increasing to 8 KiB. For each configuration, CXL-Bench performs memory accesses for ten seconds to a virtual memory region of 7 GiB.

Results. Figures 3e to 3h show the throughput when accessing one CXL device's memory. Figures 3a to 3d show the sustained memory throughput for CPU memory as a reference for comparison. Sequential reads achieve a maximum of ~40 GB/s. Sequential writes peak at 18 GB/s with 24 and 28 threads and an access size of 8 B. With an increasing number of threads, the sequential read throughput slightly decreases, stagnating at 95%. We observe a similar behavior with random reads: while both CPU and CXL memory require an access size of at least 4 KiB to achieve higher throughput, the throughput for CXL memory decreases with an increasing number of threads, stagnating at 93%.

The maximum write throughput is between 35% and 45% of the maximum read throughput. This matches the relative write throughput to CPU memory compared to read throughput.

The heatmaps for random accesses show a range of access sizes for which the throughput is significantly lower than with lower and higher access sizes. With CPU memory, the throughput decreases between 512 B and 2 KiB for random reads (see Figure 3b) and at 1 KiB for random writes (see Figure 3d). We measure this throughput drop for random access patterns to CPU memory on additional servers with AMD Genoa, AMD Rome, Intel Sapphire Rapids, Intel Ice Lake, and Intel Cascade Lake CPUs in the range from 512 B to 8 KiB (not shown). In these cases, the hardware prefetcher causes poor throughput for a range of access sizes. When disabling the hardware prefetcher, the performance does not decrease. The dominant access sizes causing the throughput decrease are 1 KiB and 2 KiB for the evaluated CPUs.

4.3 Scaling-Up Throughput With Devices

Setup. We evaluate the throughput with one, two, three, and four CXL devices. We use CXL-Bench with a memory region of 7 GiB pinned to a set of the CXL devices' NUMA nodes. This results in a round-robin interleaving of pages with two or more devices.

Results. Figure 4 shows the memory access throughput for different numbers of CXL devices, threads, and access sizes. The throughput for random access varies depending on the access size. The throughput converges to approximately the same level for sequential accesses. Sequential writes show the lowest throughput variance. Interleaving pages round-robin across multiple devices increases each workload's throughput. For sequential 4 KiB reads with 24 threads (matching the core count), the throughput achieves 1.9×, 2.3×, and 2.5× with two, three, and four devices compared to one device (with 38 GB/s). For sequential 4 KiB writes with 24 threads, the throughput achieves 2×, 3×, and 3.7× compared to one device (with 13.5 GB/s). The number of contiguous, sequentially accessed cache lines is higher with larger access sizes. This allows the CPU to utilize the hardware prefetcher, which benefits the throughput.

4.4 Latency

Database operations, such as joins or aggregates [63], as well as transaction processing [50] are typically memory latency-bound. With CXL memory, the interconnects and controllers involved when accessing memory differ from DDR-attached CPU memory, hence impacting data access latency.

Setup. We quantify idle latency percentiles for 8-B reads and writes to CPU, remote CPU, and CXL memory. For reads, we measure the latency of the load instruction followed by a memory fence (`mfence`) with sequential and random access patterns. For writes, we first load the cache line into the cache and then measure the store instruction followed by a cache line write back (`clwb`) instruction and a memory fence (similar to previous work on benchmarking

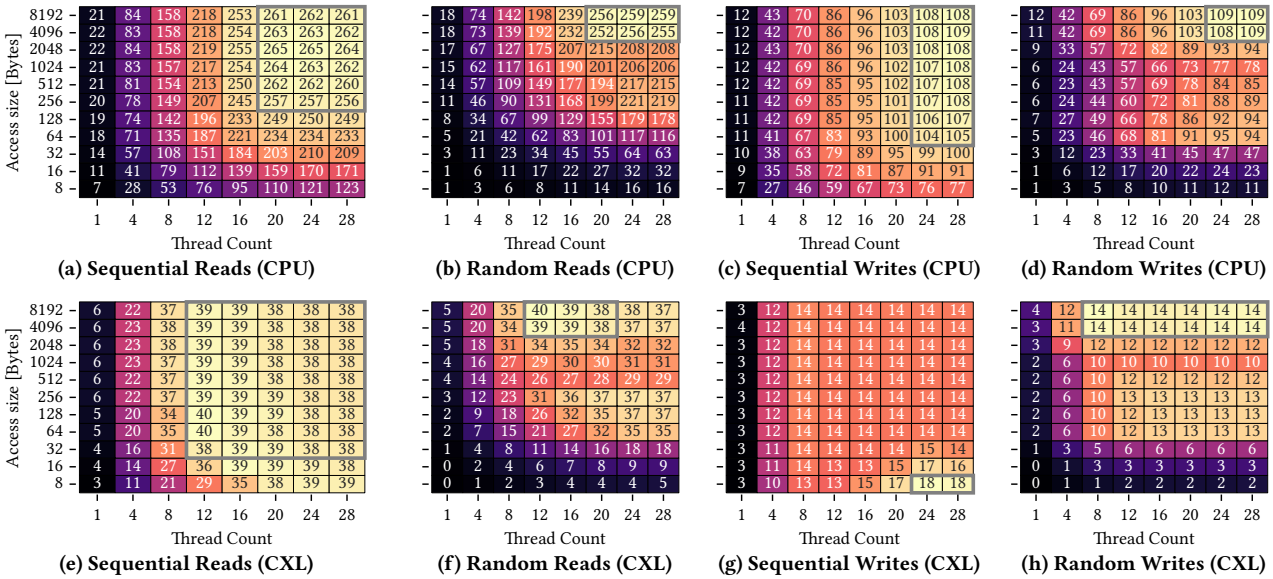


Figure 3: CPU and CXL memory throughput [GB/s]. We use one CXL device for CXL memory.

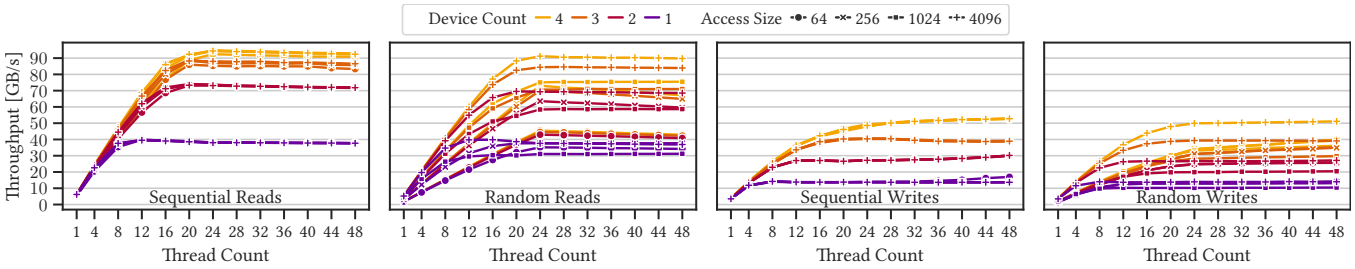


Figure 4: Accesses throughput to memory of multiple CXL devices.

memory accesses [83]). We perform the `clwb` to ensure that the target cache line is written back to memory. We increase the address to be accessed by 8-B after each access for a sequential pattern. For random accesses, each accessed cache line stores the position of the next memory access. This results in a chain of dependent accesses and ensures that a subsequent access operation can only be performed after the current read is finished. We run 100 M accesses and measure the latency in nanoseconds (ns) for every 10 000th memory access. A single thread executes the respective access operation to an 8 GiB memory region.

Results. Figure 5 shows the latency across different percentiles for 8 B accesses to CPU, remote CPU, and CXL memory. We round absolute latencies to the nearest multiple of five ns. For sequential reads, 90% of the accesses to all types of memory finish within 40 ns. For random reads, compared to the median access latency of CPU memory, the 99.9th percentile latencies for CPU, remote CPU, and CXL memory are about 160 ns, 280 ns, and 785 ns higher.

When writing to a cache line that already resides in the cache and flushing the modified cache line shows median latencies of 125 ns, 240 ns, and 345 ns. Compared to the median access latency of CPU memory, the 99.9th percentile latencies for CPU, remote CPU, and CXL memory are about 180 ns, 300 ns, and 410 ns higher.

On average, random reads show the highest CXL memory access latency of 520 ns, followed by a write latency of 350 ns. Sequential reads show the lowest average access latency of 65 ns, which is only slightly higher than 45 ns for CPU memory. We conclude that, despite the higher access latency of CXL memory, the hardware prefetcher can hide the access latency for 8-B sequential accesses.

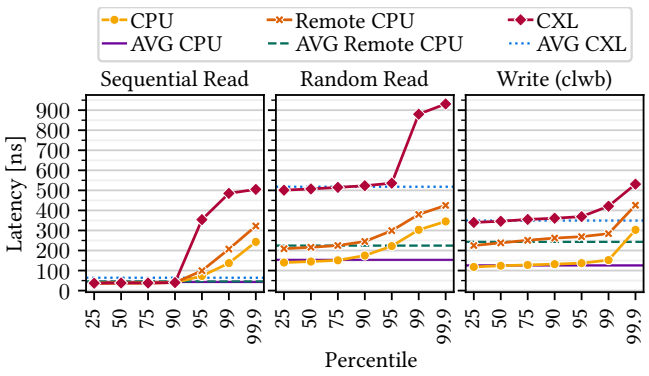


Figure 5: Latency of reads and cached writes with flushes.

4.5 Cost of CXL Memory Accesses

Database management systems (DBMSs) often need to serve workloads with datasets larger than CPU memory. Such DBMSs use a memory buffer and flush buffer pages to disk when the buffer is full [27, 45]. The resulting storage I/O is a key performance bottleneck [27]. CXL memory attached to a server system expands the system's total memory capacity with CXL memory as an additional memory tier [62]. In this experiment, we investigate the impact of storing data in CXL memory on access performance.

Setup. CXL-Bench runs a workload with a virtual memory region of 7 GiB and one pool of worker threads performing the memory accesses. We partition the memory region and pin one partition to CPU memory and the other to CXL memory. The partition sizes are set according to the share of pages in CPU or CXL memory. The shares of pages vary from 100% in CPU and 0% in CXL memory to 0% in CPU and 100% in CXL memory in steps of 5%. A workload performs either read or write operations with uniform random access patterns. We vary the number of threads used to execute the memory accesses. A workload runs for 10 seconds.

Results. Figure 6 shows the results with access sizes of 64 B and 4 KiB. The overall throughput is higher the more threads are used. The throughput decrease with an increasing share of data stored in CXL memory varies significantly depending on the access sizes and access operations (i.e., reads or writes). A larger share of sequential accesses with an access size of 4 KiB allows the CPU to prefetch cache lines, which benefits throughput. The figure shows that the decrease in throughput flattens out the more threads access the data. The configuration with 4 KiB accesses and 10% of data in CXL memory is a notable example demonstrating this behavior. While the performance decreases to a marginal extent with 12 threads, we do not observe a decrease with 16 threads. Using more threads increases the number of requests sent to the CPU memory controllers, which are a common point of contention for sequential accesses [40]. This contention is amplified by the sequential access pattern, triggering the CPU's prefetcher to send load requests to the controllers. In this microbenchmark, the contention at the CPU memory controllers is the highest when all data is stored in CPU memory. In the CPU memory-only case, the memory controller of the CXL device (see Figure 2) is underutilized. Increasing the share of data located in CXL memory moves the corresponding share of memory requests to the CXL device's memory controller. This, in turn, releases the contention at CPU memory controllers.

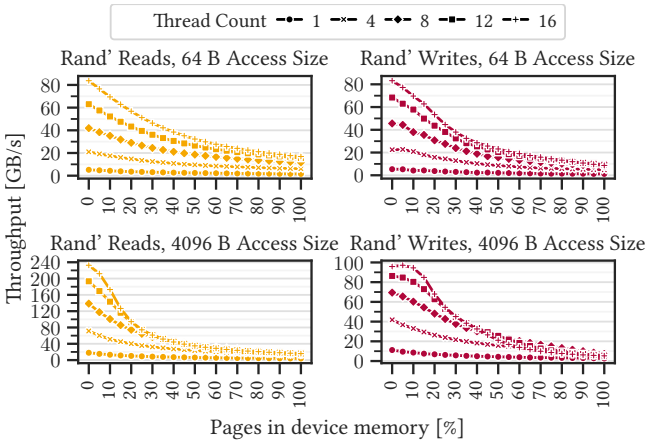


Figure 6: Impact of storing pages in CXL device memory.

5 INDIVIDUAL DATABASE OPERATIONS

Database operations include aspects such as value comparisons, predicate evaluation, and index operations. We evaluate the impact of storing data in CXL memory with filtering vectorized column scans and B+tree index operations. We particularly investigate the performance impact of placing data in multiple CXL devices. We characterize the workloads' memory bottleneck into latency- and bandwidth-bound using Intel's *VTune Profiler* and the top-down microarchitecture analysis (TMA) method [4, 59, 69, 84].

5.1 Top-Down Microarchitecture Analysis

Micro-Operations and Pipeline Slots. The instruction pipeline of modern out-of-order CPU cores contains two major components: the front-end and the back-end. The front-end fetches program instructions, decodes them into micro-operations (μOps), and issues them to the back-end. The back-end executes the μOps on available execution units. TMA abstracts the hardware resources required to execute a μOp into *pipeline slots*, assuming four slots being available per cycle and core.

Classification Categories. TMA classifies each pipeline slot as *retiring*, *bad speculation*, *front-end-bound*, or *back-end-bound*. In each cycle, a slot can be empty or filled with a μOp. If it is filled, the μOp either retires or does non-useful work due to bad speculation. An empty slot is caused by a front-end or back-end stall. We focus on workloads that are memory-bound — a sub-category of back-end-bound. TMA breaks down memory-bound into *store-bound*, *L1-bound*, *L2-bound*, *L3-bound*, and *external memory-bound* (also *DRAM-bound*). DRAM-bound consists of the sub-categories (memory) *bandwidth* and (memory) *latency*. The TMA reports the share of the total pipeline slots corresponding to each category.

Normalized Metrics. Overlaps in stall time can be counted double when measuring the memory-bound and DRAM-bound sub-categories. In this case, the combined share of the sub-categories can exceed the parent's value. Following previous work [4, 69], we normalize the sub-categories' values to match the parent-level:

$$\text{DRAM}_{norm} = \frac{\text{DRAM} \times \text{memory bound}}{\text{L1} + \text{L2} + \text{L3} + \text{DRAM} + \text{Store}} \ ,$$

$$\text{S}_{norm} = \frac{\text{S} \times \text{DRAM}_{norm}}{\text{Bandwidth} + \text{Latency}} \ ,$$

where *S* represents the sub-category *bandwidth* or *latency*.

5.2 In-Memory Scan

We evaluate a vectorized filtering integer scan with 4-B unsigned integer values. Each scan processes a separate column, storing 512 MiB of data (i.e., about 134 M values). We use an AVX-512 scan implementation [6] and adapt it for different data placements. The implementation writes (4-B unsigned integer) offsets as tuple identifiers (TIDs) for tuples matching the filter predicate to a memory region. This is common in several database systems [43, 60, 88].

Data Placement. We allocate a memory region for each column and each TID list. We bind a region to the target NUMA nodes via the `mbind` system call. We differentiate between the location of the columns and the TIDs. The column placement determines what memory type the CPU reads data from, while the TID list placement determines the memory type the CPU writes data to. We evaluate different data placement configurations. A configuration either stores columns and TIDs in CPU memory (*CPU*), in CXL memory (*CXL*), or columns in CXL memory while TIDs are written to CPU memory (*ColumnsCXL*). We further store data placed in CXL memory on one or four CXL devices (indicated with the suffix *1* or *4*). The implementation stores only heap memory allocations for these two types of data either in CPU or CXL memory. Other data structures remain in stack memory, which is CPU memory.

Workload. We perform scan executions with multiple numbers of threads and different selectivities. Each thread scans an individual column with one less-than predicate. The integer values in the columns are uniformly distributed. The filter selectivity steers the read and write ratios. With a selectivity of 100%, each 4-B value qualifies, and the scan operator writes the corresponding 4-B TID to the result list. We measure the time required to finish all column scans. We calculate the throughput in values per second, given the total number of scanned values and the duration. We multiply this metric by the value size of 4 B and report the scan throughput in gigabytes per second. We execute each benchmark configuration four times and report the average.

Results. Figure 7 shows that the throughput across all configurations increases with lower selectivity. This is expected as lower selectivity results in fewer writes. When all data is stored in CPU memory, the scan processes about 260 GB/s with a selectivity of

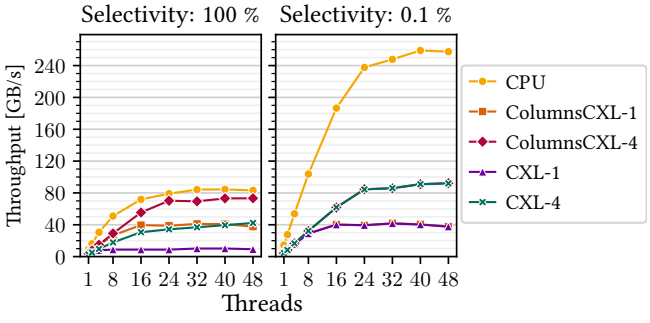


Figure 7: Filtering integer scan performance.

0.1%, which is close to the maximum sequential read-only throughput measured with CXL-Bench (see Figure 3a). With a selectivity of 100%, the throughput plateaus at about 84 GB/s. This is close to the maximum sequential write-only throughput quantified in Section 4.2 (see Figure 3c). In the write microbenchmark, we minimize control flow instructions and keep the fixed data to be written in a vector register without loading new data into the register. This does not apply to a database scan where the data to be written needs to be loaded from memory first and where the read values need to be compared with the filter predicate. While the CPU can read column data faster, the write throughput limits the scan progress since it writes a 4-B TID for each read 4-B value.

The scan achieves higher throughput with more data in CPU memory. For a given number of CXL devices, the scan throughput is equal for both *ColumnsCXL* and *CXL* configurations with a selectivity of 0.1% since the number of writes is marginal. With a selectivity of 0.1%, the throughput reaches about 40 GB/s with one device. This aligns with the maximum throughput measured with the sequential read microbenchmark (see Figure 3e). The throughput reaches 92 GB/s with four devices and 48 threads, which is close to the maximum of 95 GB/s measured with the microbenchmark (see Figure 4). Unlike the low-selectivity scans, which perform only a few writes, the placement decision has a significant performance impact on high-selectivity scans. With a selectivity of 100% and one CXL device, storing all data in CXL memory achieves 10 GB/s. Storing only the columns in CXL memory while writing the TIDs to CPU memory increases the throughput by 4×, reaching the maximum read throughput of 40 GB/s for a single device.

Increasing the number of CXL memory devices across which the columns are interleaved increases the scan throughput. When storing both columns and TIDs in CXL memory, the 100%-selectivity scan achieves 42 GB/s with four devices and 48 threads. This is an increase of 4× compared to a single device. The achieved 42 GB/s corresponds to 80% of the write throughput measured in the write microbenchmark with four devices (see Figure 4). While the microbenchmark achieves higher throughput due to its write-optimized design, the scaling with four devices is similar (i.e., 3.7×).

With only columns in CXL memory, the throughput for a selectivity of 100% with four devices increases by 1.8× with 73 GB/s compared to 40 GB/s with one device. With a selectivity of 0.1%, the throughput improves by 2.3× from 40 GB/s to 92 GB/s.

5.3 Hybrid Column Placement

We measure the throughput with a share of columns in CXL memory while TIDs are written to CPU memory.

Data Placement. We place a column entirely either in CPU or CXL memory. We run threads concurrently, each thread scanning an individual column. Unlike the previous experiment, we place a share of columns in CXL memory and the remaining columns in CPU memory. We vary the share of columns placed in CXL memory in steps of 10%, starting from 0% up to 100%.

Results. Figure 8 shows the resulting throughput for scan selectivities of 0.1% and 100% with 10, 20, and 40 threads. The throughput decreases with an increasing share of data in CXL memory in all configurations. The throughput decrease is steep for a selectivity of 0.1%, while the decrease is more gradual for a selectivity of 100%.

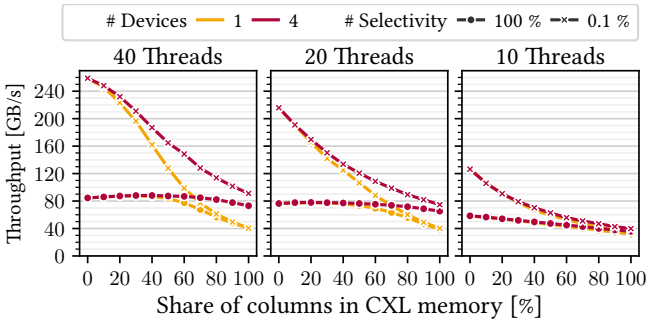


Figure 8: Scans with a share of columns in CXL memory.

Since the scan throughput is limited by the writes, the absolute throughput of the high-selectivity scan is significantly lower.

Four devices improve throughput compared to one device. We discuss the throughput with 40 threads in the following. We measure throughput with all data in CPU memory as the baseline. With a selectivity of 0.1%, the throughput achieves 16% of the baseline throughput with one device when all data is in CXL memory. With four devices, the throughput achieves 35%. This corresponds to a speedup of 2.2× with four devices compared to one device.

With a selectivity of 100%, the throughput achieves 48% of the baseline throughput with one device. With four devices, the throughput achieves 86%. This corresponds to a speedup of 1.8× with four devices compared to one device.

For the high-selectivity scan, interleaving a column's memory pages across multiple CXL devices improves throughput when data is stored in CXL memory. With four devices and 20 threads, 50% of the columns can be stored in CXL memory without a throughput decrease. A higher level of parallelism with 40 threads allows putting even 70% of the columns in CXL memory without decreasing the throughput. For in-memory database workloads, including column scans, the memory controllers are a common point of contention [40]. The contention increases the more threads utilize memory connected via the same set of memory controllers.

The random write microbenchmark with 4 KiB accesses in Section 4.5 increases the load at the memory controllers with additional threads and the vector store instructions. In this section's workload, the increased number of concurrently running scans increases both read and write requests to the memory controller. In both experiments, more data can be placed in CXL memory without a decrease in throughput when the contention at the controllers is high.

Bottleneck Analysis. To better reason about the impact of interleaving data across multiple CXL devices, we quantify the CPU's performance bottleneck while running the in-memory scan using the TMA method (see Section 5.1). Figure 9a (left) shows the performance breakdown for the scan with 40 threads and different data placements. More than 85% of the scan's μOps are backend-bound in all configurations, and more than 65% μOps are DRAM-bound. The breakdown into memory bandwidth-bound and memory latency-bound (Figure 9a, right) shows that the scan is by 37% memory bandwidth-bound with all data in CPU memory. The share of bandwidth-bound μOps is significantly higher with about 70% when all data is stored in CXL memory due to the lower bandwidth

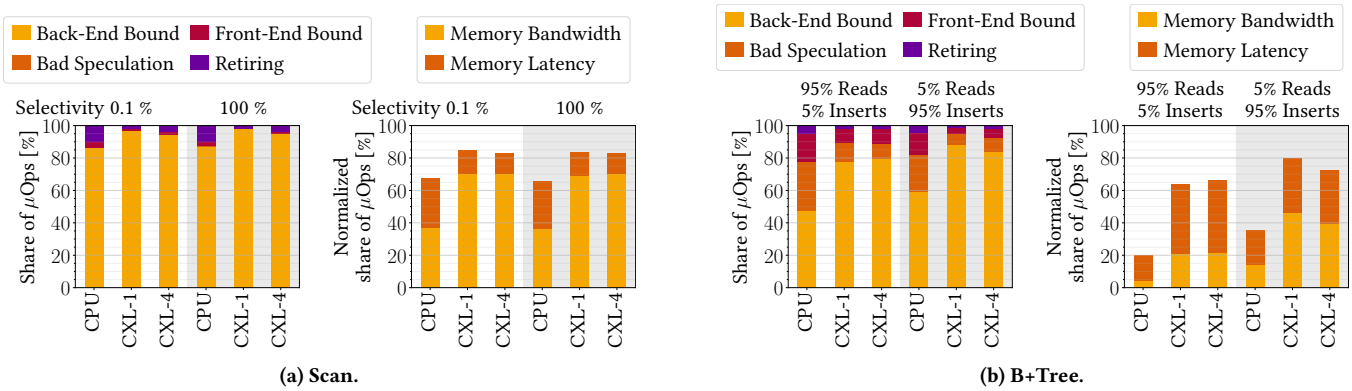


Figure 9: Performance breakdown with 40 threads and all data on one device (CXL-1), four devices (CXL-4), or CPU memory (CPU) with normalized bandwidth- and latency-bound shares.

of the CXL device(s). With this high degree of bandwidth limitation, adding more bandwidth with multiple CXL devices increases the operation's performance as Figure 8 shows.

5.4 In-Memory B+Tree

The B+tree is a ubiquitous tree-based index structure for DBMSs and key-value stores. We evaluate a state-of-the-art B+tree with optimistic lock coupling (BTreeOLC) [46] in CPU memory and CXL memory, using Mühlig et al.'s implementation [57].[1]

Data Placement. We quantify the performance with different data placements. All tree nodes are placed in either CPU memory (*CPU*) or CXL memory (*CXL*). We evaluate these placement configurations with one and four CXL devices. We allocate a consecutive memory region and bind it to the corresponding NUMA memory nodes. Tree node allocation requests receive pointers to chunks of that memory region with the requested size.

Workloads. We use the YCSB [17] benchmark to run transactional workloads on the BTreeOLC. We perform a read-heavy (95% reads, 5% inserts) and write-heavy (5% reads, 95% inserts) workload. Following previous work [57], each workload performs 100 M operations with a Zipfean request distribution on a tree initialized with 100 M records. We run the experiments with different thread counts. Each thread executes operations in batches of 500. The tree stores pairs with 8-B keys and 8-B values. Each node has a size of 1024 B.

Results. Figure 10 shows the resulting throughput in million operations per second. For a given number of CXL devices, storing all data in CPU memory yields the highest throughput. More threads increase the throughput across all placement configurations. 48 threads achieve the highest throughput of 94 M op/s and 56 M op/s for the read-heavy and write-heavy workloads.

Read-Heavy. The throughput measured with all data in CPU memory is the baseline. We discuss the throughput with 48 threads. Storing all nodes in CXL memory achieves 38% and 40% of the baseline throughput with one and four CXL devices. Using four devices instead of one increases the throughput by 6%.

Write-Heavy. All nodes in CXL memory achieves 25% and 39% of the baseline throughput with one and four CXL devices. Four devices increases the throughput by 57% compared to one device. Using multiple devices increases the throughput more for the write-heavy workload than for the read-heavy workload.

Bottleneck Analysis. Figure 9b (left) shows the performance breakdown for the two workloads with 40 threads and different data placement configurations. The workload is significantly more backend-bound with data in CXL memory (77% to 88%) than with all data in CPU memory (47% to 59%). The breakdown into memory bandwidth-bound and memory latency-bound (Figure 9b, right) shows different shares between the two workloads. Overall, the write-heavy workload is significantly more bandwidth-bound: The read-heavy workload is by only 4% bandwidth-bound with all data in CPU memory, while the write-heavy workload is 14% bandwidth-bound. For both CXL data placements, the read-heavy workload is by 21% bandwidth-bound. The write-heavy workload is by 46% and 39% bandwidth-bound with one and four CXL devices. This indicates the potential to increase the bandwidth-bound workload's throughput by interleaving the data across multiple devices as measured in the previous experiment (see Figure 10).

The results indicate that interleaving pages of tree nodes across multiple memory devices is more beneficial for write-intense workloads than for read-intense workloads. The TMA shows that the insert-heavy workload is more memory bandwidth-bound than the read-heavy workload. This allows for improving the performance of write-heavy workloads with additional CXL devices compared to using only one device.

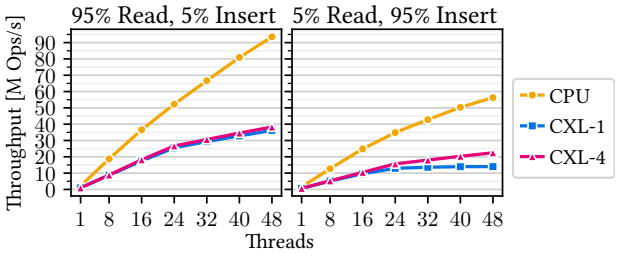


Figure 10: BTreeOLC performance with different data placement configurations (1024 B nodes).

[1] Source code: https://github.com/jmuehlig/btree-benchmarks

5.5 B+Tree Node Size Comparison

We analyze the performance impact of placing data across a different number of CXL devices with different B+tree node sizes.

Setup. We investigate data placement options with all data in either CPU or CXL memory. We perform the two B+tree workloads with 48 threads and different node sizes as shown in Figure 11.

Results. A node size of 1 KiB results in the highest throughput across the three placement options for the read-heavy workload. The throughput improvement when storing node pages on multiple devices compared to a single device is higher with a larger node size. Four devices increase the throughput with a node size of 256 B by only 4% while the throughput increases by 18% with a node size of 4 KiB. Interleaving node pages across multiple CXL devices also shows a higher throughput increase for larger node sizes for the write-heavy workload: four CXL devices increase the throughput with a node size of 256 B by 10%, while the throughput increases by 3.2× with a node size of 4 KiB.

The write-heavy workload with different node sizes shows that the node size leading to the highest performance depends on the data placement. A node size of 1 KiB results in the highest throughput when all data is in CPU memory. When all data is stored on one CXL device, the throughput is the highest with a node size of 512 B. With data interleaved across four devices, a node size of 2 KiB yields the highest throughput.

Bottleneck Analysis. We quantify the memory latency and bandwidth limitations of the write-heavy workload with the TMA method. Figure 12 shows the results, where the share of DRAM-bound μOps is the sum of latency-bound and bandwidth-bound shares. The share of latency-bound μOps is the highest with a node size of 256 B and decreases with higher node sizes. The opposite is true for the bandwidth-bound share, which is the lowest with 256-B nodes and increases with the node size. The share of latency-bound μOps shows a similar trend for both placement configurations. The share is slightly lower with four devices. In contrast, the bandwidth-bound share of μOps diverges with an increasing node size, where the workload becomes more bandwidth-bound with the one-device configuration due to the lower bandwidth.

Our node size investigation results in two findings. First, it shows that the optimal node size for high throughput depends on whether data is stored in CPU memory or CXL memory and the number of CXL devices. Storing all nodes either in CPU memory, one CXL device, or four CXL devices requires four different node sizes to maximize throughput. Second, our investigation shows that changing the B+tree's node sizes can shift the shares to which a workload is bandwidth- or latency-bound. Storing the nodes on multiple CXL devices compared to a single device enables the workload to utilize the increased available CXL memory bandwidth. This reduces the workload's bandwidth-bound share.

6 ANALYTICAL DATABASE WORKLOAD

After evaluating CXL memory's impact on isolated database operations with microbenchmarks, we investigate the impact of placing data in CXL memory for analytical database workloads. We execute the TPC-H benchmark on the in-memory database system Hyrise (Section 6.1) with two data placement configurations (Section 6.2) and compare the resulting performance (Section 6.3).

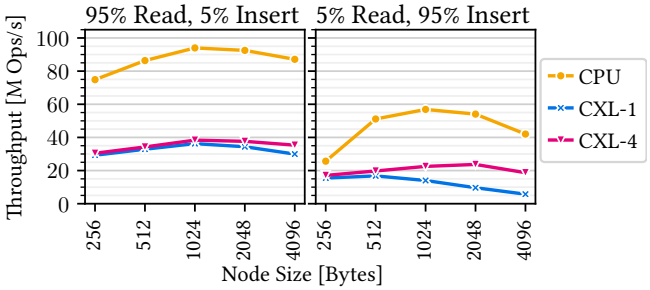


Figure 11: BTreeOLC performance with different data placement configurations and node sizes for 48 threads.

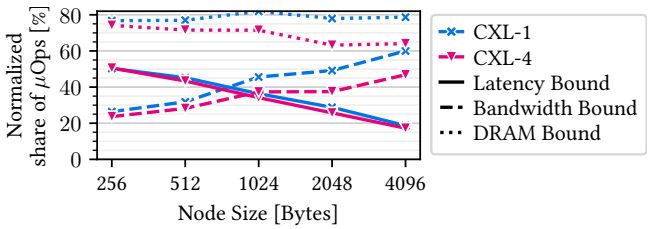


Figure 12: Share of latency- and bandwidth-bound μOps of the write-heavy workload for different node sizes.

6.1 Database System: Hyrise

Hyrise [21] is an open-source, columnar, in-memory database system. It uses a vectorized, push-based execution model with operator-at-a-time execution [10]. Hyrise uses MVCC for transaction processing and an append-only approach comparable to PostgreSQL.

Table Layout. Hyrise divides tables into horizontal partitions with a fixed tuple count. This partitioning splits each column into segments. A partition stores one segment for each column. Segments can be individually encoded with various encoding schemes [9], with dictionary encoding as the default.

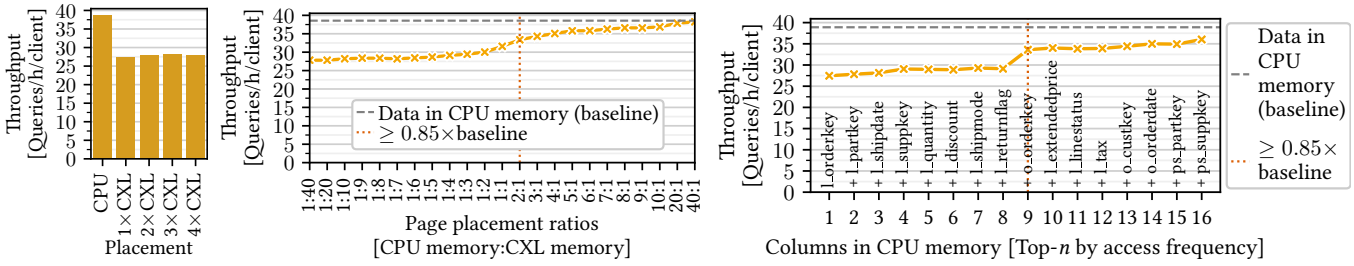
Data Placement. Hyrise uses the polymorphic memory resource (PMR) from the C++ standard library to place data structures in different types of memory [19, 33, 81]. This is achieved by constructing a data structure with a polymorphic allocator. This allocator uses a PMR, which defines memory (de)allocation logic. We implement a PMR (see Section 6.2) to place data in CPU and CXL memory.

Access Counters. Hyrise tracks how often column segments are accessed [19]. For each segment, it counts accesses for sequential, monotonic, random, and point access patterns. We use these access count statistics in one of the placement strategies to identify frequently accessed columns.

6.2 Data Placement Strategies

We store base table data in CPU or CXL memory using two different placement strategies. Other data, including temporary data generated during query processing, is placed in CPU memory.

Linear Memory Allocator. We implement a simple linear memory allocator. It performs allocations by returning pointers to consecutive chunks of a pre-allocated memory region, starting from the region's base address. When pre-allocating the memory region



(a) Round-robin page interleaving. **(b) Weighted page interleaving across CPU memory and one CXL device.** **(c) Column placement across CPU memory and one CXL device.**

Figure 13: TPC-H throughput for different placements. (a) shows results for multiple CXL devices, (b,c) for one CXL device.

from which requests to the allocator are served, we place pages in either CPU or CXL memory. For that pre-allocation, the allocator requires a list of NUMA nodes $n_1, n_2, ..., n_m \in \mathbb{N}_0$ and an interleaving type as input parameters. The interleaving type can either be round-robin or weighted interleaving. For the latter, we additionally define a list of page counts $p_1, p_2, ..., p_m \in \mathbb{N}_0$. Starting with $i = 1$ for the allocated memory region, p_i specifies the number of consecutive pages placed on node n_i. The subsequent p_{i+1} pages are placed on n_{i+1}. If i reaches $m + 1$, it rolls over to 1.

Regardless of the interleaving type, we pre-allocate the memory region using the `mmap` system call and place pages in physical memory of NUMA nodes according to the interleaving type. We implement the allocation logic as PMRs to use it with Hyrise's data placement approach (see Section 6.1).

Page Interleaving Strategy. This strategy places data with OS page granularity in CPU and CXL memory. It uses the linear memory allocator to pre-allocate one large memory region and to interleave pages using one of the described interleaving types. We then use the allocator to construct the segments of all base tables in the pre-allocated and interleaved memory region.

Column Placement Strategy. This strategy places data with table column granularity. Using two linear allocators, we first pre-allocate two separate memory regions: one for CPU memory and the other for CXL memory. Pages of one memory region can still be interleaved across multiple NUMA nodes. When using multiple CXL devices, this allows interleaving pages across all devices, assuming that each device is configured as a separate NUMA node. After pre-allocating the two memory regions, we then construct all segments of a table column in either the CPU or CXL memory region by requesting memory from the corresponding allocator. The placement decision of whether column segments are constructed in CPU or CXL memory is based on access frequencies. We sum all access counters of a column to determine a column's total access count. The placement strategy then stores the (user-defined) n most frequently accessed columns in CPU memory, while the remaining columns are placed in CXL memory.

6.3 Performance Evaluation

We quantify the average TPC-H query throughput for the two data placement strategies. In all experiments, we run the TPC-H benchmark on Hyrise with a scale factor of 100. 10 simulated clients execute the 22 TPC-H queries in random order for 20 minutes. We pin threads to the CPU directly connected to the CXL devices. We use Hyrise's default dictionary encoding for the segments.

6.3.1 Impact of Multiple Devices. **Setup.** We first evaluate the TPC-H performance when placing data across multiple CXL devices. We place all data in the local CPU's memory and interleave pages in a round-robin fashion across one to four devices.

Results. Figure 13a shows the results. Storing all segments in CPU memory achieves a throughput of 38.5 queries per hour per client (Q/h/c). This throughput is the baseline for the remaining placement configurations in our evaluation. Placing all segments in CXL memory with a round-robin page interleaving across one to four devices yields 71% to 73% (i.e., 27.5 Q/h/c to 28 Q/h/c) of the baseline throughput. Using multiple devices does not show significant throughput increases (< 2%) over one CXL device.

The results indicate that the TPC-H workload on Hyrise is mainly latency-bound. This is supported by Dreseler et al.'s [20] analysis, showing that Hyrise spends most of the TPC-H query execution in hash joins and hash aggregations, which results in a large share of random accesses.

6.3.2 Page Interleaving Strategy. **Setup.** We interleave pages with different page count ratios for CPU and CXL memory of one device, starting with 1:40 (i.e., 98% in CXL memory) up to 40:1.

Results. Figure 13b shows the results. Starting with 73% (i.e., 28 Q/h/c) of the baseline throughput for a 1:40 ratio, the throughput converges to the baseline with an increasing share of pages in CPU memory. The configuration with a 2:1 ratio is the first to place most pages in CPU memory with only $\frac{1}{3}$ of the pages in CXL memory. This configuration achieves 87% percent of the baseline throughput.

6.3.3 Column Placement Strategy. **Setup.** We first run all 22 TPC-H queries once to determine the most frequently accessed columns. We then take the access counters of all table segments and calculate the total accesses per column (see Section 6.2). Before running the TPC-H workload for the main experiment, we place the n most frequently accessed columns in CPU memory and the remaining columns in CXL memory.

Results. Figure 13c shows the results. Throughput increases with more columns in CPU memory. Storing the 16 most frequently accessed columns in CPU memory and the rest in CXL memory achieves 94% (36 Q/h/c) of the baseline throughput. The cumulative segment size of the 16 columns is 20.4 GB, which corresponds to

23% of all encoded TPC-H columns (with a total size of 87.7 GB). Storing the eight most frequently accessed columns (i.e., 15% of all segments) in CPU memory results in 75% (29 Q/h/s) baseline performance. Storing `o_orderkey` additionally in CPU memory results in 87% (33.5 Q/h/s) with only 16% of all segments in CPU memory. Keeping `o_orderkey` in fast CPU memory significantly improves the throughput, as this is the join column for the most expensive joins in the TPC-H workload in Hyrise.

We conclude that using CXL memory for cold and warm data while hot data resides in CPU memory can significantly improve the performance when a database system uses both CXL and CPU memory. For workloads with more access skew, we expect an even higher benefit of access frequency-based data placement.

7 ECONOMIC VIABILITY

Traditional Memory Setups. Increasing the memory capacity in a traditional server setup with DDR-attached memory is limited by the number of CPUs and the number of DIMMs that can be attached to a CPU. Current 4th and 5th Gen Intel Xeon scalable processors[2] support eight memory channels with up to two DIMM per channel (DPC) and 4 TiB memory capacity. A setup with 4 TiB require DIMMs with 256 GiB (assuming eight memory channels and two DPC per CPU). Comparing DIMM prices on NewEgg.com[3] shows that current DDR5 DIMMs with such capacities have a more than 2× higher price per GiB capacity than DIMMs with 64 GiB or less. 128 GiB DIMMs are by 1.4× to 1.7× more expensiven per GiB capacity. When avoiding expensive 128 GiB and 256 GiB DIMMs, a CPU with 16 DIMMs can be configured with up to 1 TiB memory.

Alternatively, server administrators can purchase more expensive servers with multiple CPU sockets, where each socket usually multiplies the number of DIMM slots. In our example with 16 DIMMs per CPU, an additional CPU per server only adds 1 TiB (i.e, 16×64 GiB) when avoiding large and expensive DIMM sizes.

CXL Memory Expansion devices can be cheaper than using only CPU memory. The CXL devices used in this work host DDR4 DIMMs, which are cheaper (by > 2× for most DIMM sizes) than DDR5 DIMMs. CXL devices are connected via a PCIe/CXL slot. A few DDR4 DIMMs are sufficient to match the theoretical bandwidth of a PCIe 5 link (e.g., ~64 GB/s with 16 lanes).

Cost Analysis. To analyze the potential cost-benefit of CXL memory devices, we estimate prices of memory configurations with and without CXL memory. A configuration's price includes the cost of CPUs and DIMMs. We use CPU data, including the recommended price, from Intel's product specifications and prices listed on NewEgg.com for DIMMs. We consider the 4th and 5th Gen Intel Xeon scalable processors, specifically the model 8452Y with a cost of about $4 000 as it is the cheapest Platinum CPU supporting two DPC. We further consider DDR4 and DDR5 DIMMs.

We assume CPUs with a full memory population (i.e., 16×16/32/64/128/256 GiB DDR5 DIMMs) and a CXL memory device with 8×128 GiB DDR4 DIMMs, like a CXL device used in our experiments. We compare configurations with only CPU memory, CPU memory plus one CXL device, and CPU memory plus one to four CXL devices. We choose the cheapest option for a given capacity demand and consider setups with up to two CPUs.

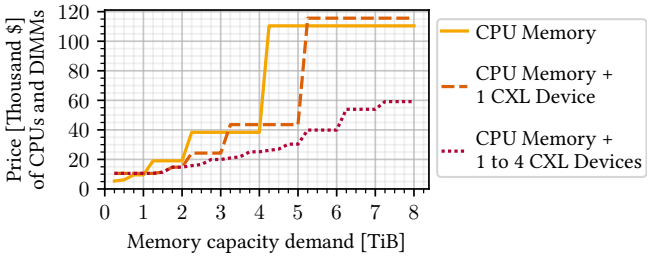


Figure 14: Cheapest CPU and memory configurations with and without CXL memory devices.

Figure 14 shows the price of the cheapest configuration for a given memory capacity demand. For demands less than 1 TiB, using only CPU memory with up to 16×64 GiB is cheaper than a setup with a CXL device. For demands larger than 3 TiB up to 4 TiB, a setup with one CXL device and two CPUs with 32×64 GiB is not sufficient (resulting in 3 TiB). The required configuration contains CPU memory that is sufficiently large (i.e., 4 TiB with 128 GiB DIMMs) plus the additional CXL device. This device generates cost for capacity that is not required. With two devices, the CXL configuration is sufficiently large and cheaper. With 4 TiB < *demand* ≤ 5 TiB, using one CXL device reduces cost by up to 61%. Demands larger than 5 TiB require CPU memory with 32×256 GiB DIMMs, even with one CXL device. The resulting CPU memory capacity is sufficiently large for demands up to 8 TiB, making the configurations with one CXL device more expensive. Using up to four devices reduces cost by 46% to 64% (4 TiB < *demand* ≤ 8 TiB).

Recent work suggests reusing memory DIMMs of decommissioned servers [8, 14, 74, 87], which further reduces the TCO of CXL devices, making them attractive for a wide range of setups.

8 DISCUSSION

Our experiments show the differences in various access characteristics of CXL memory and CPU memory.

CXL for Sequential Accesses. In our latency study, especially random reads exhibit high access latency, while sequential accesses allow the hardware prefetcher to hide the increased access latency for the majority of accesses. Results in Figure 5 demonstrate that writes have a latency of almost 3× of CPU memory writes. These results indicate that, when a database system utilizes both CPU and CXL memory, data structures that are primarily read randomly (e.g., hash tables) or frequently written should be placed in CPU memory to optimize performance, while sequentially accessed data (e.g., frequently scanned columns) can be placed in CXL memory. This aligns with our column scan experiment for high selectivity, where reading columns sequentially from four devices and writing data to CPU memory achieves almost 90% of the baseline performance with all data in CPU memory. While the hardware prefetcher cannot hide the higher latency of random accesses to CXL memory, software prefetching can be used [38, 42, 52]. When database systems increasingly use CXL memory, software prefetching will become even more attractive.

[2]Intel Product Specifications: https://www.intel.com/content/www/us/en/ark.html
[3]We collected prices of NEMIX DDR4/DDR5 288-PIN RDIMMs in March 2025.

Multiple Devices for Bandwidth Expansion. Attaching multiple CXL memory expansion devices to a CPU increases the overall memory bandwidth. Both our throughput microbenchmarks and database operation analysis show that bandwidth-bound workloads can benefit from the bandwidth of multiple devices. Our B+tree performance study with varying node sizes demonstrates that hardware-conscious tuning of data structures can shift a workload accessing a data structure from memory latency-bound to memory bandwidth-bound and vice versa. With limitations of CXL memory both in terms of latency and bandwidth, memory bottlenecks can shift significantly when the corresponding data is in CXL memory. This shows the importance of careful hardware bottleneck analysis of existing data structures and database operators and of adapting them to the characteristics of CXL memory.

CXL for Warm and Cold Data. Our TPC-H performance evaluation shows that prioritizing hot data for CPU memory and placing warm and cold data in CXL memory allows for placing the majority of data in CXL memory with only a moderate performance decrease. As real-world workloads often have more access skew than the TPC-H workload [11, 61], access frequency-based data placement could be even more beneficial in real deployments. While finding an optimal placement strategy is not the focus of this work, developing lightweight approaches for identifying frequently accessed data and studying the benefit of different placement granularities (e.g., table, column, or page) can improve the utilization of CXL memory in database systems. Such approaches may involve co-designs between database systems and operating systems [5].

CXL Lowers Costs. In our economic viability study, we show that server setups with CXL memory can be significantly cheaper. Combined with access frequency-based data placement, CXL memory serves as an option for processing larger amounts of data in memory with CXL-attached memory capacities for significantly lower cost and with only a moderate performance decrease.

New Use Cases with CXL. While our work focuses on the memory expansion use case, recent work suggested database architectures with shared CXL memory attached to multiple compute nodes [14, 32]. Traditionally, when large intermediate results exceed the available CPU memory capacity, the database system spills data partially to storage. Reading data sequentially from CXL memory achieves more throughput than random accesses. Using CXL shared memory as an alternative for storing large intermediate results can be an approach for increasing performance in out-of-CPU-memory scenarios. When intermediate results are subsequently processed by a filtering scan or join—both of which involve scanning, i.e., sequentially processing the input table—spilling to CXL memory can be a suitable use case for shared CXL memory.

9 RELATED WORK

Our work is an in-depth performance study investigating the impact of interleaving data across multiple real CXL memory devices on individual database operations. Existing related studies focus on a single CXL memory expansion device [24, 25, 31, 39, 72], or do not investigate the CXL memory performance impact on individual database operations [2, 3, 26, 44, 51, 68, 72, 73].

Research efforts on SAP HANA [2, 3, 44] have investigated the impact of storing table and temporary operational data on CXL memory devices with end-to-end database benchmarks. Recent work [3] shows that the performance decrease is low when placing data on CXL memory for transactional (TPC-C) workloads, while it is significant for analytical (TPC-DS). The authors attribute the low impact for transactional workloads to general synchronization overhead due to many conflicting locks. Our work complements end-to-end workload evaluations by a detailed analysis of the key database operations from such workloads, which provides deeper insights for algorithm and data structure designs on CXL.

Cho et al. [13] and Tang et al. [73] demonstrate the value of reducing the memory controller contention using CXL with simulations and ASIC-based CXL memory devices, respectively. Our findings on performance decrease when placing an increasing share of data on CXL memory align with the results of these works.

Recent experimental evaluations [31, 51, 72, 73] investigate memory bandwidth expansion using Linux's weighted page interleaving to place pages across local and CXL memory. Liu et al. [51] show that page interleaving can outperform CPU-only memory performance, but the performance impact highly depends on the interleaving ratio. The optimal interleaving ratio is workload-dependent, and poor interleaving can significantly reduce performance. Similar to our work, these works highlight the importance of examining data structures and recent hardware for different approaches to interleaving to develop generalizable recommendations.

10 CONCLUSION

In this paper, we experimentally evaluate and analyze CXL memory performance with four x16 PCIe/CXL memory expansion devices directly attached to a CPU. We quantify the CXL memory access throughput and latency with microbenchmarks, study the performance of single instruction multiple data (SIMD) column scans and B+tree workloads, and quantify the impact of storing data in CXL memory for concurrently running TPC-H queries. We investigate the impact of interleaving data across multiple CXL devices.

Our evaluation shows that bandwidth-bound workloads (e.g., SIMD scans and write-heavy B+tree workloads) can benefit from scaling up the CXL memory bandwidth with multiple devices. We show that workloads limited by write-throughput profit from storing read-only data in CXL memory while performing writes to faster CPU memory. We conclude from our B+tree study with multiple node sizes that, for memory-bound workloads, adapting data structures according to the memory bottleneck (i.e., bandwidth- or latency-bound) improves performance. We conclude that, for analytical workloads, server setups with CXL memory extension devices have the potential to significantly reduce server costs while still providing high query throughput when using placement strategies that prioritize keeping frequently accessed data in CPU memory.

ACKNOWLEDGMENTS

We thank Seagate Technology LLC and Intel Corporation for their support, and Martin Boissier and the anonymous reviewers for their feedback. This work was partially funded by SAP, the German Research Foundation (ref. 414984028), the European Union's Horizon 2020 research and innovation programme (ref. 957407), and the Independent Research Fund Denmark's Inge Lehmann program (grant agreement number 0171-00062B).

REFERENCES

[1] Advanced Micro Devices, Inc. 2023. AMD CDNA 3 Architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf Last access: 2025-06-13.

[2] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 1–5.

[3] Minseon Ahn, Thomas Willhalm, Norman May, Donghun Lee, Suprasad Mutalik Desai, Daniel Booss, Jungmin Kim, Navneet Singh, Daniel Ritter, and Oliver Rebholz. 2024. An Examination of CXL Memory Use Cases for In-Memory Database Management Systems using SAP HANA. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 3827–3840.

[4] Mikkel Møller Andersen and Pinar Tözün. 2022. Micro-architectural analysis of a learned index. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*. 5:1–5:12.

[5] Alexander Baumstark, Marcus Paradies, and Kai-Uwe Sattler. 2025. Lightweight Memory Access Monitoring for Dynamic Data Placement in Tiered Memory Systems. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*. 265–276.

[6] Lawrence Benson, Richard Ebeling, and Tilmann Rabl. 2023. Evaluating SIMD Compiler-Intrinsics for Database Systems. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[7] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-bench: benchmarking persistent memory access. *Proceedings of the VLDB Endowment (PVLDB)* 15, 11 (2022), 2463–2476.

[8] Daniel S Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D Hill, et al. 2023. Design tradeoffs in CXL-based memory pools for public cloud platforms. *IEEE Micro* 43, 2 (2023), 30–38.

[9] Martin Boissier. 2021. Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems. *Proceedings of the VLDB Endowment (PVLDB)* 15, 4 (2021), 780–793.

[10] Martin Boissier, Marcel Weisgut, and Tilmann Rabl. 2025. Compression in Main Memory Database Systems: Cost and Performance Trade-Offs of Workload-Driven Data Encoding. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*. 779–786.

[11] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Proceedings of the TPC Technology Conference (TPCTC)*. 103–119.

[12] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2477–2489.

[13] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. 2024. COAXIAL: A CXL-Centric Memory System for Scalable Servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[14] Yannis Chronis, Anastasia Ailamaki, Lawrence Benson, Helena Caminal, Jana Giceva, Dave Patterson, Eric Sedlar, and Lisa Wu Wills. 2025. Databases in the Era of Memory-Centric Computing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[15] CCIX Consortium. 2019. An Introduction to CCIX - White Paper. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf. Last access: 2025-06-13.

[16] CXL Consortium. 2023. Compute Express Link Specification - Revision 3.1.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*. 143–154.

[18] NVIDIA Corporation. 2024. NVIDIA GH200 Grace Hopper Superchip Architecture. Whitepaper. Version V1.21.

[19] Markus Dreseler. 2022. *Automatic Tiering for In-Memory Database Systems*. Doctoral Thesis. University of Potsdam.

[20] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proceedings of the VLDB Endowment (PVLDB)* 13, 8 (2020), 1206–1220.

[21] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 313–324.

[22] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proceedings of the VLDB Endowment (PVLDB)* 16, 11 (2023), 2769–2782.

[23] Mohamad El-Batal and Hongjian Fan. 2024. Seagate Composable Memory Appliance (Presentation at Open Compute Project). https://www.youtube.com/watch?v=RjKP1mg7bu8&t=1660s Last access: 2025-06-13.

[24] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. 2023. CXL Memory as Persistent Memory for Disaggregated HPC: A Practical Approach. In *Proceedings of the Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*. 983–994.

[25] Andreas Geyer, Johannes Pietrzyk, Alexander Krause, Dirk Habich, Wolfgang Lehner, Christian Färber, and Thomas Willhalm. 2023. Near to Far: An Evaluation of Disaggregated Memory for In-Memory Data Processing. In *Proceedings of the Workshop on Disruptive Memory Systems (DIMES)*. 16–22.

[26] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 287–294.

[27] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph A. Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *Proceedings of the VLDB Endowment (PVLDB)* 8, 1 (2014), 37–48.

[28] Yunyan Guo and Guoliang Li. 2024. A CXL-Powered Database System: Opportunities and Challenges. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 5593–5604.

[29] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 1 (2024), 1–26.

[30] Seokbin Hong, Wonok Kwon, and Myeonghoon Oh. 2020. Hardware Implementation and Analysis of Gen-Z Protocol for Memory-Centric Architecture. *IEEE Access* 8 (2020), 127244–127253.

[31] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. 2024. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[32] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. 2025. Pasha: An Efficient, Scalable Database Architecture for CXL Pods. *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[33] Ben Hurdelhey, Marcel Weisgut, and Martin Boissier. 2023. Workload-Driven Data Placement for Tierless In-Memory Database Systems. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*. 47–70.

[34] Intel Corporation. 2022. Technical Overview Of The 4th Gen Intel Xeon Scalable processor family. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html Last access: 2025-06-13.

[35] Intel Corporation. 2023. Product Brief: 5th Gen Intel® Xeon® Scalable Processors for Edge - Accelerate Demanding and Evolving Edge Workloads with Built-in AI and Security. https://cdrdv2-public.intel.com/795371/5thgen-xeon-edge-product-brief.pdf Last access: 2025-06-13.

[36] Intel Corporation. 2024. 4th Gen Intel® Xeon® Scalable Processor XCC (Codename Sapphire Rapids) Uncore Performance–Monitoring Guide. Revision 001. https://cdrdv2.intel.com/v1/dl/downloadStart/639667?fileName=639667-SPR_XCC_UPG_Guide-Rev_001.pdf Last access: 2025-06-13.

[37] Intel Corporation. 2024. Intel® Xeon® Gold 6542Y Prozessor. https://www.intel.de/content/www/de/de/products/sku/237559/intel-xeon-gold-6542y-processor-60m-cache-2-90-ghz/specifications.html Last access: 2025-06-13.

[38] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proceedings of the VLDB Endowment (PVLDB)* 11, 11 (2018), 1702–1714.

[39] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: Software-Defined Memory Tiering for Heterogeneous Computing Systems With CXL Memory Expander. *IEEE Micro* 43, 2 (2023), 20–29.

[40] Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2018. Adaptive Energy-Control for In-Memory Database Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 351–364.

[41] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[42] Roland Kühn, Jan Mühlig, and Jens Teubner. 2024. How to Be Fast and Not Furious: Looking Under the Hood of CPU Cache Prefetching. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 9:1–9:10.

[43] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on

Compressed Storage using both Vectorization and Compilation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 311–326.

[44] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. 2023. Elastic Use of Far Memory for In-Memory Database Management Systems. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 35–43.

[45] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 185–196.

[46] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Engineering Bulletin* 42, 1 (2019), 73–84.

[47] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *Proceedings of the VLDB Endowment (PVLDB)* 17, 10 (2024), 2568–2575.

[48] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 574–587.

[49] Shang Li, Dhiraj Reddy, and Bruce L. Jacob. 2018. A performance & power comparison of modern high-speed DRAM architectures. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 341–353.

[50] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[51] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. 2024. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization. *CoRR* abs/2409.14317 (2024). arXiv:2409.14317

[52] Fabian Mahling, Marcel Weisgut, and Tilmann Rabl. 2025. Fetch Me If You Can: Evaluating CPU Cache Prefetching and Its Reliability on High Latency Memory. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*.

[53] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 742–755.

[54] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 3461–3472.

[55] Seagate Technology Team Member(s). 2024. Composable Memory Appliance (CMA) Base Specification V1.1. Open Compute Project.

[56] Timothy Prickett Morgan. 2020. CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/ Last access: 2025-06-13.

[57] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1331–1344.

[58] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 1–26.

[59] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *Proceedings of the VLDB Endowment (PVLDB)* 28, 4 (2019), 451–471.

[60] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1981–1984.

[61] Tilmann Rabl, Meikel Poess, Hans-Arno Jacobsen, Patrick E. O'Neil, and Elizabeth J. O'Neil. 2013. Variations of the star schema benchmark to test the effects of data skew on query performance. In *Proceedings of the International Conference on Performance Engineering (ICPE)*. 361–372.

[62] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. 2024. A Three-Tier Buffer Manager Integrating CXL Device Memory for Database System. In *Proceedings of the Joint International Workshop on Big Data Management on Emerging Hardware and Data Management on Virtualized Active Systems (HardBD & Active)*.

[63] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Computing Surveys (CSUR)* 55, 2 (2023), 11:1–11:38.

[64] Subhash Saini, John Baron, Johnny Chang, Robert Hood, and Haoqiang Jin. 2022. Performance Evaluation of a Supercomputer Based on AMD Rome and Intel Cascade Lake Processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 848–859.

[65] Gabin Schieffer, Ruimin Shi, Stefano Markidis, Andreas Herten, Jennifer Faj, and Ivy Peng. 2024. Understanding Data Movement in AMD Multi-GPU Systems with Infinity Fabric. In *Proceedings of the Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*. 567–576.

[66] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proceedings of the VLDB Endowment (PVLDB)* 17, 11 (2024), 3290–3303.

[67] Debendra Das Sharma. 2023. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy. *IEEE Micro* 43, 2 (2023), 99–109. https://doi.org/10.1109/MM.2022.3228561

[68] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Computing Surveys (CSUR)* 56, 11 (2024), 290:1–290:37.

[69] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. 2017. A methodology for OLTP micro-architectural analysis. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 1:1–1:10.

[70] Lukas Steiner, Matthias Jung, and Norbert Wehn. 2021. Exploration of DDR5 with the Open-Source Simulator DRAMSys. In *Proceedings of the Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV)*. 1–11.

[71] Jeffrey Stuecheli, William J. Starke, John D. Irish, L. Baba Arimilli, Daniel M. Dreps, Bart Blaner, Curt Wollbrink, and Brian Allison. 2018. IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI. *IBM Journal of Research and Development* 62, 4/5 (2018), 8:1–8:8.

[72] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 105–121.

[73] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 818–833.

[74] Jaylen Wang, Daniel S Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, et al. 2024. Designing cloud servers for lower carbon. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 452–470.

[75] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the International Conference on Management of Data (SIGMOD / PODS)*. 37–44.

[76] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1033–1048.

[77] Ruihong Wang, Chuqing Gao, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2024. Optimizing LSM-based indexes for disaggregated memory. *The VLDB Journal* 33, 6 (2024), 1813–1836.

[78] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)* 16, 1 (2022), 15–22.

[79] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 2835–2849.

[80] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 609–621.

[81] Marcel Weisgut, Daniel Ritter, Martin Boissier, and Michael Perscheid. 2022. Separated Allocator Metadata in Disaggregated In-Memory Databases: Friend or Foe?. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1202–1208.

[82] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. On-Demand State Separation for Cloud Data Warehousing. *Proceedings of the VLDB Endowment (PVLDB)* 15, 11 (2022), 2966–2979.

[83] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 169–182.

[84] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44.

[85] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *Proceedings of the Conference on Innovative Data Systems*

Research (CIDR).

[86] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)* 14, 10 (2021), 1900–1912.

[87] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, et al. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 37–56.

[88] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin* 28, 2 (2005), 17–22.