A Three-Tier Buffer Manager Integrating CXL Device Memory for Database Systems

Niklas Riekenbrauck
Hasso Plattner Institute
Potsdam, Germany
niklas.riekenbrauck@student.hpi.de

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

Daniel Lindner
Hasso Plattner Institute
Potsdam, Germany
daniel.lindner@hpi.de

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

***Abstract*—Compute Express Link (CXL) is a new interconnect for attaching byte-addressable memory on PCI-connected devices to a CPU. The interconnect allows a database system to place data on local memory, CXL device memory, and persistent disk storage. While three-tier buffer managers integrating persistent memory (PMem) exist, CXL device memory has not been integrated into a multi-tier buffer manager architecture. Existing three-tier buffer managers integrating PMem use pointer swizzling to address buffered pages, which is an invasive and hard-to-implement technique. This work presents a three-tier buffer manager that integrates CXL device memory. The design combines hardware-supported virtual memory for efficient page translation and a probabilistic page migration policy to determine on which tier pages are located. We demonstrate that these approaches combined allow a simple integration of CXL device memory into a database system. We evaluate the buffer manager with different configurations and workloads based on the YCSB benchmark on a CXL Type 3 device prototype. Our evaluation demonstrates that expanding a server's memory with CXL device memory can be used to keep more data in memory and to reduce spilling data to slow disk storage.**

***Index Terms*—CXL, buffer manager, page management, database system, virtual memory, probabilistic page migration**

I. Introduction

Compute Express Link (CXL) is a new interconnect based on the physical layer of PCIe. CXL can connect a peripheral device with a CPU, allowing cache-coherent access to the device memory [1]. Accessing memory over CXL exhibits different memory characteristics, such as higher latency than local memory connected via Double Data Rate (DDR) [1], [2].

Traditional disk-based database management systems (DBMSs) use secondary disk storage as primary data location. For query processing, a buffer manager loads data into local memory. With additional CXL device memory, data can be located on three tiers: on byte-addressable local and device memory and on persistent disk storage. While three-tier buffer managers exist for DRAM, persistent memory (PMem), and solid-state drive (SSD) [3], [4], CXL device memory has not been integrated into a multi-tier buffer manager architecture.

HyMem is a single-threaded buffer manager using PMem and DRAM as selective caches on top of the SSD level [3]. Zhou et al. [4] extended the work on HyMem with *Spitfire*, a concurrent buffer manager. It uses a probabilistic migration policy to determine on which tier pages are located and has superior performance over HyMem's page migration.

HyMem and Spitfire use pointer swizzling to address buffered pages, which is an invasive technique that requires a buffer-managed data structure to be adapted accordingly [5]. Leis et al. proposed hardware-supported virtual memory-based translation of page identifiers (PIDs) to physical memory addresses as a non-invasive and easy-to-implement alternative to pointer swizzling [5]. While the approach shows high performance and low implementation complexity, it lacks support for multiple byte-addressable memory tiers.

In this work, we present a three-tier buffer manager that integrates CXL device memory. The design combines virtual memory-based PID translation [5] and a probabilistic page migration policy [4] to determine on which tier pages are located. This work demonstrates that these approaches combined allow a simple integration of CXL device memory into a DBMS. We evaluate the buffer manager design and its components in an isolated manner with different configurations and workloads based on the YCSB benchmark [6]. Our evaluation demonstrates that expanding a server's memory with CXL device memory can be used for a buffer manager to keep more data in memory and to reduce spilling data to slow disk storage. We present primitives to integrate the proposed design into an in-memory DBMS and demonstrate the integration of the buffer manager into the in-memory DBMS *Hyrise* [7]

In summary, this work makes the following contributions:

1) We demonstrate the integration of CXL device memory into a database system's buffer manager using virtual memory-based PID translation and probabilistic page migration (Section III). We provide integration details and an open-source implementation[1] (Section III-G).
2) We experimentally evaluate the buffer manager with prototypical CXL device memory and show its benefit of supporting larger-than-local-memory workloads with higher throughput than a traditional two-tier design locating data only on local memory and SSD (Section IV).
3) We discuss how page migration across multiple memory tiers can further be optimized (Section V).

II. Background

This section introduces the CXL interconnect and buffer pool management concepts that we build our work upon.

[1]Source code: https://github.com/hyrise/hyrise/tree/paper/buffermanager.

A. Compute Express Link (CXL)

This section is based on Sharma's introduction to Compute Express Link [1] if not mentioned otherwise.

Protocols. CXL is an interconnect standard to connect CPUs and peripheral devices. It is based on PCIe's physical layer and adds coherency and memory semantics. The standard specifies three protocols. *CXL.io* provides non-coherent load and store semantics and is used for device discovery, status reporting, address translation, and direct memory access (DMA). *CXL.cache* allows a device to cache data stored on system memory. *CXL.mem* allows CPUs to access CXL device memory as cache-able memory.

Devices. The standard specifies three device types. Each device supports CXL.io. *Type 1* devices support CXL.cache, *Type 2* devices support CXL.cache and CXL.mem, and *Type 3* devices support CXL.mem. Use cases for the device types are smart network interface cards with coherent access to system memory (Type 1), accelerators, such as GPUs and FPGAs (Type 2), and memory expansion devices (Type 3).

Revisions. Five backward-compatible revisions exist. CXL 1.0 specifies the three protocols and device types. CXL 1.1 adds compliance test mechanisms. Later revisions introduce resource pooling and topologies with single-level switching (2.0), sharing device memory and support for larger topologies with up to 4096 endpoints and multi-level switching (3.0), and DMA across system domains (3.1). In this work, we use a CXL 1.1-compliant Type 3 device prototype.

Programming with CXL Device Memory. CXL device memory is exposed to a CPU as a memory-only NUMA node [2], [8]. This allows programmers to utilize NUMA-related system calls to interact with CXL. For example, `mbind` allows setting a memory policy for a given virtual memory region and `move_pages` allows moving individual operating system (OS) pages between nodes.

B. Buffer Pool Management

Traditional DBMSs store data mainly on disk for cost-efficiency and persistence [9]. Stored data is split into pages, with a page size being a multiple of the OS page size. The DBMS loads pages into a buffer pool to access the data and writes modified pages back to disk. The buffer manager is in charge of handling the buffer pool, which holds a fixed number of pages, aiming to reduce disk I/O as it is more expensive than accessing data in memory. Unused pages must be *evicted* if the DBMS needs to access a page not present in the buffer pool and the pool is full. A page eviction strategy decides on which page to evict. A page is pinned during access to avoid eviction and later unpinned for release. The design of a buffer manager depends on several factors, such as the hardware setup, given DBMS guarantees, and workloads. The design restrictions of a buffer manager influence the storage layout of tuples or index structures.

Conventional Page Management. Maintaining a central hash table is a common approach to addressing pages, enabling a direct mapping from PIDs to respective page pointers. While this approach worked for traditional disk-based DBMS, it is a bottleneck for processing data that fully fits into memory: Each lookup accesses the hash table, even for cache hits [5]. Furthermore, a lookup requires another indirection when the virtual memory pointer has to be translated to a physical page pointer through the OS page table [5]. Pointer swizzling is an invasive technique eliminating the central hash table lookup for in-memory workloads: a unique PID is *swizzled*, i.e., transformed to a virtual memory pointer, for access and *unswizzled*, i.e., converted back to the corresponding PID, before eviction to disk [10]. Following a swizzled reference does not require a hash table lookup. While this technique improves in-memory workload performance, each data structure has to be adapted, complicating the implementation [5]. Using data structures not based on trees or lists requires complex workarounds, making pointer swizzling unsuitable in many use cases.

OS-Supported Page Management. OS-provided memory-mapped file I/O can be used as an alternative to buffer managers in a DBMS. However, Crotty et al. [11] identified the use of memory-mapped file I/O via the `mmap` system call to be unsuitable for a DBMS due to data safety and performance issues. One major problem with using file I/O via `mmap` is that the DBMS cannot control page eviction.

Leis et al. [5] presented *vmcache*, a buffer manager that relies on virtual memory for translating PIDs to physical memory addresses but hands page faulting and eviction to the DBMS. Their approach neither requires additional hash tables for page management nor invasive pointer swizzling for managed data structures. Instead of handling page faults, *vmcache* creates a non-file-backed, anonymous virtual memory region using `mmap` and relies on the OS page table and the translation lookaside buffer (TLB) for fast page accesses. Page eviction is controlled by the `madvise(MADV_DONTNEED)` system call, which frees the physical page backing the virtual memory, and I/O using system calls. Unlike using file-backed `mmap`, the DBMS controls page eviction and reading and writing pages. This approach enables a simple, scalable implementation while supporting arbitrary data structures.

Page Sizes. Buffer managers can support a fixed page size or variable page sizes. A fixed page size simplifies the allocation logic, but objects larger than the page size must be split across multiple pages. Accessing such objects requires more complex code paths [5]. With variable page sizes, the buffer manager can store a larger object on a page with a suitable size, circumventing object splitting. However, allowing multiple page sizes in a single buffer pool causes external fragmentation issues [12]. Utilizing the OS's mapping between virtual and physical addresses can avoid this fragmentation [5], [12]. A buffer manager can create a virtual memory region for each supported page size that is large enough to hold the entire buffer pool, but each region is not fully backed with physical memory. The buffer manager can use system calls (e.g., `madvise(MADV_DONTNEED)`) to ensure that the accumulated amount of allocated physical memory does not exceed the buffer pool's capacity.

Probabilistic Page Migration. A three-tier hierarchy offers multiple page movement paths between the tiers (see [4],

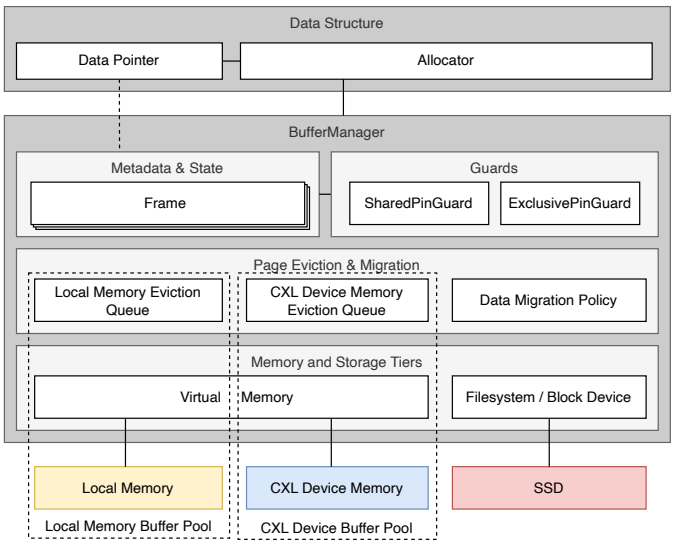

Fig. 1. Architecture of the three-tier buffer manager.

Fig. 3). Zhou et al. proposed moving pages between DRAM, PMem, and SSD based on configurable probabilities. A read path moves data from SSD to PMem, then to DRAM, and lastly to the CPU cache. The write path moves data from the CPU cache to DRAM, then to PMem, and finally to an SSD. Furthermore, the CPU can directly (1) read from and (2) write to PMem, and pages can directly be (3) copied to DRAM from an SSD and (4) written back from DRAM to an SSD. With specific probabilities for the latter four data paths, pages are transferred to a higher tier on access. If accessed frequently, the page is eventually promoted to the fastest tier.

III. CXL-Based Buffer Management

This section presents the design of our three-tier buffer manager and how it integrates CXL device memory.

A. System Overview

Figure 1 shows the buffer manager components. *Frames* handle the pages' states and offer additional methods for latching operations when accessing a page. *Guards* are helper objects simplifying pinning and concurrent page handling. One buffer pool exists per byte-addressable tier. A migration policy decides where to move pages across the tiers. It controls page eviction with one eviction queue per buffer pool. The pools combine the eviction mechanism and access to the physical memory through virtual memory. A buffer-managed data structure uses the buffer manager as an allocator.

B. Page Management

The buffer manager utilizes virtual memory for PID-to-pointer translation. In contrast to pointer swizzling or hash-table-based buffer management, virtual memory-based translation offers an abstraction over local memory or CXL device memory without implementing complex code logic. The buffer manager creates a large virtual memory region using `mmap` without file backing. With demand paging, new pages are allocated either on local memory or CXL device memory. We assume that each byte-addressable tier is exposed as one or more NUMA nodes. Memory allocations for a virtual memory region can be bound to the desired NUMA node via the `mbind` system call. On access, the physical memory is allocated on the configured node(s), leading to an interleaved memory region with physical pages being backed by either local memory, CXL device memory, or nothing (when initially read). Maintaining only one buffered page copy is a limitation of virtual memory-based PID translation and simplifies memory management compared to existing three-tier buffer managers [4], [13]. However, it loses the ability of parallel access to copies of the same page on both tiers. Inspired by vmcache, we evict pages to SSD using `madvise(MADV_DONTNEED)` and `pwrite` system calls and read into virtual memory using `pread`. Leis et al. [5] argue that the performance of virtual memory management approaches is limited by the impact of TLB flushes and the unscalable OS page allocator. This also applies to moving pages from and to CXL device memory.

C. Variable Page Sizes

We define a set of page sizes in power-two steps from 4 KiB to 2 MiB similar to *Umbra* [12]. Each page size is assigned to a fraction of the virtual memory region and managed by an array of frames. Using large pages improves the performance of allocating large objects. However, larger pages also result in higher latency for buffer manager operations. Variable-sized pages can store data structures without fragmenting to multiple pages. Discrete page sizes allow to directly calculate a page's address (see Section III-G).

D. State Management

We leverage the state management and synchronization mechanism proposed by *vmcache* [5]. Frames store the metadata of individual pages. A frame is a 64-bit atomic integer storing the page state in the upper 16 bits and the page version in the lower 40 bits. The state encodes if the page is currently evicted or latched in exclusive or shared mode. While vmcache uses the version for optimistic latching, we use it for the page eviction mechanism. Additionally, we store a dirty flag with one bit and the current NUMA node in seven bits. Storing this data requires us to reduce the number of bits for the version compared to vmcache. The atomic integer is updated using a compare-and-swap operation. Busy waiting on the frame's latching state serializes concurrent page accesses. We use one array of frames per page size. Each array stores the frames for all pages of the same size. Frame arrays reside on local memory, allowing fast access to the state by a fixed offset.

E. Page Eviction

Our eviction strategy approximates least-recently-used (LRU) using a second-chance first-in, first-out (FIFO) queue [14, p. 212] as commonly implemented [15], [16]. The queue can contain frames belonging to different size types. After unlocking a page, it is inserted into the queue with its current version. When reaching a predefined memory limit, several pages in the queue are evaluated for batch eviction to fulfill a

page placement request. If the page version in the queue and the frame's page version do not match or if the page is locked, the queue contains a newer version of the page so we can discard the current page. If the versions match, the page may be ready for eviction. While in `UNLOCKED` state, we reinsert the page at the end of the queue and transition to state `MARKED`. Only `MARKED` pages qualify for eviction. If so, we acquire an exclusive latch of the page and evict the data to CXL device memory or SSD. Reinserting pages ensures the second-chance criterion [14, p. 212]. We use *oneTBB*'s [17] concurrent queue implementation (`tbb::concurrent_queue`).

F. Page Migration

We apply Zhou et al.'s probabilistic page migration approach defined by four probabilities. When a CPU accesses a page on CXL device memory, the buffer manager moves the page to local memory before performing a read and write operation with the probabilities L_r and L_w. When the buffer manager loads a page from SSD, it loads the page into CXL device memory with a probability of C_r and into local memory with a probability of $1-C_r$. When the buffer manager evicts a page, the page is written to CXL device memory with a probability of C_w and to SSD with a probability of $1-C_w$. The buffer manager considers an initial page allocation as a virtual load of a zero page from SSD, so C_r applies. Lazy policies with lower probabilities lead to less frequent migrations, while eager policies with high probabilities move pages more often. We use the system calls `mbind` and `move_pages` to move pages between tiers.

G. Integration Concepts

We integrate our buffer manager into the in-memory DBMS Hyrise without changing the system's core concepts.

Mapping Pages to Data Structures. Page state and actual data are stored in separate memory locations. The page's data can be addressed via a virtual memory pointer and the state by an index in the frame array. As shown in Figure 2, a page identifier is a 64-bit value containing a validity bit, 5 bits for the page size, and an index (58 bits). The combination of the page size and the index can be unambiguously converted into a pointer to the page and vice versa.

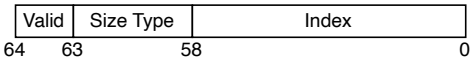

Fig. 2. Layout of a page identifier.

Pin Guards. Pin guards are programming primitives that simplify latching and pinning operations of data structures [15]. We use the *RAII* technique in C++ to ensure safe memory access within a scope. Pin guards must be explicitly placed at appropriate locations where data is read or written.

IV. Evaluation

We evaluate the buffer manager and its components with different configurations and workloads based on the YCSB benchmark [6]. We investigate the scalability of the buffer manager on local and CXL device memory, as well as the performance impact of migration policies.

We conduct the experiments on a dual-socket platform with 4th Generation Intel Xeon CPUs (*Sapphire Rapids*), each having 48 cores. The server has 256 GiB (8 × 32 GiB) DDR5 DIMMs on the first socket (NUMA node 0), which we use as local memory in the experiments. A CXL 1.1 Type 3 device prototype provided by Seagate Technology LLC is attached via a PCIe/CXL x8 Gen4 connection to node 0. The server contains a *Micron Crucial CT500P1SSD8* NVM Express (NVMe) SSD with 500 GB of capacity. The device is an FPGA-based (Bittware Xilinx VU13P) memory controller containing 64 GiB (4 × 16 GiB) DDR4 DIMMs with a speed of 2400 MT/s. The OS is a pre-released Linux kernel version 6.3.0-060300rc1-generic to support CXL memory.

We use Intel's Memory Latency Checker to measure memory access performance. The measurements show an idle latency of 125.5 ns and a maximum bandwidth of 224.3 GB/s when accessing local memory from node 0. For CXL device memory, we observe 557 ns and 9.4 GB/s. Note that we use a preliminary CXL device prototype. Substantially lower access latency and higher bandwidth are expected for production-ready CXL Type 3 devices.

A. Scalability

We run microbenchmarks with workloads based on the YCSB benchmark to evaluate various read and write patterns. We use one cache line (64 Byte) representing one tuple and AVX-512 load-store instructions to simulate tuple access. The microbenchmarks perform point lookups, updates, and full-page scans. We adapt the workloads by creating a fixed set of tuples distributed over pages representing a table and a set of operations that are executed as the workload on a table. We model access skew by generating tuple identifiers for each operation with a *zipfian* distribution [18].

When an operation is executed, we pin the page in shared mode or exclusively for updates. For point lookups and update operations, we select a random cache line on the page and perform the `_mm512_load_si512` AVX-512 instruction for loading a cache line into a register and `_mm512_stream_si512` for simulated stores with a non-temporal hint. Scan operations sequentially access a page using subsequent calls of `_mm512_load_si512`. We do not perform delete and insert operations as they require additional tracking of tuple lifetimes. For all benchmarks, we measure the throughput in operations per second and the latency of each operation. We use the following workload definitions:

- **Update Heavy** consists of 50% point lookups and 50% updates (equivalent to YCSB Workload A).
- **Read Mostly** performs about 95% of point lookups and 5% updates (similar to YCSB Workload B).
- **Scan** performs full scans of pages with 5% updates (similar to YCSB Workload E).

Setup. We evaluate the buffer manager's scalability with increased concurrent access to local and CXL device memory. The benchmark execution is bound to NUMA node 0 to

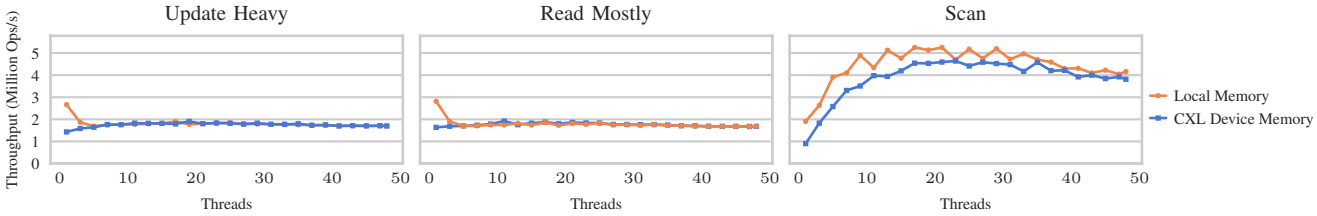

Fig. 3. Throughput on local and CXL device memory with an increasing number of threads.

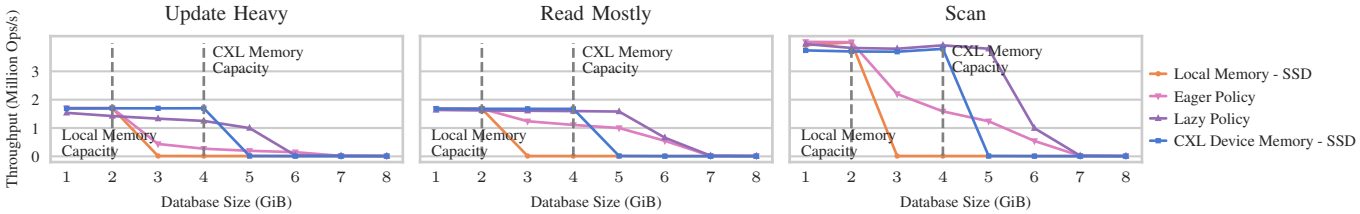

Fig. 4. Throughput for different migration policies and database sizes. 2 GiB buffer pool on local and 4 GiB on CXL device memory.

avoid NUMA effects. We turned off hyper-threading, CPU scaling, and all NUMA balancing mechanisms. We execute the workloads with a fixed database size of 2 GiB. The benchmark is separately executed for a buffer pool of 2 GiB for local and CXL device memory. For each workload, we increase the number of threads from one to 48 in steps of two.

Results. Figure 3 shows the results. Workloads that primarily use point lookups, i.e., *Update Heavy* and *Read Mostly*, behave similarly on local and CXL device memory. *Update Heavy* on CXL device memory benefits slightly from the increased number of threads. Local memory benefits from fewer threads. With more than 10 threads, both local and CXL device memory exhibit similar throughput. *Scan* on local memory is up to two times faster for a low number of threads but converges to the throughput of CXL device memory with an increasing number of threads. The average latency increases linearly for all workloads with more threads. Operations for *Read Mostly* and *Update Heavy* workloads take about 300 ns for a low number of threads and 28 µs for 48 threads. The average latency for *Scan* is about 12 µs on CXL device memory at 48 threads. Access latencies differ between local and CXL device memory for scans with fewer threads.

Discussion. The results suggest that the achieved performance is limited by our buffer manager implementation. With increasing threads, multiple threads access several of the eight atomic page states in a 64-byte cache line that must be synchronized between CPU cores, causing cache-line contention. This effect is called *false sharing* [14, p. 561]. Furthermore, we see the queue implementation with `tbb::concurrent_queue` as another point of contention [19]. A lower amount of threads and large scan operations induce less contention.

B. Migration Policy

We evaluate the impact of two-tier and three-tier hierarchies on the throughput of the YCSB workloads with fixed buffer pool sizes. With increasing database size, we induce more page eviction and migration operations.

Setup. We set a fixed buffer pool size of 2 GiB for local memory and 4 GiB for CXL device memory. The buffer manager uses all three tiers. We specify an *Eager Policy* with a uniform page migration probability of one, leading to frequent page migrations, and a *Lazy Policy* with a uniform probability of 0.2, leading to less frequent migrations. For comparison, we configure two-tier hierarchies with an SSD and either local memory (*Local Memory-SSD*), or CXL device memory (*CXL Device Memory-SSD*). For the Local Memory-SSD setup, the buffer manager neither loads pages from SSD into CXL device memory ($C_r = 0$) nor evicts pages from local memory to CXL device memory ($C_w = 0$). For the CXL-SSD setup, C_r and C_w are set to one, but accessing a page on CXL device memory never triggers a move to local memory (i.e., L_r and $L_w = 0$). The database size varies between 1 GiB and 8 GiB. The benchmark is executed with 48 threads.

Results. Figure 4 shows the YCSB throughput for different migration policies. In the two-tier setups, the throughput is similar in all workloads until the buffer pool size is reached. For *Scan*, *Local Memory-SSD* shows a slightly higher throughput than *CXL Device Memory-SSD*. The throughput drops drastically when the database size exceeds the buffer pool size due to costly page migration to the SSD. With three tiers, the total buffer pool size expands to 6 GiB. The lazy policy outperforms most other configurations in all workloads due to reduced page migrations and a larger buffer. The eager policy's throughput already drops with smaller database sizes due to many page migrations. In the *Update Heavy* workload, we observe a tipping point at a database size of 6 GiB, where the eager policy outperforms the lazy policy. Moving pages to SSD has the highest throughput impact, making the respective workload I/O-bound.

Discussion. The results show that integrating CXL device memory into a storage hierarchy can achieve higher throughput for working sets that exceed the local memory capacity. It also shows that a lazy migration policy performs overall better than an eager policy, which aligns with Zhou et al.'s observations [4]. For both three-tier policies, the throughput

decreases with database sizes that do not exceed the combined buffer capacity of local memory and CXL device memory. This happens because loaded and newly created pages can be allocated on local memory or CXL device memory, requiring page migrations and evictions when one individual tier reached its capacity. The optimal policy configuration lies in between the bounds of lazy and eager policies. A lazy policy can trigger even less page migrations with probabilities smaller than 0.2. While we demonstrate the performance of exemplary configurations, the optimal policy configuration depends on the workload and the devices access characteristics and requires parameter tuning for a given hardware setup.

V. Migration Optimization Discussion

Experiments suggest that the performance of `move_pages` between volatile tiers is a limitation and that using `mbind` with the `MPOL_MF_MOVE` flag doubles the page movement throughput. However, both system calls are serialized due to the single-threaded design of the OS [20]. The movement throughput can be improved without relying on kernel modifications: Each thread maintains a thread-local buffer with a region backed by the target memory tier. For page movement, the buffer manager copies the data from the target memory region into the threads' buffers. Then, it calls `madvise(MADV_DONTNEED)` to remove the physical memory backing of the target region and reassigns the NUMA node using `mbind`. Then, the data is copied back from the threads' buffers into the target region, allocating physical memory on the defined NUMA node. While this increases the number of data copies, it reduces the time spent in system calls to maximize parallelizable operations.

VI. Related Work

OS-Based Page Migration. Several OS-based approaches exist to perform page management across multiple memory tiers. Yan et al. propose *Nimble* [20], a modified Linux kernel optimized for page migration. Nimble provides transparent huge pages (THP) migration, multi-threaded page migration, and concurrent migration of multiple pages, leading to up to 15 times higher throughput of raw page migration. For automatic page placement across multiple memory tiers, the Linux kernel offers *AutoTiering* [21] and optimized NUMA balancing (*AutoNUMA*) [22]. The latter promotes hot pages using page faulting and demotes cold pages in the background.

AutoNUMA can starve pages on CPU-less nodes (e.g., CXL device memory). Application performance degrades the more pages are placed on CXL device memory due to higher access latency [23]. Maruf et al. [23] propose an application-transparent page placement mechanism (TPP) approach for CXL device memory, which decouples memory reclamation from allocation, allowing to free local memory pages asynchronously. Their approach outperforms the previous approaches for memory tiering by up to 18%. TPP does not allow explicit control over page allocation and swapping to SSD.

Three-Tier Buffer Pool Management. Recent three-tier buffer manager designs leverage PMem as an additional tier for memory and persistent storage [4], [13]. Van Renen et al. [3], [13] proposed HyMem, a single-threaded buffer manager, which uses PMem and DRAM as selective caches on top of the SSD level. Whenever a page is accessed, it resides in DRAM. For both DRAM and PMem, the clock replacement algorithm is used for page eviction. While a page to be accessed residing on SSD is always loaded into DRAM, a page residing on DRAM gets evicted to PMem if it was recently evicted to SSD, and to SSD otherwise. HyMem uses fixed-sized 16 KiB pages. However, pages located on PMem can be partially loaded into DRAM using *cacheline-grained loading* and a smaller page type (mini page). The authors evaluated HyMem on emulated PMem hardware.

Zhou et al. [4] extend previous work on HyMem with Spitfire, a concurrent buffer manager evaluated on real PMem hardware. The authors employed a probabilistic data migration strategy (see Section II-B), which has superior performance over HyMem's data migration policy.

HyMem [13] and Spitfire [4] show that a three-tier buffer manager design can be beneficial over a two-tier DRAM-SSD design. However, the implementations are fairly complicated and require the use of pointer swizzling. Leis et al.'s virtual memory-based buffer manager approach [5] (see Section II-B) serves as a simple and extendable architecture. None of the existing related work allows simple integration of local memory, CXL device memory, and SSDs for explicit buffer management in a DBMS, which we address with this work.

VII. Conclusion & Future Work

For working sets larger than local memory, our evaluation shows that a buffer manager with local memory and additional CXL device memory as memory capacity expansion maintains a higher throughput compared to a traditional two-tier hierarchy with local memory and SSD. Loading and evicting pages from and to SSD has the highest throughput impact. While the used CXL device memory has higher access latency and lower throughput than local memory, accesses to the device's memory are faster than accesses to SSD. Current results are bound to the CXL device prototype characteristics. We expect significantly increased performance for production-level devices. Future work needs to improve the buffer manager's scalability and evaluate the impact of storing data on CXL device memory in a three-tier storage hierarchy compared to a two-tier hierarchy. While our work is limited to a single device prototype, future work needs to evaluate how different throughput and latency characteristics of CXL Type 3 devices influence the quality of three-tier buffer manager designs.

Acknowledgements

We thank Seagate Technology LLC, especially Hongjian Fan, Jon Trantham, and Matthew Totin, for their support of this research and for providing us with the CXL device. This work was partially funded by SAP SE, the German Research Foundation (ref. 414984028), and the European Union's Horizon 2020 research and innovation programme (ref. 957407).

References

[1] D. D. Sharma, R. Blankenship, and D. S. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," Jun. 2023. [Online]. Available: http://arxiv.org/abs/2306.11227

[2] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, I. Jeong, R. Wang, and N. S. Kim, "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices," Apr. 2023. [Online]. Available: https://arxiv.org/abs/2303.15375

[3] A. van Renen, "Persistent memory in database systems," Ph.D. dissertation, Technical University of Munich, Germany, 2022. [Online]. Available: https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20221026-1638761-1-6

[4] X. Zhou, J. Arulraj, A. Pavlo, and D. Cohen, "Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2021, pp. 2195–2207.

[5] V. Leis, A. Alhomssi, T. Ziegler, Y. Loeck, and C. Dietrich, "Virtual-Memory Assisted Buffer Management," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–25, 2023.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010, pp. 143–154.

[7] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner, "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)*, 2019, pp. 313–324.

[8] M. Ahn, A. Chang, D. Lee, J. Gim, J. Kim, J. Jung, O. Rebholz, V. Pham, K. Malladi, and Y. S. Ki, "Enabling CXL Memory Expansion for In-Memory Database Management Systems," in *Data Management on New Hardware*, 2022, pp. 1–5.

[9] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 4, pp. 560–595, 1984.

[10] A. Kemper and D. Kossmann, "Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis," *The VLDB Journal*, vol. 4, no. 3, pp. 519–567, 1995.

[11] A. Crotty, V. Leis, and A. Pavlo, "Are You Sure You Want to Use MMAP in Your Database Management System?" in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2022, pp. 1–7.

[12] T. Neumann and M. Freitag, "Umbra: A Disk-Based System with In-Memory Performance," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2020, pp. 1–7.

[13] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato, "Managing Non-Volatile Memory in Database Systems," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. Houston TX USA: ACM, 2018, pp. 1541–1555.

[14] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Prentice Hall Press, Feb. 2014.

[15] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "LeanStore: In-Memory Data Management beyond Main Memory," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2018, pp. 185–196.

[16] L. Kuiper, M. Raasveldt, and H. Mühleisen, "Efficient External Sorting in DuckDB," *Proceedings of BICOD 2021/2022*, pp. 40–45, 2021.

[17] "oneAPI Threading Building Blocks," 2024. [Online]. Available: https://github.com/oneapi-src/oneTBB

[18] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," *Proceedings of the International Conference on Management of Data (SIGMOD)*, vol. 23, no. 2, pp. 243–252, 1994.

[19] C. Desrochers, "A Fast General Purpose Lock-Free Queue for C++," Nov. 2014. [Online]. Available: https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.htm#benchmarks

[20] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 331–345.

[21] J. Kim, W. Choe, and J. Ahn, "Exploring the Design Space of Page Management for Multi-Tiered Memory Systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2021, pp. 715–728.

[22] A. Arcangeli, "AutoNUMA," May 2012. [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf

[23] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "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)*, vol. 3, 2022, pp. 742–755.