Datenbank-Spektrum
https://doi.org/10.1007/s13222-026-00562-8

SCHWERPUNKTBEITRAG

Active CXL Memory for Database Engines

Alberto Lerner[1] · **Gustavo Alonso**[2]

Received: 1 June 2026 / Accepted: 17 August 2026
© Gesellschaft für Informatik e.V. and Springer-Verlag GmbH Germany, part of Springer Nature 2026

Abstract
CXL offers the potential to increase the memory available to a server, both in terms of capacity and total bandwidth. However, recent measurements suggest that this expansion may introduce a critical drawback. CXL traffic can compete with conventional memory traffic on host-side interconnects, and the additional bandwidth may not always be accessible to the processor cores. In this paper, we argue that processing data on the CXL side of the memory, referred to as Near-Data Processing, may be a more effective way to use CXL in database engines. This approach is supported by CXL through a special class of devices, called Type 2, which combines memory that can be coherently shared with the host and local computational elements on the device. We propose strategies for redesigning existing database engines to leverage such devices and suggest targeted architectural changes, particularly in the buffer manager, that can enable the integration of multiple CXL-backed memory pools and process data on the CXL device side.

Keywords Database Systems · CXL Memory · Near-Data Processing

1 Introduction

CXL is an open interconnect standard that provides cache coherence and shared memory access between host processors and attached devices [4]. It supports a hierarchy of device types with different levels of sophistication and capabilities [5]. Of these, Type 3 devices, also called memory expanders, allow adding capacity beyond what the CPU memory controllers can support. This allows systems to scale memory size independently of CPU limitations. Some initial measurements and simulations indicate that the expanded memory bandwidth is similar and the latency only slightly higher than that observed when accessing memory across a NUMA design [6, 7]. Such measurements creates a compelling opportunity to redesign systems for larger capacity, particularly for memory-intensive database engines [8].

Type 3 devices' memory can be managed transparently to applications by either the Operating System (OS) or directly through the hardware. If using the OS, a possible strategy is Transparent Page Placement (TPP), which monitors page temperatures and migrates them between local DRAM and CXL memory as needed [9]. If the choice of managing the memory falls on the hardware, a possible strategy is using Intel's Flat Memory Mode, which interleaves local DRAM and CXL memory at cache-line granularity and exposes the combined memory as one, also making temperature-based migrations in the background [7]. Recent measurements on SAP HANA show that the hardware-managed path can match a DRAM-only baseline for analytic workloads even with CXL devices of relatively low bandwidth [10]. Either way, these examples use CXL as an additional, transparent memory tier.

The natural expectation, then, is that one can continue adding CXL memory to a system and that this will provide additional bandwidth and capacity to a database system. However, this expectation does not always hold. Type 3 devices expand memory capacity, but pulling all data to host cores eventually creates a bottleneck on the host CPU interconnect (Sect. 2).

Type 2 devices address this bottleneck by integrating compute capabilities directly onto the memory device alongside coherent storage. This gives query workers the option to execute on the device itself and process data locally, avoiding host interconnect traffic. To support this model, this paper proposes that database systems need to

✉ Alberto Lerner
lernera@computingflows.com

[1] Computing Flows, Fribourg, Switzerland

[2] ETH Zurich, Zurich, Switzerland

Published online: 12 September 2026

coordinate data placement across disaggregated host and CXL memory pools (Sect. 3).

A central element of our proposal is the expansion of the *Buffer Manager* (BM), the component in the database engine that mediates memory access for query workers. Typical implementations track in-memory data pages, fetch missing ones, and enforce replacement policies. In our design, the buffer manager is aware of CXL-backed memory pools, explicitly controlling where pages are allocated or evicted (Sect. 4). Interestingly, by controlling the data placement, and in particular data movement across host and device, the buffer manager can help more efficiently implementing some common query data flow patterns. For instance, we present a producer-consumer use case demonstrating direct data exchange between host- and device-based workers (Sect. 5).

This data-flow coordination applies not only between software workers but also between host workers and I/O hardware devices that function as Type 2 devices, such as CXL-enabled NICs or SSDs [11, 12]. When these devices act as data producers or consumers, data can enter and exit the system directly through memory requests rather than traditional network or storage I/O calls. This allows the buffer manager to coordinate data flows directly instead of issuing traditional I/O requests, opening new opportunities to integrate database systems with specialized hardware (Sect. 6).

In summary, our key contribution is extending the classic buffer manager interface to coordinate data placement across multiple CXL-backed memory pools. This expands the in-memory working set while directly preventing the host CPU interconnect from becoming a processing bottleneck.

2 Background and Motivation

The potential for interference is not merely hypothetical. Recent measurements demonstrate that contention can arise on the path connecting cores, host memory, and CXL-attached memory, as this latter scales. Mao et al. [13] report that the contended structure is neither the CXL link nor the memory controllers, but the Table of Request (TOR), an on-chip queue between cores and the last-level cache. Under interference, TOR occupancy increases significantly, while last-level cache miss latency remains roughly constant, indicating that the on-chip path between cores and cache rather than to the memory subsystem itself is under stress.

Such a negative pattern also appears in other scenarios. Li et al. [14] profile concurrent CXL streams and observe that the latency can rise by a factor of 4.3 under contention. Hou et al. [15] measure device-to-device PCIe traffic that crosses the host root complex, dropping from 9.8 GB/s to 2.3 GB/s when this traffic could otherwise stay inside a switch subtree. Agarwal et al. [16] show that even without CXL, a NIC's DMA throughput drops by roughly 15% when CPU cores load the memory bus, because the memory controller serves CPU and device requests in roughly first-come-first-served order. Across these measurements, the bottleneck sits on the host side of the device, at elements that both compute and CXL traffic have to cross. Adding CXL bandwidth increases the competing traffic.

This does not mean that the promise of scaling up cannot be realized. It means that adding memory while concentrating processing power on the host may not be the most efficient way to do so.

Two research directions in the database community suggest strategies that can achieve better scalability. The first is adopting *controlled data movement*, where the engine explicitly determines the placement and movement of data, rather than relying on page faults or access patterns to guide these decisions. This view appears in recent proposals for data flow architectures in databases [17], declarative interfaces for data movement between storage layers [18], and throughput-guided data movement in scale-up analytic engines [19].

The second is a shift toward *processing closer to memory*, where database operators execute on a device co-located with the data rather than on the host CPU. Earlier proposals established this direction for database engines using memory-centric devices [20]. More recent work has shown the value of the approach for memory in smart storage [21, 22], in specialized processing-in-memory devices [23], and in disaggregated memory with FPGA offload [24].

The central tenet of what we propose here is that controlled data movement and local processing represent promising but traditionally separate directions for scaling database systems. With the emergence of CXL, these strategies can now be unified within a single architectural framework.

To understand how CXL enables this unification, we need to consider the range of devices defined in the standard. In addition to memory expanders, CXL includes devices with computational capabilities. They are called

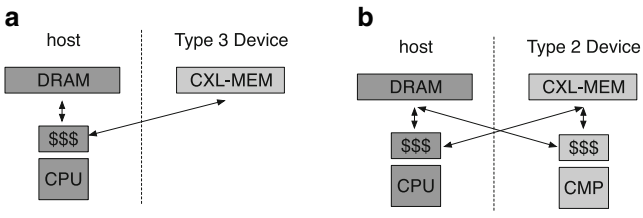


Fig. 1 Different CXL device types. **a** Type 3 devices offer additional memory that can be coherently cached by CPU thread; **b** Type 2 devices do so as well but can also cache data from the local or host's memory, also coherently

Type 2 devices and are easy to understand when compared to Type 3 devices. Figure 1 depicts both.

Type 3 devices provide memory that is managed by the CPU, as the device does not have compute capabilities. Being managed by the CPU only, cache memory from that device is fully coherent with memory cached from the host. Threads running on the host CPU can access memory addresses on either side indistinctively.

In contrast, Type 2 devices have local computing elements (CMP in Fig. 1b). They expose memory to the host as well, but coherency is now maintained cooperatively between host and device (in CXL 1.1 and 2.0) or entirely by the device (in CXL 3.x). Threads running on the host CPU will behave exactly the same as in a Type 3 device. The key difference is that threads or computations running on the device's CMP element may cache memory on either side coherently, just as their host counterparts do.

This enables processing closer to the expanded memory. If a page resides on a Type 2 device, the computational element in that device could behave as a query execution worker accessing local data faster, coherently—and avoiding the traffic that running all workers on the host would cause.

Interestingly, CXL can also help if the database system takes control of the data movement. The hardware machinery implementing the protocol is built to support low-latency, fine-grained data movements. For instance, this machinery can be used to overwrite the contents of an empty page frame with the contents of another frame. Figure 2 depicts how, by showing how to lay out a buffer manager's components in different portions of the available memory and using CXL commands that can implement larger DMA transfers. The page map can reside in the host DRAM and could index pages from different local page buffers. The buffer manager then becomes the place where the placement of data and the placement of the operator are decided together.

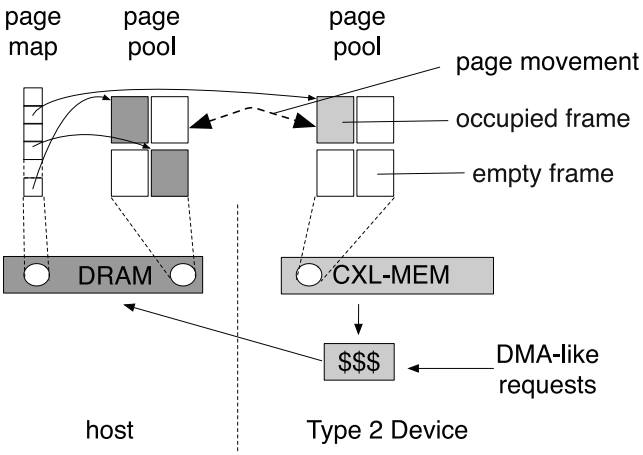


Fig. 2 A page migration leveraging CXL support: the coherency machinery can be co-opted as a DMA engine

Table 1 The Baseline Buffer Manager Contract

Operation	Contractual Responsibility
Fix	Resolve page and acquire access latch
Unfix	Release page reference and latch
MarkDirty	Flag modified frame for eventual writeback
Allocate	Reserve a free frame in memory
Evict	Select resident page to free its frame
Flush	Persist dirty frame contents to storage

To ground our proposal, we first recall the traditional abstraction that database systems use to interact with the storage subsystem: the buffer manager's functionality. As summarized in Table 1, the buffer manager presents a contract to query workers comprising six core operations [3]. These include fixing a page to make it resident, unfixing it after access, marking it dirty when modified, allocating a frame for new data, evicting a resident frame under memory pressure, and flushing a dirty page to storage. While modern storage engines vary significantly in how they implement this contract, the functionality presented to query workers has remained remarkably stable. Extending database engines to leverage disaggregated CXL memory requires expanding this operational contract rather than discarding it.

3 Multi-Pool Buffer Managers Regimes

We propose a multi-pool buffer manager that manages memory across host DRAM and the memory pools of one or more Type 2 devices. Its design hinges on two related questions. The first is where a page resides at any given moment. The second is whether the side that does not host the page may access it via CXL (without migration).

The combinations of answers define at least three *page access regimes*. Figure 3 illustrates these regimes along with a baseline in which the Type 3 device contributes memory but does not compute (Tiered in Fig. 3a). In this baseline, the CXL memory acts as an additional level in the memory hierarchy with less latency than storage.

In the first regime, a page may reside exclusively on one side (Single-Homed Exclusive Pages in Fig. 3b). This configuration expands the memory available and assumes that processing happens on the side that is housing a given page. As a result, pages may need to be migrated before they can be processed, but as discussed above, CXL can offer efficient mechanisms for the data transfers. This regime, as well as the others, helps alleviate contention on the host, since it allows a portion of the computations on the device without transferring the data.

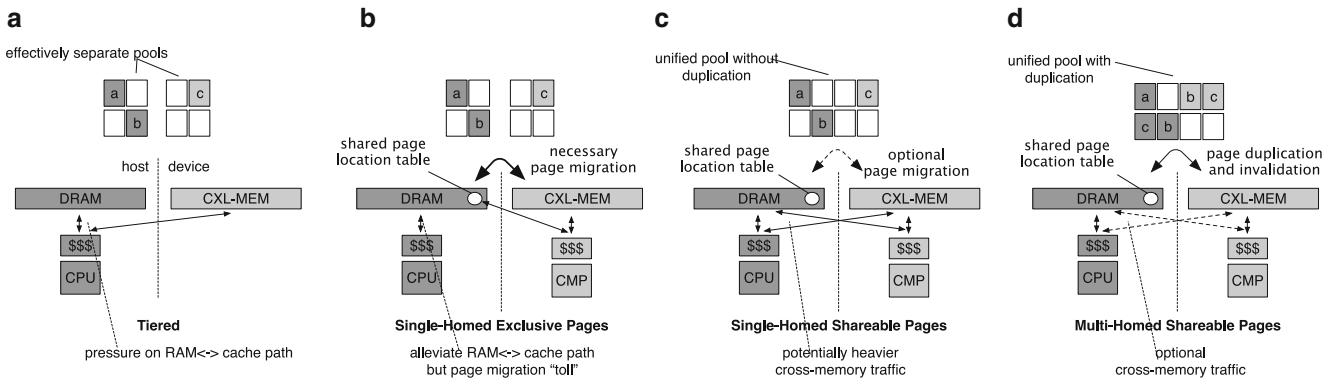


Fig. 3 Baseline and three different configurations for CXL memory. **a** Baseline where the CXL memory has no compute capacity; **b** Pages reside exclusively on one side and access across sides implies page migration; **c** Pages are placed in a unified memory space and access does not necessarily imply migration; **d** Unified memory pool with page duplication for lower latency access

In the second regime, a page may reside on one side while remaining accessible from the other side, making migration optional rather than necessary (Single-Homed Shareable Pages in Fig. 3c). This configuration takes advantage of the possibility of having a unified memory space where the memory on the CPU and the CXL device look like a continuous memory region to threads on either side, and the hardware transfers the data as needed directly to the caches. The approach offers the flexibility of full transparency, but it requires careful placement. Frequent cross-access may introduce interference.

In the third regime, a page may reside on both sides simultaneously, in which case the copies must remain consistent (Multi-Homed Shareable Pages in Fig. 3d). This configuration minimizes cross region traffic as it is optional whether to replicate a page or to leave it on only one side, but requires careful handling of updates.

The regimes outlined above resemble, in their concern for locality and partitioning, earlier work on NUMA-aware database deployments [25, 26]. However, the multi-pool setting presents distinct challenges. Unlike NUMA, CXL-based multi-pool environments involve heterogeneous memory pools, both in size and bandwidth. Moreover, the addressing used to access the pools may need translation on the device side. As a result, data placement and access patterns require new strategies that account for these differences. In contrast, multi-pools give more control of data placement to the engine than NUMA settings. This raises the question of how the buffer manager should expose such control.

4 Extending the Classic BM Interface

To enable this control, the buffer manager can be extended to support explicit page movement between memory pools. Traditional buffer managers operate on a single memory pool using the baseline contract described in Table 1. Disaggregation requires extending this model. We propose new operations that allow the engine to decide where a page should reside and when it should move across pools.

Our proposal is not the only recent attempt to extend buffer manager functionality. Much of the recent resurgence in this area, however, has focused on optimizing traditional implementations. LeanStore [1], Umbra [27], and vmcache [28] lower the cost of locating a page through pointer swizzling or by offloading translation to the MMU. In a similar spirit, LIPAH pairs each page reference with a frame hint, reaching the page directly when the hint is valid and consulting the mapping table only when it is not [29]. Predictive translation [2] anticipates these lookups rather than serving them on demand. Calico replaces the central hash table with a direct-indexed array [30]. Buffer management for tiered main memory reaches CXL but treats it as a passive tier behind the same interface [31]. These modern systems vary significantly in their implementation mechanics, replacing central page tables and manual latches with mechanisms like pointer swizzling or scoped guards. Nevertheless, every operation of the baseline contract retains a direct functional counterpart across them all.

These advances show the role of the buffer manager in improving the efficiency of manipulating a single memory pool. What we suggest here is to extend the buffer manager interface in an orthogonal direction, enabling the engine to manage multiple memory pools. In practice, we propose a set of new *primitives* that a multi-pool buffer manager can support, each representing a distinct form of control.

As a preliminary design, we propose four such primitives: *Move, Share*, and *Sync* are new, while the fourth, *Allocate*, is a location-aware variation of an existing operation. An engine need not implement all four, and each may be realized by more than one command. Our intent is to identify new forms of control and open an exploration space, rather than prescribe a single interface.

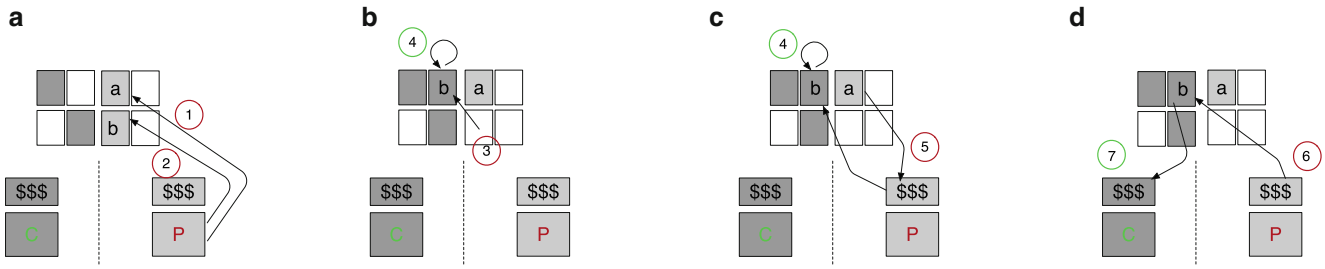


Fig. 4 A producer-consumer exchange across two pools. The host runs consumer C and the device runs producer P. The host and device each have a page pool and can cache from each other. **a** The producer fixes input page a ① and allocates a volatile output page b ②; **b** It then moves the empty b to the host as a metadata-only operation ③, whereupon the consumer can start spinning on b with sync ④; **c** The producer reads a and writes the result into b ⑤ while the consumer is waiting; **d** The producer unfixes b ⑥, the signal sync awaits, and the consumer reads b ⑦

- *Move* transfers a page from one pool to another. The engine may use this primitive to place a page next to the compute element that will operate on it, or to rebalance how pages are distributed across pools. When the destination frame is empty, the operation reduces to a metadata update, and otherwise copies the page contents into the destination pool.
- *Share* makes a page reachable from a pool other than the one that holds it, without relocating or duplicating it. A compute element on one side can then read a page that physically resides on the other, relying on the coherence that Type 2 devices provide. This avoids the cost of a move when both sides need only occasional access to the page, at the expense of the cross-pool traffic that each remote access generates. The implementation must provide a way to deal with the address translation necessary to access the same page from a device or from the host.
- *Sync* coordinates access to a page across pools, letting one side block until another signals a chosen event, such as having finished writing the page. Although *Sync* resembles a traditional latch, it operates on a different axis: it governs page visibility across pools rather than worker access modes (such as the shared, exclusive, or optimistic access used in systems like LeanStore) within a single pool. For instance, this primitive can provide the ordering that a produce-then-consume handoff requires, which the single-pool interface leaves the engine to arrange on its own. We will discuss an example of this arrangement shortly.
- *Allocate* reserves working memory, that is, pages that hold intermediate computation results rather than cached copies of stored data. Such pages have no backing store, so the manager neither loads them from storage nor writes them back. This is not strictly a new primitive. Some engines choose to allocate working memory this way, placing the scratch space under the same tracking and movement machinery as ordinary pages. In our case, this means that the page can be moved or shared as any other page.

These primitives expand the buffer manager's capabilities and enable more precise control over data placement and access.

5 A Producer-Consumer Example

A good way to showcase the possibilities of this design space is to show, at a high level, a buffer manager that supports exchanging data across pools in a producer-consumer fashion. This also illustrates how to leverage the desirable controlled data movement and processing closer to memory features we discussed above. Figure 4 depicts how the producer and consumer use a buffer manager with these commands to interact.

To describe the scenario, we introduce a set of simple commands it could use. The commands derive almost directly from the primitives in the previous section, but we fill in more details about how CXL can facilitate implementing these commands. We will adopt the Single-Homed Shareable Pages regime described in Fig. 3.

This buffer manager would contain four additional commands. The command *Move* uses CXL's ability to transfer data between the pools via CXL mechanisms, *e.g.,* non-temporal writes issued by caches on either side. This allows each side to overwrite data at a destination by using the contents of the source. We note that CXL operates at a cache-line granularity, so transferring a page takes several (fast and low-latency) cache-line moves, as shown in Fig. 2. We assume that the page will not be accessed in the destination as a side effect of the move command, even if CXL would allow it. This is an arbitrary restriction to make our example simple.

We, therefore, use an explicit *Share* command, issued by the producer, to allow other entities to see a page that will eventually become available. Once again, for simplic-

ity, we prevent the consumer from accessing it immediately, which would be possible via CXL coherency support, forcing the consumer to wait until the producer relinquishes the page. Lifting this restriction can enable interesting concurrent data manipulation opportunities.

This wait is an implementation of the *Sync* primitive. It is a blocking command that can use, for instance, a spin lock over an area that, thanks to CXL, the current owner of the page (who fixed it) shares. Lastly, we allow a *Allocate* to be issued with no address, which means that the page will be used for working memory and can be discarded when its fix count is zero. This handoff exemplifies the second axis of buffer management. Specifically, *Sync* coordinates page visibility across memory pools during cross-device execution rather than controlling concurrent access within a single pool.

A key advantage of this exchange is that neither side needs to implement specialized code for data movement; the buffer manager handles this transfer. Moreover, the two sides may consist of conventional threads, or the producer may be a compute element located near storage or the network, as we discuss shortly. Another advantage is that this simplicity adds minimal performance overhead. A page can be moved through CXL's low-latency coherent path rather than through a slower DMA transfer over PCIe.

Ultimately, the biggest promise with buffer pools that manage data movement and with scenarios that allow local computations in CXL devices is to avoid the issue raised in Sect. 2: interference. This approach keeps data and computation co-located, reducing unnecessary data movement across the system. Reading page a in our example occurs on the device, right next to where page a resides. Were the producer a host thread instead, it would have pulled page's a data across the CXL link, and into the host's cores interconnect to read it, loading the very element identified as the bottleneck.

6 Beyond Pagination

In the preceding example, both the producer and the consumer were threads. However, a Type 2 device is not limited to DRAM-based accelerator cards. The CXL specification allows devices such as NICs and SSDs to function as Type 2 devices, exposing memory and participating in cache coherence. When such a device acts as either the producer or the consumer, data can enter or exit the system directly via memory requests to the device, rather than through API calls.

This can further streamline data flow in an application. For example, a NIC can receive pages from the network into its own memory pool before the host accesses them, while an SSD can write pages from its pool to storage after the host has staged them. In this scenario, the buffer manager relinquishes two of its traditional responsibilities. First, it no longer performs page I/O, as the device itself manages data ingress and egress. Second, it does not translate block identifiers into memory locations to fetch pages, since the page arrives through the device.

Instead, the buffer manager assumes the role of coordinating data flows that originate or terminate at the device. This added responsibility can pay off even further if these devices can process data in transit, for example, by applying projection or selection near the network or storage before the host executes the rest of the query [17]. Multi-pool buffer managers on these specialized Type 2 devices open an interesting space for exploration.

7 Conclusion

The multi-pool design space we have outlined is no longer hypothetical. Commercial Type 2 devices that combine a host-accessible memory pool with substantial compute on the device side are entering the market. The Altera Agilex 7 M-Series [32] pairs an FPGA fabric with ARM cores and can expose DDR5 and HBM2e memory to the host. The Marvell Structera A [33] packages sixteen ARM Neoverse V2 cores alongside four DDR5 memory channels and roughly 200 GB/s of memory bandwidth. These devices make it possible to prototype the regimes outlined above and measure their tradeoffs.

References

1. Leis V, Haubenschild M, Kemper A, Neumann T (2018) Leanstore: In-memory data management beyond main memory. 34th IEEE International Conference on Data Engineering. https://doi.org/10.1109/ICDE.2018.00026
2. Zinsmeister M, Nguyen L, Leis V, Neumann T (2026) Predictive translation: High-performance buffer management without the trade-offs. Proc. ACM Manag. Data 4(1):64–16416. https://doi.org/10.1145/3786678
3. Effelsberg W, Härder T (1984) Principles of database buffer management. ACM Trans. Database Syst 9(4):560–595. https://doi.org/10.1145/31858.31859
4. CXL Consortium (2023) Compute Express Link (CXL) Specification. Technical Specification Revision 3.1, Compute Express Link Consortium.
5. Das Sharma D, Blankenship R, Berger DS (2024) An introduction to the Compute Express Link (CXL) interconnect. ACM Computing Surveys 56(10):1–37. https://doi.org/10.1145/3669900
6. Li H, Berger DS, Hsu L, Ernst D, Zardoshti P, Novakovic S, Shah M, Rajadnya S, Lee S, Agarwal I, Hill MD, Fontoura M, Bianchini R (2023) Pond: CXL-based memory pooling systems for cloud platforms. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS. https://doi.org/10.1145/3575693.3578835
7. Zhong, Y., Berger, D.S., Waldspurger, C.A., Wee, R., Agarwal, I., Agarwal, R., Hady, F., Kumar, K., Hill, M.D., Chowdhury, M.,

Cidon, A. (2024) Managing memory tiers with CXL in virtualized environments. In: 18th USENIX Symposium on Operating Systems Design and Implementation, OSDI. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong
8. Lerner A, Alonso G (2024) CXL and the return of scale-up database engines. Proc. VLDB Endow 17(10):2568–2575. https://doi.org/10.14778/3675034.3675047
9. Maruf HA, Wang H, Dhanotia A, Weiner J, Agarwal N, Bhattacharya P, Petersen C, Chowdhury M, Kanaujia SO, Chauhan P (2023) TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS. https://doi.org/10.1145/3582016.3582063
10. Ahn M, Willhalm T, Lee D, May N, Kim J, Ritter D, Rebholz O (2025) Exploiting locality in flat memory with CXL for in-memory database management systems. Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN. https://doi.org/10.1145/3736227.3736230
11. Lee, S., Lerner, A., Bonnet, P., Cudré-Mauroux, P.: Database kernels (2024) Seamless integration of database systems and fast storage via CXL. In: 14th Conference on Innovative Data Systems Research CIDR. https://vldb.org/cidrdb/2024/database-kernels-seamless-integration-of-database-systems-and-fast-storage-via-cxl.html
12. Ji H, Yuan Y, Zhou Y, Jeong I, Wang R, Agarwal S, Kim NS (2025) Re-architecting end-host networking with CXL: coherence, memory, and offloading. Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture MICRO. https://doi.org/10.1145/3725843.3756102
13. Mao S, Luo J, Li Y, Zhou J, Zhang W, Liu Z, Ma T, Deng S (2025) CXL-Interplay: unraveling and characterizing CXL interference in modern computer systems. 62nd ACM/IEEE Design Automation Conference, DAC. https://doi.org/10.1109/DAC63849.2025.11132607
14. Li X, Guo Z, Bai Y, Ketkar M, Wilkinson H, Liu M (2025) Understanding and profiling cxl.mem using pathfinder. Proceedings of the ACM SIGCOMM 2025 Conference, SIGCOMM. https://doi.org/10.1145/3718958.3750479
15. Hou, W., Zhang, J., Wang, Z., Liu, M. (2024) Understanding routable pcie performance for composable infrastructures. In: 21st USENIX Symposium on Networked Systems Design and Implementation, NSDI. https://www.usenix.org/conference/nsdi24/presentation/hou
16. Agarwal S, Agarwal R, Montazeri B, Moshref M, Elmeleegy K, Rizzo L, Kruijf MA, Kumar G, Ratnasamy S, Culler DE, Vahdat A (2022) Understanding host interconnect congestion. Proceedings of the 21st ACM Workshop on Hot Topics in Networks, HotNets. https://doi.org/10.1145/3563766.3564110
17. Lerner A, Alonso G (2024) Data flow architectures for data processing on modern hardware. 40th IEEE International Conference on Data Engineering. https://doi.org/10.1109/ICDE60146.2024.00439
18. Vogel, L., Ritter, D., Porobic, D., Tözün, P., Wang, T., Lerner, A. (2023) Data pipes: Declarative control over data movement. In: 13th Conference on Innovative Data Systems Research. https://www.cidrdb.org/cidr2023/papers/p55-vogel.pdf
19. Nicholson, H., Raza, A., Sanca, V., Ailamaki, A.: A-scan (2026) Efficient scale-up analytics via throughput-guided data movement. In: 42nd IEEE International Conference on Data Engineering.
20. Babarinsa OO, Idreos S (2015) JAFAR: near-data processing for databases. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. https://doi.org/10.1145/2723372.2764942
21. Woods L, Z I, Alonso G (2014) Ibex - an intelligent storage engine with support for advanced SQL off-loading. Proc. VLDB Endow 7(11):963–974. https://doi.org/10.14778/2732967.2732972
22. Bernhardt A, Tamimi S, Stock F, Koch A, Petrov I (2025) Update NDP: on offloading modifications to smart storage with transactional guarantees in near-data processing DBMS. ACM Trans. Database Syst 51:11–11145. https://doi.org/10.1145/3774753
23. Baumstark A, Jibril MA, Sattler K (2023) Accelerating large table scan using processing-in-memory technology. Datenbank-Spektrum 23(3):199–209. https://doi.org/10.1007/s13222-023-00456-z
24. Korolija, D., Koutsoukos, D., Keeton, K., Taranov, K., Milojicic, D.S., Alonso, G. (2022) Farview: Disaggregated memory with operator off-loading for database engines. In: 12th Conference on Innovative Data Systems Research. https://www.cidrdb.org/cidr2022/papers/p11-korolija.pdf
25. Porobic D, Pandis I, Branco M, Tözün P, Ailamaki A (2012) OLTP on hardware islands. Proc. VLDB Endow 5(11):1447–1458. https://doi.org/10.14778/2350229.2350260
26. Porobic D, Liarou E, Tözün P, Ailamaki A (2014) Atrapos: Adaptive transaction processing on hardware islands. IEEE 30th International Conference on Data Engineering, ICDE. https://doi.org/10.1109/ICDE.2014.6816692
27. Neumann, T., Freitag, M.J. (2020) Umbra: A disk-based system with in-memory performance. In: 10th Conference on Innovative Data Systems Research. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf
28. Leis V, Alhomssi A, Ziegler T, Loeck Y, Dietrich C (2023) Virtual-memory assisted buffer management. Proc. ACM Manag. Data 1(1):7–1725. https://doi.org/10.1145/3588687
29. Otaki, R., Chang, J.H., Benello, C., Elmore, A.J., Graefe, G. (2025) Resource-adaptive query execution with paged memory management. In: 15th Conference on Innovative Data Systems Research, CIDR. https://www.vldb.org/cidrdb/papers/2025/p2-otaki.pdf
30. Zhou X, Hu J, Pavlo A, Stonebraker M (2026) Making array-based translation practical for modern, high-performance buffer management. CoRR. https://doi.org/10.48550/arXiv.2604.00423
31. Hao X, Zhou X, Yu X, Stonebraker M (2024) Towards buffer management with tiered main memory. Proc. ACM Manag. Data 2(1):31–13126. https://doi.org/10.1145/3639286
32. Altera (2024) Agilex 7 FPGA M-Series Development Kit - HBM2e Edition. https://www.altera.com/products/devkit/po-3207/agilex-7-fpga-m-series-development-kit-hbm2e-edition-3x-f-tile-1x-r-tile
33. Marvell (2024) Structera A Near-Memory Accelerator Product Brief. https://www.marvell.com/content/dam/marvell/en/public-collateral/assets/marvell-structera-a-2504-near-memory-accelerator-product-brief.pdf

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.