Mainframe-Style Channel Controllers for Modern Disaggregated Memory Systems

Zikai Liu
zikai.liu@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Jasmin Schult
jasmin.schult@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Pengcheng Xu
pengcheng.xu@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Timothy Roscoe
troscoe@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

Despite the promise of alleviating the main memory bottleneck, and the existence of commercial hardware implementations, techniques for *Near-Data Processing* have seen relatively little real-world deployment. The idea has received renewed interest with the appearance of disaggregated or "far" memory, for example in the use of CXL memory pools.

However, we argue that the lack of a clear OS-centric abstraction of Near-Data Processing is a major barrier to adoption of the technology. Inspired by the *channel controllers* which interface the CPU to disk drives in mainframe systems, we propose *memory channel controllers* as a convenient, portable, and virtualizable abstraction of Near-Data Processing for modern disaggregated memory systems.

In addition to providing a clean abstraction that enables OS integration while requiring no changes to CPU architecture, memory channel controllers incorporate another key innovation: they exploit the cache coherence provided by emerging interconnects to provide a much richer programming model, with more fine-grained interaction, than has been possible with existing designs.

CCS Concepts

• **Software and its engineering** → **Operating systems**;
• **Computer systems organization** → **Processors and memory architectures**; *Distributed architectures*; • **Hardware** → *Hardware accelerators*.

Keywords

Near-data processing, operating systems, cache coherence, far memory, disaggregation, offloading, accelerators.

1 Introduction

Limited bandwidth to main memory, and the high latency of main memory accesses relative to CPU frequency, have long been performance bottlenecks despite the widespread use of techniques to hide or reduce memory access times (caches, prefetchers, multi-threaded cores, out-of-order execution, etc.). This in turn has led to a long line of architecture research on Processing In Memory (PIM), Near-Memory Processing (NMP), or Near-Data Processing (NDP), whereby processing elements are placed close to main memory and act as offload engines for some CPU tasks. However, to date, very little of this work has found its way to product, and almost nothing to widespread deployment.

Recently, the deployment of *disaggregated*, *pooled*, or *far memory*, often via new interconnect protocols like Compute eXpress Link (CXL) [15], has given new impetus to this idea, since far memory incurs significantly higher access latency and delivers lower throughput than local DRAM [33, 34, 51]. For instance, Marvell recently introduced Structera A, a series of near-memory accelerators with Arm Neoverse V2 cores positioned next to CXL-attached memory [38].

We contend that the failure of NDP (which we will henceforth use as an umbrella term encompassing PIM, NMP, and related techniques) to impact practice is due to the lack of an OS-centric perspective on the technique [3]. Such a perspective on NDP would balance, on the one hand, hardware

constraints and improvements in application efficiency with, on the other, the oft-neglected requirements for complete realistic systems: secure multiplexing, resource management, virtualization, portable abstractions, and so on.

In this paper we develop such an OS-centric view and its implications, focusing on three facets. The first is the **application model**: the abstractions presented to application writers. We argue these abstractions should not only be easily usable (by both application programmers and compiler writers) and efficient, but also portable, fully virtualized, and free from arbitrary resource limitations.

Second is the **system software model**, in other words the non-functional system-wide properties we care about: NDP should preserve the application's existing security and isolation properties, applications using it should not experience livelock or starvation, etc.

Finally, the **hardware design** should follow the requirements of the application and system software models. To date, most proposals for NDP have been largely bottom up, with the broader issues left unresolved. This has left it unclear which hardware designs do or do not make sense in a real computer system with multiple tenants. Moreover, given the nature of the hardware market, it is desirable to avoid intrusive changes to CPU architecture: the latter break backward compatibility and limit deployment. As we argue in this paper, such architectural changes are unnecessary.

Based on this perspective, we take inspiration from an old idea: *channel controllers*. In additional to the main processors, IBM mainframes have long had processors dedicated to I/O operations [25, 32, 41]. These processors are typically at least as powerful as the CPU itself and come with a clean programming abstraction, the *channel program* [26, 41]. Applications like the DB2 relational database include highly-tailored channel programs as part of the main executable. Under the coordination of the OS, mainframe channel controllers can serve multiple applications simultaneously and dramatically cut the overhead of accessing disk storage.

We revisit this idea but in a very different context: disaggregated memory systems rather than disks, and the use of cache-coherent access to far memory (via interconnects such as CXL, or CCIX [6]) for both data transfer *and* a control interface to the remote device. This permits fine-grained, low-latency synchronization between an application and an NDP accelerator [22]. We present both a clean abstraction of such a *memory channel controller (MCC)*, and a combined hardware/software design that can efficiently provide such an abstraction in the context of a modern kernel-based OS.

2 Background: the memory bottleneck

Remotely attached, pooled DRAM attached via new interconnects like CXL are an attractive proposition for data centers, providing elastic scaling, extended capacity, and even reuse of older DRAM silicon.

However, such flexibility and cost saving comes with a price: disaggregated memory exhibits higher latency and lower bandwidth than local DRAM. Evaluations on existing CXL memory devices suggest latency from 150 to 400ns and bandwidth from 18 to 52GB/s [33, 34, 51]. Moreover, performance *variance* is also higher compared to local DRAM. CXL switches additionally introduce a per-hop latency of 200 to 400ns [34]. This leads to performance characteristics which are very different from a classical "balanced system" [17], and can significantly impact a range of applications [33, 34].

At the same time, there are compelling reasons to adopt disaggregated memory for applications which require rapid random access to large amounts of data, such as in-memory databases. Such databases can store more data on a large disaggregated memory pool and share it with multiple compute nodes for scalability. However, query execution is typically bottlenecked on data transfer between the far memory and the CPU cores [34, 61]. The same argument applies to graph processing workloads: disaggregated memory allows storing and sharing larger graphs, but many graph algorithms are highly sensitive to memory latency [36, 60] – indeed, even local memory is typically a bottleneck for these algorithms, and disaggregated memory amplifies the problem.

This has led to renewed interest in the long-standing research area of NDP (broadly construed). For example, PIM systems have been proposed for database acceleration [4, 21], graph processing [1, 63] and other workloads. In those systems, computing units are integrated with DRAM chips and can operate on individual DRAM rows and latches. This minimizes the distance that data must be moved, but imposes strong, hardware-specific constraints on how computation can be performed. For example, in UPMEM (a commercial PIM device), each DRAM processing unit (DPU) can only access a fixed 64MB DRAM slice. This requires complex software on the CPU to steer the data flow to and from DPUs [21]. From an OS-centric perspective, the memory and computing resources are too closely-coupled to be virtualizable or to provide anything more than coarse-grained inter-application isolation. Instead, the application developer is required to partition the data for a specific hardware platform. Virtualizing and multiplexing UPMEM has been suggested but at a granularity of multiple gigabytes [52].

Other NMP systems are less architecturally restrictive. Lockerman *et al.* [35] suggest adding compute units *throughout* the memory hierarchy. A downside with this approach is the pervasive hardware changes it entails. The additional die area is small, but the end-to-end usability additionally requires hardware verification (on every changed component), and the issue of secure multiplexing the new compute resources is left unaddressed.
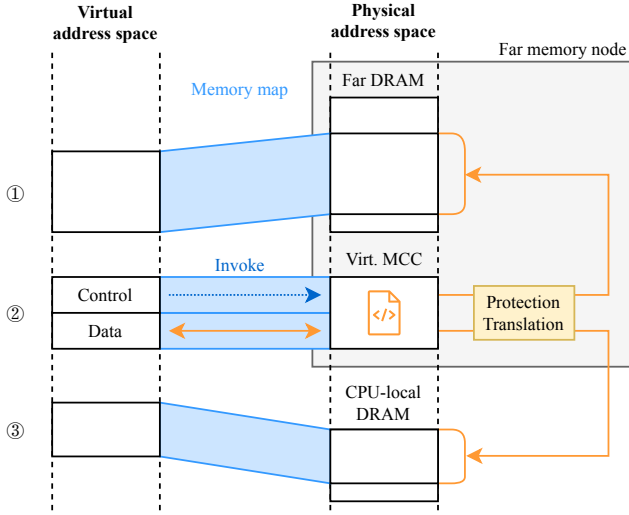
Figure 1: The far memory and MCC abstraction

We are not the first one to notice those problems. Gao *et al.* [19] and Ghose *et al.* [20] observe the gaps in address translation, memory protection and isolation functionality. Barbalace *et al.* [3] additionally discuss the problems in scheduling and the programming model, and call for better runtime and OS support. More recently, Ham *et al.* propose M^{2}NDP [22], which focuses on NDP for CXL-based disaggregated memory, emphasizing the limited hardware changes required as an important factor.

The NDP landscape, including recent work using CXL-attached memory [22, 23, 27, 48] shows a wide range of system designs, with divergent and largely incompatible programming models and constraints on secure multiplexing. We build on this work, but by taking an OS-centric perspective we hope to create more broadly practical systems, particularly in the context of disaggregated far memory.

3 Design overview

We now discuss our proposals for abstracting NDP in a portable, multiplexed, usable manner. A good interface makes the task of the application developer easier by providing high-level abstractions, without compromising performance by allowing the underlying system software and hardware to work in the most efficient way possible. It should also allow a programmer (or code generator) to reason about the performance impact of using the interface.

A key feature of OS abstractions (like virtual memory, files, or sockets) is that they impose no arbitrary limits on usage: modulo complete (and rare) resource exhaustion, a program is always free to create a new file, extend an existing one, use more virtual memory, etc. When acquiring processing resources close to far memory, this means that a *user program*

should not be limited by a fixed number of hardware units. This feature can also be viewed as one case of a general notion of *portability*: code written to use one hardware platform should, ideally, run correctly on a different platform.

These requirements lead us to abstract NDP processes as a set of virtual, dynamically instantiated processors (MCCs) close to memory, with a standardized interface. In order to both take advantage of existing OS abstractions, and to exploit the new possibilities presented by cache-coherent interconnects for fine-grained interaction between CPU threads and NDP resources, we build the MCCs abstraction over a process' virtual address space, as shown in Figure 1.

First, application code on the CPU might have direct access to a region of far memory mapped (via the MMU) to a contiguous region of the virtual address space (①). Access to far memory in this way completely bypasses NDP resources, but nevertheless this illustrates a key design decision: different remote memory nodes map to different regions of the virtual address space, and thus *the physical location of memory is explicit in the virtual address space layout.*

This is in stark contrast to transparent tiered memory systems like Intel Flat Memory Mode (FMM) [62], TPP [37] and M5 [50], where data placement is not exposed to the applications, but similar to that of M^{2}NDP [22] and CTXNL [55].

While there may be advantages to hiding memory properties from legacy applications which simply want to exploit expanded memory, efficient use of NDP strongly motivates making the distributed nature of far memory explicit. For memory-intensive applications like databases and graph processing, software has a better knowledge of data placement and access patterns, while hardware can only speculate. Past experience suggests transparent features like FMM end up being bypassed: for example, DBMSs generally disable Transparent Huge Page (THP) support [46] and graph applications manage NUMA memory explicitly [60].

Second, each MCC occupies its own region of virtual address space (②), and a user program communicates with its private MCCs using memory operations on these regions, as described in the next section. The control and data flow are indicated by the two arrows in the region in Figure 1.

Finally, each MCC has the ability to access both far memory and host-local memory belonging to its application in a conventional way using Direct Memory Access (DMA) operations (③), illustrated by the other two orange arrows.

An application wishing to use NDP with far memory first acquires access to the relevant region(s) (for example, on Linux via a variant of mmap()), and then asks the kernel to instantiate one or more MCCs, resulting in the creation of a new region (②) of virtual address space for each one. Since each MCC is private to an application, and its access rights are limited by the OS and hardware to the application's

virtual address space, the application can assume protection and operate as if it had exclusive access.

4 The MCC abstraction

The central contribution of this work is the nature of the interface to an MCC over virtual memory accesses. The MCC memory region is divided into two areas, one for control, and one for data. The control area allows the application to configure the MCC directly over Memory-Mapped I/O (MMIO) without involving the local kernel, including downloading a *channel programs* to the MCC.

A channel program (CP) differs significantly from previous models for programming NDP accelerators. In a typical existing approach, the NDP resource is given a (potentially lengthy) task to perform (such as zeroing memory, or materializing a view in an in-memory database). It runs to completion, potentially reading and writing both local (to itself) memory and host memory.

The CP programming model, however, also includes ongoing interactions with the host application, which occur via transactions using the cache coherence protocol in the data area. For example, a CP might *stream* results packed in cache lines directly to the cache running the application, using coherence for synchronization and message passing as in [47], or present an ongoing query-style interface to a remote data structure using cache-line-sized reads and writes as in [22]. The MCC abstraction therefore exposes, in the form of a CP, a much richer programming model to applications.

One advantage of this approach is performance: it is hard to beat the coherence protocol for latency in transferring data units up to a few kilobytes in size, and it also eliminates the overhead of setting up a new "task" for each NDP operation to be performed by the application.

In addition, it provides a *logical view* of data produced by the MCC. The application issues loads and stores to this region, but instead of accessing physical memory, the CP generates responses programmatically at runtime.

At the same time, it avoids the portability and compatibility issues, and also the security risks, that accompany extending the processor architecture to communicate with far memory, as in Intel's new AiA interface [2, 24].

The precise semantics for CPs is, at this point, an open question, but a promising approach is, at the lowest level, an event-driven model where the events include the arrival of coherence messages from the host (typically triggered by load and store operations in application code) and completions of local operations to access DRAM. However, the programming model exposed to application writers should clearly hide most of these details.

Another open question is whether an MCC should be restricted to accessing memory on a single far memory node
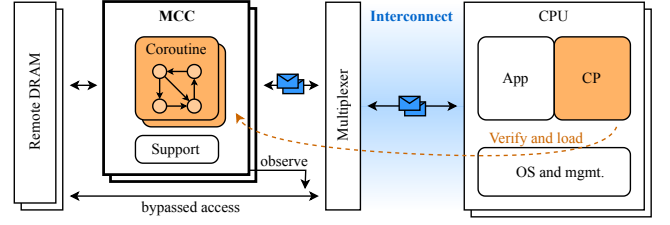

Figure 2: System architecture

(typically, the one where the hardware that it runs on is located), or if should see anything accessible from the application's virtual address space. In the latter case, we argue it is still important that each MCC has an *affinity* which specifies what memory is local to the MCC. When the system scales to multiple memory nodes, data placement strategies become relevant for reducing data movement across nodes (potentially with additional latency due to interconnect switches).

5 System design issues

We now discuss implementation issues in implementing the MCCs abstraction, summarized shown in Figure 2.

5.1 Assumptions on the interconnects

The system is designed around the emerging interconnects with memory access semantics. While we do not bound the design to a specific protocol, we make several assumptions.

First, the interconnect is message-based and encodes **memory transactions in a fixed-size granule** (usually a cache line). This is the case for CXL, Cache Coherent Interconnect for Accelerators (CCIX), and the Enzian Coherence Interface (ECI) on the Enzian research platform [14, 45] we are using to prototype the design. In contrast, PCIe, originally designed for peripheral devices with limited intelligence, is oriented heavily towards device-initiated bulk DMA transfers and word-sized CPU-initiated programmed I/O, which is a poor fit for encoding memory transactions.

Furthermore, we focus on protocols that allow **symmetric coherency**, meaning each party can actively control the cache line ownership symmetrically. This is *not* the case for the bias-based CXL.cache protocol, where a device needs to query the host CPU to resolve the coherency, incurring an extra round-trip delay. CXL.mem 3.0 *does* have this property by including a back-invalidation channel, allowing the device to independently resolve coherence with a directory (or "snoop filter" in CXL terms). However, to date no implementations yet exist. However, ECI, originally designed as a coherence interconnect for CPU sockets, also adopts this model, with performance [14, 45] that is comparable with CXL.

Symmetric coherence is important for our design as it allows MCC to perform fine-grained data movement independently. However, we note that the need for coherence *mechanisms* does not imply *full coherency* on the whole memory space. With the knowledge of high-level memory access patterns, MCC and CP can eliminate unnecessary coherence traffic while maintaining application correctness.

5.2 MCC execution environment

The physical MCC complex consists of general-purpose processors (albeit not as powerful as mainframe channel controllers) and a set of supporting units: a fast scratchpad memory, a split-phrase copy engine between scratchpad and DRAM, and streaming interfaces to coherence through which the processor receives and responds to coherence messages. The scratchpad can be accessed locally in a single cycle. Between it and the DRAMs, data movement is performed through the copy engine under the instruction of the processor. The transfer can be initiated in one cycle and is asynchronous, allowing the CP to hide the data movement latency with proper instruction scheduling.

A key implication of our OS-centric abstraction is that it is fully virtual: there will be a fixed number of physical processors on a node for running CPs, but an unbounded number of MCCs, each of which is dedicated to an application. This imposes several requirements for the processor and the supporting system software.

First, a physical processor on a far memory node must **multiplex** a number of MCCs, which also implies that it must perform **scheduling**. Given that CPs are specified at a high level, this scheduling need not be preemptive – effectively the physical processor can cooperatively schedule CP interpreters as *coroutines* without sacrificing performance. While relatively simple scheduling might provide sufficient guarantees against starvation under load, we might consider more complex policies (such as weighted fair queuing for memory and interconnect access).

In addition to multiplexing, MCCs must provide **isolation and memory protection**. Effectively, this means that the physical processors on the far memory node must be kept up-to-date with the virtual address spaces of applications on the host CPU.

At first sight, this would seem to introduce a high system overhead, but we make two observations. First, the entire address space need not be replicated on the channel controller, and restricting far memory to contiguous mappings further simplifies the metadata that must be kept consistent, effectively segmentation [5, 11]. Second, we are encouraged by recent proposals for fine-grained synchronization between CPU-based OSes and network interfaces on a coherent interconnect [57], which suggest that even scheduling state

can be efficiently shared using the same techniques [22, 47] we propose for data transfer. Furthermore, extending on the same interface to the interconnect, MCC can also observe memory requests from CPUs, which enables another group of applications (section 6).

We note that, while we are proposing a minimum level of functionality on the far memory nodes (i.e. processors capable of scheduling multiple channel programs, some memory protection, and low-level access to the coherence protocol), we do *not* require any changes to the CPU architecture or memory interface, nor to the interconnect protocols. The cost of the additional hardware can be amortized by the forthcoming far memory controllers that need to be designed and manufactured anyway.

5.3 Preliminary CP design

The CP model design is ongoing work. We are experimenting with a design where the CP polls coherence messages on the MCC-specific region (② in Figure 1), performs computations, and responds with coherence messages if needed.

When a reply message is needed (e.g. memory read transactions), the CP is on the critical path. If it does not produce an output in time, the interconnect can be deadlocked, which makes a hard real-time problem. This is challenging, but we believe it is solvable, following the line of hard real-time systems that have already been built (e.g. avionic flight controllers). Specific to our design, there are two additional advantages. First, latencies of operations, such as fetching a cache line from the DRAM controller, are highly predictable. Second, interconnects are typically lenient about timeout. Both CXL and ECI allow timeout up to the millisecond scale.

When considering the MCC virtualization, additional handling is needed. For example, application and CP can be **co-scheduled** to ensure no in-flight memory transaction can be issued when the CP is descheduled. If the physical MCCs are overloaded, using the CPU to simulate the MCC may be a feasible option.

Above this, we are also developing a **safe high-level programming model**, which needs to hide the hardware execution environment discussed above and ensure it is used in a safe manner, while still allowing the application developer to specify the data movement explicitly. One notable idea is to use a model like DataPipes [53], where data locations are specified explicitly but the low-level memory operations are abstracted away. The transformation from a high-level CP to a correct compiled CP requires a combination of language design, compiler checks, and run-time verification.

6 Mapping workloads to the system

To illustrate how MCCs works in practice, consider calculating common neighbors in a graph (LinkedIn uses this to find
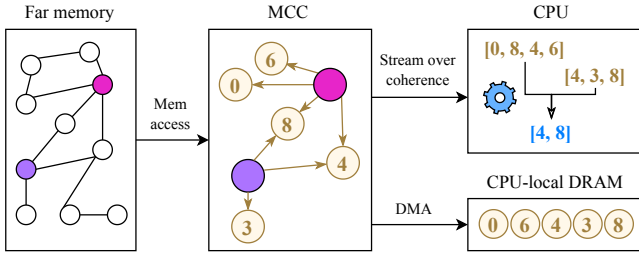
Figure 3: *n*-hop common neighbor pipeline

n-hop common connections between users [54]). Figure 3 shows a way to map the processing pipeline using an MCC:

- The complete social graph resides in far memory.
- The CP traverses the graph from each source node specified by the CPU and gathers a list of (possibly indirect) neighbors.
- This list of node IDs is streamed to CPU caches and then registers over the coherence protocol.
- The second stage (list intersection) is perform on the CPU, which now has excellent data locality.
- In addition, if the CPU needs to access the auxiliary node information, the CP can exercise the DMA capability to copy those data to the CPU-local memory.

Like many **graph processing workloads**, this is a good fit for an MCC since it has relatively low computation per node but highly irregular and unpredictable memory access patterns [36, 60], and so benefits from the reduced latency from an MCC to far memory. Other examples include online and query-heavy graph processing workloads like graph traversal (e.g. BFS), single-source shortest path, k-nearest neighbors, and pattern matching. These workloads, in their practical use-cases, tend to have real-time requirements of low latency and high throughput [9, 54], which motivates the usage of disaggregated memory and MCC.

In contrast, workloads like PageRank [42] and triangle counting are less good fits for the system. They require massive computation and a high degree of parallelism, which is not a key strength of the MCC approach. In practice, these workloads benefit more from cluster-based offline parallel processing, and existing batch processing systems work well.

The example above is one way to use an MCC – splitting a linear pipeline between MCC and CPU. Another approach is to perform smart prefetching, an idea well explored by architecture research [49, 58]. MCC, being a generous-purpose processor, is likely to be somewhat slower than hardware prefetchers, but much more flexible with regard to data layouts. Graph applications can bundle CPs that understand the in-memory format of the graph and perform tailored prefetch, while CPUs perform computations in parallel.

Another class of workload is **in-memory databases**. Unlike graph processing, databases use extensive runtime information on their own data access patterns and computations. Previous work [28, 30, 56] has shown that by pushing query operators closer to data drastically reduces the memory bottleneck. Korolija *et al.* pioneered offloading query operators to remote memory in Farview [30] using RDMA-based access with an FPGA as the NDP unit. Such an approach would map well to an MCC.

Database management systems can include CPs or synthesize them at query time to perform computations near data and/or steer data movement, much as IBM DB2 used mainframe channel controllers. Moreover, a CPs executing part of the operator graph can exploit different data transfer paradigms to interface with software. Bulk transfers of large data sets (e.g. with low selectivity and/or using eager materialization) benefit from DMA to CPU-attached memory, while when operators on a CPU core can consume intermediate tuples in real time, streaming to the CPU cache over the coherent interconnect is more efficient.

These exercise the full range of computation and communication mechanisms available to MCCs. However, there are also simpler workloads that utilize a smaller feature set. For example, **bulk memory copying and clearing** is more efficient on a MCC due to its proximity to far memory, and is important for huge page zeroing [43] and copy-on-write [29], hypervisor paging [31, 44], VM migration [12, 13], and replication for in-memory databases [39].

Mapping this workload to MCC is intuitive: a simple CP is invoked by the CPU and the MCC performs the memory operations asynchronously. The CPs can be parameterized, and the host application can specify the memory regions at the invocation time using the control channel. We expect similar performance benefits to previous work [29, 59].

Finally, an MCC can observe memory accesses and assemble fine-grained **memory access statistics** for application software. Such information can be used for hot page migration in tired memory systems [18, 37, 50], Garbage Collection (GC) [8, 10, 16] and profile-guided optimizations [7, 40].

7 Status and conclusion

An OS perspective on NDP calls for clean, portable, and usable abstractions to applications, while providing inter-application safety, scheduling, and virtualization. It also helps in mapping the hardware design space that makes sense in the context of a complete system.

By prototyping our ideas on a real hardware platform, we expect to establish the programming models, isolation properties, and application areas that make sense for mitigating the overheads of far memory.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. doi:10.1145/2749469.2750386

[2] Arjan van de Ven. 2024. VFIO: Add the SPR_DSA and SPR_IAX Devices to the Denylist. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95feb3160eef

[3] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 56–61. doi:10.1145/3102980.3102990

[4] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Processing-in-Memory for Databases: Query Processing and Data Transfer. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 107–111. doi:10.1145/3592980.3595323

[5] A. Bensoussan, C. T. Clingen, and R. C. Daley. 1972. The Multics Virtual Memory: Concepts and Design. *Commun. ACM* 15, 5 (May 1972), 308–318. doi:10.1145/355602.361306

[6] CCIX Consortium, Inc. 2019. *CCIX Base Specification Revision 1.0a Version 1.0 for Evaluation*. Technical Report. 346 pages.

[7] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. Association for Computing Machinery, New York, NY, USA, 12–23. doi:10.1145/2854038.2854044

[8] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-Guided Proactive Garbage Collection for Locality Optimization. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 332–340. doi:10.1145/1133981.1134021

[9] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proc. VLDB Endow.* 15, 9 (May 2022), 1965–1977. doi:10.14778/3538598.3538616

[10] Trishul M. Chilimbi and James R. Larus. 1998. Using Generational Garbage Collection to Implement Cache-Conscious Data Placement. *SIGPLAN Not.* 34, 3 (Oct. 1998), 37–48. doi:10.1145/301589.286865

[11] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. 1999. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 140–153. doi:10.1145/319151.319161

[12] Anita Choudhary, Mahesh Chandra Govil, Girdhari Singh, Lalit K. Awasthi, Emmanuel S. Pilli, and Divya Kapil. 2017. A Critical Survey of Live Virtual Machine Migration Techniques. *J. Cloud Comput.* 6, 1 (Dec. 2017), 92:1–92:41.

[13] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, USA, 273–286.

[14] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 434–451. doi:10.1145/3503222.3507742

[15] Compute Express Link Consortium, Inc. 2023. *Compute Express Link Specification Revision 3.1*. Technical Report. 1166 pages. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf

[16] Robert Courts. 1988. Improving Locality of Reference in a Garbage-Collecting Memory Management System. *Commun. ACM* 31, 9 (Sept. 1988), 1128–1138. doi:10.1145/48529.48536

[17] P.J. Denning. 1969. Equipment Configuration in Balanced Computer Systems. *IEEE Trans. Comput.* C-18, 11 (Nov. 1969), 1008–1012. doi:10.1109/T-C.1969.222571

[18] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 727–741. doi:10.1145/3582016.3582031

[19] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 113–124. doi:10.1109/PACT.2015.22

[20] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. 2019. *The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption*. Springer International Publishing, Cham, 133–194. doi:10.1007/978-3-319-90385-9_5

[21] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022), 52565–52608. doi:10.1109/ACCESS.2022.3174101

[22] Hyungkyu Ham, Jeongmin Hong, Geonwoo Park, Yunseon Shin, Okkyun Woo, Wonhyuk Yang, Jinhoon Bae, Eunhyeok Park, Hyojin Sung, Euicheol Lim, and Gwangsun Kim. 2024. Low-Overhead General-Purpose Near-Data Processing in CXL Memory Expanders. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 594–611. doi:10.1109/MICRO61859.2024.00051

[23] Wenqin Huangfu, Krishna T. Malladi, Andrew Chang, and Yuan Xie. 2022. BEACON: Scalable Near-Data-Processing Accelerators for Genome Analysis near Memory Pool with the CXL Support. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 727–743. doi:10.1109/MICRO56248.2022.00057

[24] Intel. 2024. CVE-2024-21823: Intel DSA and IAA Escalation of Privilege. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01084.html

[25] International Business Machines Corporation. 1964. *IBM System/360 Principles of Operation*. IBM Press. https://dl.acm.org/doi/book/10.5555/1102026

[26] International Business Machines Corporation. 1969. IBM System/360 Component Descriptions 2314 Direct Access Storage Facility and 2844 Auxiliary Storage Control.

[27] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

[28] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A High-Performance Database System Leveraging in-Storage Computing. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 924–935. doi:10.14778/2994509.2994512

[29] Aditya K Kamath and Simon Peter. 2024. (MC)2: Lazy MemCopy at the Memory Controller. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1112–1128. doi:10.1109/ISCA59077.2024.00084

[30] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. 2021. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. doi:10.48550/arXiv.2106.07102 arXiv:2106.07102 [cs]

[31] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2017. Ingens: Huge Page Support for the OS and Hypervisor. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 83–93. doi:10.1145/3139645.3139659

[32] Norman Layer and Edwin D. Reilly. 2003. IBM System 360/370/390 Series. In *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., GBR, 828–832.

[33] 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 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[34] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1203–1217. doi:10.1145/3676641.3715987

[35] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 417–433. doi:10.1145/3373376.3378497

[36] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters* 17, 01 (2007), 5–20. doi:10.1142/S0129626407002843 arXiv:https://doi.org/10.1142/S0129626407002843

[37] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[38] Marvell. 2024. *Marvell Structera A 2504 Memory-Expansion Controller*. Technical Report Marvell_Structera_A MV-SLA25041 _PB. 3

pages. https://www.marvell.com/content/dam/marvell/en/public-collateral/assets/marvell-structera-a-2504-near-memory-accelerator-product-brief.pdf

[39] MongoDB. [n. d.]. In-Memory Databases Explained - MongoDB. https://www.mongodb.com/resources/basics/databases/in-memory-database

[40] Nayana Prasad Nagendra, Grant Ayers, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2020. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. *IEEE Micro* 40, 3 (2020), 56–63. doi:10.1109/MM.2020.2986212

[41] A. Padegs. 1964. The Structure of SYSTEM/360, Part IV: Channel Design Considerations. *IBM Systems Journal* 3, 2 (1964), 165–179. doi:10.1147/sj.32.0165

[42] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab / Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/

[43] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 347–360. doi:10.1145/3297858.3304064

[44] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2830772.2830773

[45] Abishek Ramdas. 2023. *CCKit: FPGA Acceleration in Symmetric Coherent Heterogeneous Platforms*. Doctoral Thesis. ETH Zurich. doi:10.3929/ethz-b-000642567

[46] Redis Development Team. 2024. Redis Documentation: Diagnosing Latency Issues. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/latency/

[47] Anastasiia Ruzhanskaia, Pengcheng Xu, David Cock, and Timothy Roscoe. 2025. Rethinking Programmed I/O for Fast Devices, Cheap Cores, and Coherent Interconnects. doi:10.48550/arXiv.2409.08141 arXiv:2409.08141 [cs]

[48] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. 2022. Computational cxl-memory solution for accelerating memory-intensive applications. *IEEE Computer Architecture Letters* 22, 1 (2022), 5–8.

[49] Y. Solihin, Jaejin Lee, and J. Torrellas. 2002. Using a User-Level Memory Thread for Correlation Prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*. 171–182. doi:10.1109/ISCA.2002.1003576

[50] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 604–621. doi:10.1145/3676641.3711999

[51] 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 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. doi:10.1145/3613424.3614256

[52] Dufy Teguia, Jiaxuan Chen, Stella Bitchebe, Oana Balmau, and Alain Tchana. 2024. vPIM: Processing-in-Memory Virtualization. In *Proceedings of the 25th International Middleware Conference (Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 417–430. doi:10.1145/3652892.3700782

[53] Lukas Vogel, Daniel Ritter, Danica Porobic, Pinar Tözün, Tianzheng Wang, and Alberto Lerner. 2023. Data Pipes: Declarative Control over Data Movement. In *Conference on Innovative Data Systems Research*.

[54] Rui Wang, Christopher Conrad, and Sam Shah. 2013. Using Set Cover to Optimize a Large-Scale Low Latency Distributed Graph. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*. https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/wang

[55] Zhao Wang, Yiqi Chen, Cong Li, Yijin Guan, Dimin Niu, Tianchan Guan, Zhaoyang Du, Xingda Wei, and Guangyu Sun. 2025. CTXNL: A Software-Hardware Co-designed Solution for Efficient CXL-Based Transaction Processing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 192–209. doi:10.1145/3676641.3716244

[56] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. doi:10.14778/2732967.2732972

[57] Pengcheng Xu and Timothy Roscoe. 2025. The NIC Should Be Part of the OS *(HotOS '25)*.

[58] Chia-Lin Yang and Alvin R. Lebeck. 2000. Push vs. Pull: Data Movement for Linked Data Structures. In *Proceedings of the 14th International Conference on Supercomputing (ICS '00)*. Association for Computing Machinery, New York, NY, USA, 176–186. doi:10.1145/335231.335248

[59] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. Mckinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *OOPSLA'11 - Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA)*. 307–324. doi:10.1145/2048066.2048092

[60] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-Structured Analytics. *SIGPLAN Not.* 50, 8 (Jan. 2015), 183–193. doi:10.1145/2858788.2688507

[61] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow.* 13, 9 (May 2020), 1568–1581. doi:10.14778/3397230.3397249

[62] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

[63] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Columbus OH USA, 712–725. doi:10.1145/3352460.3358256