Databases in the Era of Memory-Centric Computing

Yannis Chronis
chronis@google.com
Google

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL

Lawrence Benson
lawrence.benson@tum.de
Technische Universität München

Helena Caminal
hcaminal@google.com
Google

Jana Gičeva
jana.giceva@in.tum.de
Technische Universität München

Dave Patterson
davidpatterson@google.com
Google

Eric Sedlar
eric.sedlar@oracle.com
Oracle Labs

Lisa Wu Wills
lisa@cs.duke.edu
Duke University

ABSTRACT

The increasing disparity between processor core counts and memory bandwidth, coupled with the rising cost and underutilization of memory, introduces a performance and cost Memory Wall and presents a significant challenge to the scalability of database systems. We argue that current processor-centric designs are unsustainable, and we advocate for a shift towards memory-centric computing, where disaggregated memory pools enable cost-effective scaling and robust performance. Database systems are uniquely positioned to leverage memory-centric systems because of their intrinsic data-centric nature. We demonstrate how memory-centric database operations can be realized with current hardware, paving the way for more efficient and scalable data management in the cloud.

1 INTRODUCTION

Databases have always been memory- and data-movement conscious, but the systems supporting them are or have been organized in a processor-centric way. In other words, most infrastructures running databases treat compute power as a first-class citizen and assume an even technology scaling between compute and memory metrics, such as FLOPs, bandwidth, and capacity. This processor-first design view is not exclusive to database infrastructure; in the cloud, resources are organized/sold around vCPUs, namely, each vCPU comes with a fixed amount of memory.

Current technological and economic trends suggest that scaling systems with a processor-centric design may become unsustainable in the future. Three key observations lead to this conclusion. First, the memory-bandwidth-per-core is not keeping up with demand. Core counts per CPU are increasing, cores continue to get performance improvements with every generation but the memory bandwidth is not increasing at the same rate (Figure 1, shows the memory-bandwidth-per-core across AMD EPYC CPU generations). Thus, this leads to underutilized chips for memory-bound jobs. Using more machines will be required to increase aggregate memory bandwidth. Second, memory is already the dominant part of the server cost and will continue to grow as the cost per byte plateaus (see Figures 2 & 3). Third, cloud vendors report that a significant portion of the memory in their datacenters is stranded/unused [18]. The result is that memory is the most expensive resource, and we are not using it efficiently.

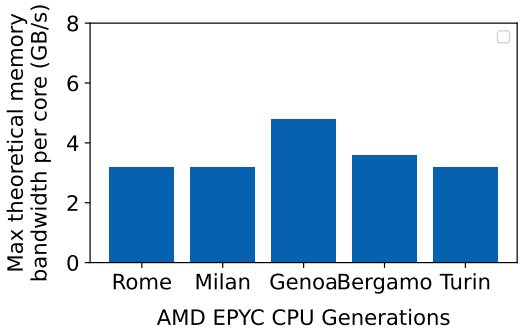

Figure 1: Theoretical maximum memory bandwidth per core for the current and past generations of AMD EPYC CPUs. Turin, the latest generation of AMD EPYC, has the lowest memory-bandwidth per core.[1]

We argue that these trends introduce a performance and cost Memory Wall. A promising direction to achieve a well-balanced system is to shift from a processor-centric to a memory-centric design. **Memory-centric computing aims to reduce the cost of memory in cloud systems by enabling efficient memory sharing. A memory-centric system treats memory itself as the dominant cost and compute power as commodity.** In a memory-centric system, memory is disaggregated, data is logically dissociated from processors, and different processing units (not just CPUs) can share memory and data via a pool (accessible over the network).

The goal of a memory-centric system is to scale memory capacity and bandwidth in a cost-efficient way for distributed query processing. A memory pool allows scaling of memory capacity while adding more compute nodes (with modest local DRAM) scales memory bandwidth. The cost savings are achieved by reducing the overall

[1]For each generation we use the CPU with the max number of cores.

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

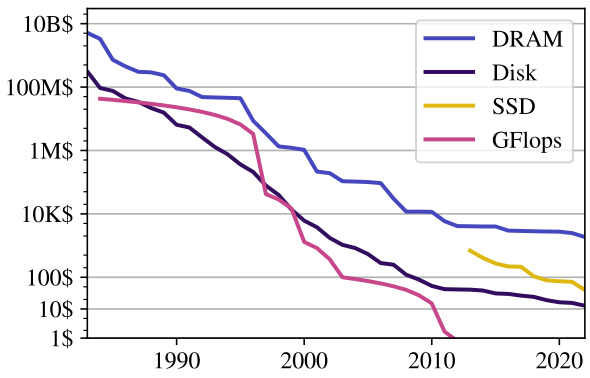

Figure 2: Cost of compute (cost/GFlops), memory, and storage (cost/TB) over time. DRAM price per byte has remained virtually constant since 2010 [1, 3].

memory required and memory waste. Interestingly, memory pools have additional benefits for distributed query processing. They can make performance more robust (for example, in the presence of data skew) and potentially make programming easier (depending on the memory pool implementation). For example, a CXL[2] shared memory makes the programming abstraction for a memory pool simple. Instances of memory-centric computing have been proposed in the past [12, 18, 21, 29].

Databases, compared to general-purpose applications, are uniquely positioned to take advantage of memory pooling for distributed query processing [16]. Databases already manage memory and data movement and use efficient out-of-core algorithms. Therefore, the design shift required to implement a memory-centric database is not disruptive. Section 3 gives examples of memory-centric database operations implemented with existing hardware technologies. Note that, even in a memory-centric design, being memory and cache-conscious inside each compute node remains important.

This paper presents the technological trends that introduce a new bandwidth and cost Memory Wall and makes the case that we need to shift from a processor-centric to a memory-centric system design. We further discuss how to design memory-centric database systems and the cost, robustness, and potential programmability benefits.

2 TECHNOLOGICAL TRENDS

The slowing of Moore's Law has forced chip manufacturers to turn to increasing the number of cores to improve performance. While chiplets have allowed AMD to increase the number of cores, increasing the memory-bandwidth-per-core is not as easy. Adding an equivalently increasing number of memory channels that run at a sufficient speed to supply cores with data is challenging[3]. Therefore, as the number of cores per chip increases, and each core's performance improves, the memory bandwidth ratio to core performance does not keep up. **Microprocessors will likely become memory-bound for many applications.**

Domain Specific Architectures (DSAs) are the only remaining opportunity for significant gains in processor performance. DSAs follow Amdahl's Law: performance improvement is limited by the fraction of the time the DSA is used. Given the high expense of developing new hardware, the domains that merit DSAs are limited. One example is Deep Neural Networks (DNNs), where GPUs and in-house alternatives developed by hyperscalers (TPUs, Azure Maia, and Cobalt) are used [10, 22]. While DSAs are easily justified for DNNs, it is hard to make a similar case for specialized compute for database management systems (DBMS). This is partly because databases are more focused on data access than computation and partly because the wide surface of databases and Amdahl's law limit the benefit of a DSA. **Ideally, we need to find a solution that reorganizes the standard components to provide a more promising foundation for DBMSs without having to justify the cost of developing and deploying specialized architectures.**

Today, the path to much faster general-purpose computing that improves all applications requires investment via scaling out by deploying more computers and building more data centers. **Improving performance when scaling out is not trivial; it requires sophisticated distributed systems that optimize resource management, data movement, coordination, and failure recovery.**

Figure 2 & 3 demonstrates that DRAM chip capacity and cost are not improving as quickly as they did in the past. **As a result, an increasing fraction of the cost of future computers will be DRAM.** Already a few years ago, Azure reports 50% of the server cost is memory [2], 40% for Meta [20], Google faces similar pressure[7].

New memory technologies like HBM have been proposed but it is unlikely that their byte/dollar can replace DRAM. **Therefore, new memory technologies are unlikely to benefit general-purpose architectures.**

At the same time, in the cloud, the fixed allocation of DRAM per CPU leads to under-utilization of the expensive memory resource. At Microsoft, an average of 25% of DRAM capacity is stranded [18]. **As DRAM becomes relatively more expensive, stranding DRAM due to its static binding to microprocessors becomes even less attractive.** In conclusion, all trends indicate that memory, not compute, is now the most precious resource, and hence, building memory-centric algorithms and systems is a logical next step.

2.1 The New Memory Wall

Until now, the term *Memory Wall* has been used to express the widening gap between processor speed and main memory access times [28]. This gap is particularly important for DBMS as analytical workloads are traditionally memory latency-bound. To bridge the gap, proposed algorithms carefully optimize the implicit "vertical" data movement from memory to the processor through the cache hierarchy. Databases use cache-conscious algorithms and data structures to mitigate the effects of this performance gap [4, 6, 23, 24].

[2]https://computeexpresslink.org/
[3]Turing, the latest generation of AMD EPYC CPUs can reach up to 192 cores/CPU and at the same time has the lowest memory-bandwidth-per-core out of all AMD EPYC CPU generations (Figure 1)

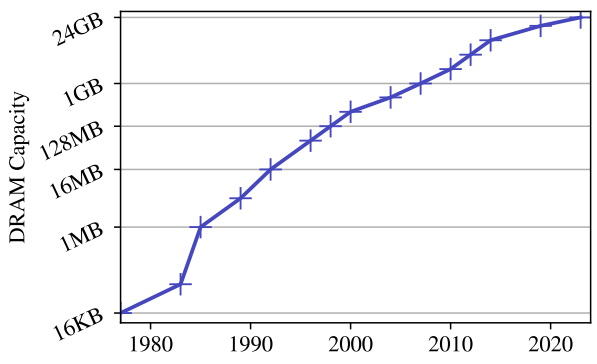

Figure 3: DRAM capacity per chip over time [8].

The combined effects of hardware and economic trends are pushing towards a new, multi-dimensional **memory wall of bandwidth, capacity, and cost**. The current conditions require increasing the number of servers in data centers to increase the aggregate memory bandwidth, which, coupled with the processor-centric design (Figure 4), memory capacity, and cost trends, will make scaling of database systems cost prohibitive.

3 A MEMORY-CENTRIC ARCHITECTURE FOR DATABASES

In this paper, we advocate to shift from a processor-centric to a memory-centric computer design (Figure 4). This naturally follows the technological trends and makes memory the centerpiece, thereby enabling memory disaggregation that can effectively minimize the cost of a cloud system.

3.1 Memory Pooling

As shown in Figure 4, in a memory-centric architecture, processors are dissociated from memory and data. This is in contrast to the traditional processor-centric computer design, where memory is a fixed resource attached to a processor core[4]. In the memory-centric design, compute nodes will have a modest amount of local DRAM. Most of the DRAM will be placed in a shared memory pool, thereby keeping the individual server and overall system cost in check. Data in the pool is not directly linked to any processor (of an execution node) and can be accessed by any compute node connected to the network fabric. Here, we want to highlight that the compute nodes that share the memory pool are not just high-end processors (left side of the memory pool) but also more power-efficient ARM/RISC-V cores (bottom side of the memory pool), or various types of accelerators (see right side of the pool). Such a memory pool-based architecture is particularly beneficial in cloud infrastructures. Not only does it help with reducing memory stranding, but it also helps with modern complex workload pipelines that move significant amounts of memory across the network between the various compute devices.

[4]For example, cloud vendors sell virtual CPUs with a fixed slice of associated memory; this is a direct result of existing hardware and software system design (https://aws.amazon.com/ec2/instance-types/).

One can reason about the memory pool as extending the memory/storage hierarchy with a layer that offers larger memory capacity than the local main memory, albeit at lower performance. Applications can use the memory pool in a few ways: (a) to extend the capacity of local DRAM memory (lowering the cost of local memory) and (b) to disaggregate memory for a distributed system (allowing the system overall to utilize memory more efficiently), and (c) to enable a more efficient data exchange.

3.1.1 ***Memory Pool Implementation.*** The move from a processor-centric model to a memory-centric model represents a *logical shift* that can be realized by reorganizing commodity computer components. In this section, we discuss the implementations of memory-centric design and the impact of emerging technologies.

Google's BigQuery [21] is an example of a database system that has a memory-centric system design for shuffles. BigQuery uses a separate shuffle service that is built on top of a disaggregated distributed memory pool to facilitate communication between compute nodes. Compute nodes that produce data to be shuffled write to the distributed memory service, and the consumer compute nodes read from the distributed memory. This design led to fast, large shuffles and reduced overall system memory requirements (and thus cost). Furthermore, it enables dynamic handling of intermediate results, re-optimization, and query check-pointing. The success of this design led to adopting it for other systems within Google[5].

Another instance of a memory-centric system design is RDMA-enabled memory disaggregation (remote memory). A notable industry example is Microsoft, which uses RDMA to temporarily "borrow" memory from remote nodes to avoid spilling data to disk in situations where not enough local memory is available [17]. However, there is no standard and easy-to-use way for cross-node memory sharing as envisioned by our paper.

Moving forward, new technologies like the Compute Express Link (CXL) promise an even more efficient and capable implementation of memory pooling. CXL is a new industry standard that connects up to 8 sockets and even various accelerators that can share a pooled DRAM memory [15]. With its larger logical aggregated memory capacity, CXL can significantly reduce the implementation effort when operating data stored in the memory pool and enable finer granular memory access. CXL is supported by current generations of computer architectures, including x86, ARM, and RISC-V. Refer to CXL papers [16, 18, 19, 25–27] for technology-specific trade-offs.

While CXL is close to general availability, other promising technologies like photonics [9] can enable even faster memory pooling and potentially facilitate deployment on a wider scale.

3.1.2 ***Cost of memory in the memory pool.*** Disaggregated and pooled memory reduces the cost of memory by reducing the overall provisioned memory. It can also improve the cost-effectiveness by recycling older generations of hardware that would have normally been retired. For example, memory modules of older technologies currently become obsolete when cloud providers upgrade their systems to the latest generation servers (e.g., move from DDR4 to DDR5). However, memories have a relatively large lifespan (i.e., 25 years) [8], and can thus technically continue to be used, although

[5]https://cloud.google.com/products/dataflow

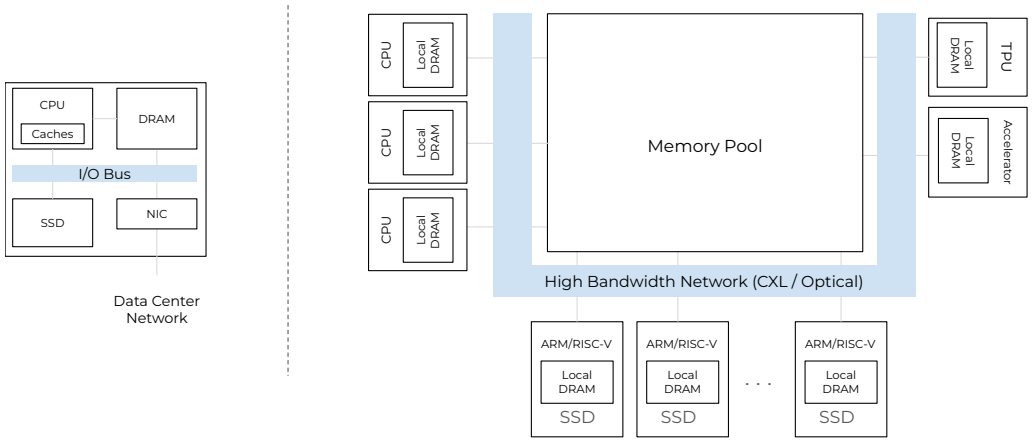

Figure 4: Left: Conventional Processor-Centric Computer. The CPU is at the heart of the design, surrounded by main memory (DRAM) and connected over I/O busses to storage and networking. The basic organization has dominated data centers and remained unchanged for decades. Right: A memory-centric architecture where memory is shared via a pool. Execution nodes contain a CPU, a TPU, or some other accelerator. Storage servers are provisioned with SSDs and low-cost processors. The memory pool can be made up from one or more physical servers, depending on the performance needed.

at the cost of a performance penalty. The drop in performance from a previous generation memory will not be noticeable in a memory pool when accessed over CXL. Thus, memory pools can help us better navigate the performance/cost trade-off by choosing the fraction and generation of memory to use in the pool and the compute nodes.

3.2 Databases and Memory Pooling

A fast memory pool is a natural fit for distributed query processing. It can be used as an extension of local memory and as a disaggregated memory for data movement between different execution stages [21] and data sharing.

Extending the memory/storage hierarchy with a memory pool layer will be challenging for many applications that do not explicitly manage data movement and placement [16]. Luckily, this is not the case with databases. Databases already manage the data movement between layers of the memory/storage hierarchy and use optimized out-of-core algorithms. This puts them in an advantageous position when it comes to realizing a memory-centric design.

Besides the cost-benefits that memory pooling brings, the extra layer in the memory/storage hierarchy can help mitigate the performance penalty of long-standing distributed query processing challenges and offer manifold system improvements. Two such challenges are: miss-estimation of intermediate results size [13, 14] and data skew. When either occurs, the working set data size can exceed the local memory capacity and degrade performance in an unpredictable way. Both are hard to predict, and when they occur, database systems either spill to secondary storage, over-provision memory to avoid spilling, abort execution, or use adaptive methods to redistribute the excess data. A memory pool benefits all approaches by extending the local memory to handle excess data, allowing easier implementation of adaptive algorithms, and enabling easier sharing of large intermediate results.

3.3 Cost of Data Movement

Traditionally, data movement is a significant cost in distributed query processing. A memory-centric design could directly or indirectly reduce data movement costs.

Section 3.1.1, discusses an efficient memory centric shuffling operation. Moving data from remote storage to an execution node or between execution nodes incurs the cost of moving data over the network but also the cost of data (de)compression), (de)serialization, (de)allocating buffers, copying, and (sometimes) format transformation [11]. Spilling to the memory pool instead of the storage layer reduces the overall cost, which is contributed to by the factors mentioned above. Emergent technologies such as CXL greatly simplify the implementation of this data movement reduction as they allow the data processing systems to spill at lower granularity instead of at storage format granularity. It is also helpful that most of the complexity of CXL is in the hardware, enabling the CXL pool to be seen as a (or multiple) remote DRAM socket(s).

3.3.1 ***Data Sharing.*** Databases are the first step in most data-intensive tasks (e.g., recommendation systems, retrieval augmented generation, ML inference, sensor data monitoring), where the database output is the input to one (or more) "consumer" systems (e.g., Tensorflow inference). End-to-end task execution is organized in a data pipeline, where each part of the pipeline uses the hardware (e.g., GPU and CPU, CPU and TPU, etc.) and software platforms suited to the operation it executes.

A memory-centric design can enable a shift where CPUs and accelerators, which are increasingly becoming part of such data pipelines, are "equidistant" from the data, enabling simpler communication and minimizing the performance cliffs observed when CPUs and accelerators operate with vastly different memory budgets.

4 EXAMPLE USE-CASE: DISTRIBUTED JOIN

In this section, we use an analytical model to compare the performance and cost of a memory-centric architecture to a processor-centric architecture for databases.

We use a distributed equi-join as our example use case. We join two relations, R (20 GB) and S (100 GB). R and S are stored on disk using a distributed storage service. The first step for the processor-centric and the memory-centric setups is to fetch the relations from the distributed storage service. This step is identical for both setups, so we omit it from our calculations. Next, the two tables are partitioned and shuffled to the execution nodes. We build hashtables on the R partitions and probe it for all tuples of S. Below, we list the performance assumptions we use in our analytical modeling:

Network: Each compute server is connected via 200 Gbit RDMA. With a 200 Gbit RDMA link, we can transfer data at 25 GB/s.

Local DRAM Bandwidth: The effective memory bandwidth for an execution node is 100GB/s.

Partitioning: For simplicity, we assume that partitioning is a memory-bound task that happens at 100GB/s in the compute servers. In a real deployment, each task would use a slice of memory bandwidth relative to the number of cores it uses. To keep this example simple, we assume we can use all of the available memory bandwidth to perform the join.

CXL memory pool: In our memory-centric setup, we add a memory pool implemented by one server connected via CXL to the compute nodes.

Join Build and Probe: We assume we can build and probe a hashtable at 8GB/s [5] when using local memory. We assume we can build and probe a hashtable at 5GB/s when CXL memory is used (the CXL performance was adjusted based on the relative performance between DDR and CXL [25]).

4.1 Join using the Processor-Centric setup

4.1.1 Join Algorithm and Latency. In this setup, we assume 10 computing servers. To perform the join, we first partition and shuffle table R (the build side). Each node partitions an equally sized chunk of R, $\frac{20\,GB}{10} = 2GB$ which takes $\text{part}_R = \frac{2\,GB}{100\,GB/s} = 0.02\,s$ to partition. For shuffling, we pessimistically assume that only 0.2 GB out of the 2 GB stays in the same compute node. Therefore, shuffling takes $\text{shuf}_R = \frac{1.8\,GB}{25\,GB/s} = 0.072\,s$. We assume that data distribution is skewed, and server 1 will end up with a $5GB$ partition of table R, server 2 with $2GB$ partition, and the rest with 1.625 GB sizes partitions. Partitioning and shuffling R takes $0.02sec + 0.072sec = 0.092sec$. To build a hashtable on table R, we spend $\text{build}_R = \frac{5\,GB}{8\,GB/s} = 0.625\,s$ (the largest partition defines the time this stage takes).

For S we follow similar steps, with the difference that partitioning, shuffling S and probing the R hashtable with S can be done in a streaming fashion. We assume that the slowest stage of this pipeline defines the speed; in this case, this is probing, which takes $\text{probe}_S = \frac{10\,GB}{8GB/s} = 1.25\,s$.

In total, this joins takes $0.092 + 0.625 + 1.25 = 1.967sec$.

4.1.2 Memory Requirements. In this setup, to perform the join without using secondary storage each node has $5.5GB$ of memory allocated. The $5.5GB$ are used to hold the largest build partition ($5GB$) and an additional $0.5GB$ buffer to partition and shuffle S in a streaming way.

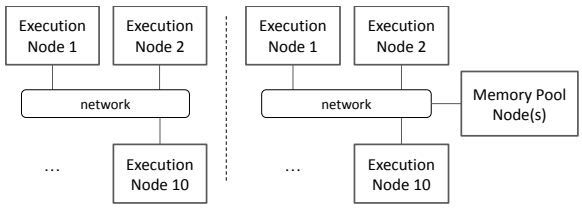

Figure 5: Processor-Centric Experiment Setup (left), Memory-Centric Experiment Setup (right)

4.2 Join using the Memory-Centric setup

In this setup, we assume 10 compute servers, each with 3 GB of local memory allocated to this join. We also assume a memory pool server connected to the execution nodes via CXL.

4.2.1 Join Algorithm and Latency. The time required to partition and shuffle R is the same as in the processor-centric design: $0.092sec$. The difference in processing the join in the memory-centric setup comes from the fact that the partition of R assigned to server 1 does not fit in its local memory ($5GB > 3GB$). Therefore server 1 will borrow $2.5GB$ of CXL memory from the pool and use it alongside $2.5GB$ of local memory to store the R hashtable. Therefore $\text{build}_R = \frac{2.5\,GB}{8GB/s} + \frac{2.5\,GB}{5} = 0.3125 + 0.5 = 0.8125\,s$. Similarly, the probe on server 1 takes $\text{probe}_S = \frac{5\,GB}{8GB/s} + \frac{5\,GB}{5GB/s}) = 1.625\,s$ (We assume that half of the probes will touch data stored in the local memory and half the data stored in the CXL memory). In total, the join takes $0.092 + 0.8125 + 1.625 = 2.5295sec$.

4.3 Memory Usage Reduction

This example shows that memory pooling can be used to reduce the total amount of memory used to perform the same join from $55GB$ ($10 * 5.5GB$) to $32.5GB$ ($10 * 3GB$ Local Memory + $2.5GB$ CXL Memory).

The processor-centric setup uses 1.7× more memory to achieve a 20% faster execution time compared to the memory-centric setup. The latest published literature describes memory as being 50% of the server cost [2]. Currently, a cloud-optimized Saphire Rapids CPU (without accelerators) costs around 4,000 USD, and loading a server with 4TB DDR5 (the maximum amount this CPU supports) costs around 40,000 USD. This represents a 10× price difference and a great opportunity for memory-centric computing. Depending on the configuration of a server (memory amount and CPU SKU), the cost difference could be anywhere from 2× to 10×[6].

We expect a much larger gain in real deployments, as memory in servers today is over-provisioned to handle spikes in usage. This means that during normal operation, memory usage is kept below 100% (for example, at 80% or even lower). This limit can be more aggressive with an efficient memory pool. In addition to cost benefits, the memory-centric setup is able to handle data skew in a natural way. The performance modeling of the memory-centric

[6]Retail prices were used for the comparison.

setup in our example is pessimistic, and detailed experiments are required to understand its performance potential.

4.4 Discussion

The example presented in this section is simple but effective at illustrating the cost benefits of a memory-centric approach. As the memory of local nodes is varied, we expect to see a trade-off between performance and memory cost, and here lies an opportunity for databases to control this trade-off to hit different cost-performance balances depending on the needs of each query.

Technologies like CXL can make the implementation and use of a memory pool much more efficient. From the programmer's point of view, local and remote memories appear similar, and cache coherence and error handling are handled by the hardware. Regarding data movement, it is likely that CXL will offer more opportunities to optimize the necessary processing costs (memory allocations, copying, format transformation, etc). Although we think CXL is a promising technology, we still need to understand the cost of additional networking it requires [16] and quantify the complexity it will bring to the deployment of resources and management.

Another area that needs careful study is the cost to performance tradeoff of the memory pool implementation. In Figure 4, the memory pool can be made up of one or more servers, depending on the performance and costs needs. More servers will increase the cost of the pool but will also increase the memory aggregate memory bandwidth the pool can support.

5 CONCLUSION

This paper is the result of discussions during Dagstuhl Seminar 24162: Hardware Support for Cloud Database Systems in the Post-Moore's Law Era [7]. We present and discuss the vision of memory-centric computing specifically for databases.

Memory-centric computing aims to introduce a new system design philosophy that treats memory as the most critical resource. It aims to address the effect of this new Memory Wall. There are multiple ways to make systems memory-centric. In this paper we discuss memory pooling as a means of memory disaggregation and memory sharing. New technologies like CXL and optical networks can enable efficient implementations of a memory-centric system. Nevertheless, we argue that the move from a processor-centric model to a memory-centric model represents a logical shift that can be realized by reorganizing existing commodity computer components. We plan to continue the exploration of the opportunities and implications of a memory-centric design for databases.

6 ACKNOWLEDGEMENTS

We are thankful to the organizers and attendees of the Dagstuhl Seminar 24162, which inspired this work. We also thank Prof. Nikos Hardavellas (Northwestern University) for providing valuable feedback.

REFERENCES

[1] 2017. Historical price of computer memory and storage. https://aiimpacts.org/wikipedia-history-of-gflops-costs/.

[2] 2020. *The Next Platform. CXL And Gen-Z Iron Out A Coherent Interconnect Strategy.* https://www.nextplatform.com/2020/04/03/cxl-and-gen-ziron-out-a-coherent-interconnect-strategy/.

[3] 2023. Historical price of computer memory and storage. https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 266–277.

[5] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 168–180. https://doi.org/10.1145/3448016.3452831

[6] Trishul Chilimbi, James Larus, and Mark Hill. 1998. *Improving pointer-based codes through cache-conscious data placement.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[7] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 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*. 727–741.

[8] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach.* Elsevier.

[9] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–14.

[10] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. https://doi.org/10.1145/3579371.3589350

[11] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual international symposium on computer architecture*. 158–169.

[12] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-centric Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2015, Portland, OR, USA, June 16, 2015*, Torsten Hoefler and Kamil Iskra (Eds.). ACM, 1:1. https://doi.org/10.1145/2768405.2768406

[13] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[14] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. https://doi.org/10.1007/S00778-017-0480-7

[15] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *arXiv preprint arXiv:2401.01150* (2024).

[16] Philip Levis, Kun Lin, and Amy Tai. 2023. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 18–24. https://doi.org/10.1145/3626111.3628195

[17] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 355–370. https://doi.org/10.1145/2882903.2882949

[18] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[19] Jie Liu, Xi Wang, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. 2024. Exploring and Evaluating Real-world CXL: Use Cases and System Adoption. *arXiv preprint arXiv:2405.14209* (2024).

[20] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and

[7] https://www.dagstuhl.de/de/seminars/seminar-calendar/seminar-details/24162

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*. 742–755.

[21] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: a decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[22] Microsoft. 2024. Azure Maia for the era of AI: From silicon to software to systems. https://azure.microsoft.com/en-us/blog/azure-maia-for-the-era-of-ai-from-silicon-to-software-to-systems/.

[23] Jun Rao and Kenneth A. Ross. 2000. Making B+- trees cache conscious in main memory. *SIGMOD Rec.* 29, 2 (may 2000), 475–486. https://doi.org/10.1145/335191.335449

[24] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 510–521.

[25] 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* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. https://doi.org/10.1145/3613424.3614256

[26] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, et al. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 818–833.

[27] Zixuan Wang, Suyash Mahar, Luyi Li, Jangseon Park, Jinpyo Kim, Theodore Michailidis, Yue Pan, Tajana Rosing, Dean Tullsen, Steven Swanson, Kyung Chang Ryoo, Sungjoo Park, and Jishen Zhao. 2024. The Hitchhiker's Guide to Programming and Optimizing CXL-Based Heterogeneous Systems. arXiv:2411.02814 [cs.PF] https://arxiv.org/abs/2411.02814

[28] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (mar 1995), 20–24. https://doi.org/10.1145/216585.216588

[29] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. 2021. Octopus+: An RDMA-Enabled Distributed Persistent Memory File System. *ACM Trans. Storage* 17, 3, Article 19 (Aug. 2021), 25 pages. https://doi.org/10.1145/3448418