Extending In-Memory OLTP with Persistent Memory

Hillel Avni, Aharon Avitzur, Nir Pachter, Vladi Vexler
firstname.lastname@huawei.com
Huawei Tel Aviv Research Center, Israel

ABSTRACT

In-memory storage engines are known to have higher throughput and lower latency than their disk-based counterparts. However, the more expensive and limited size DDR main memory, hinder their adoption. High capacity persistent memory (PMEM), which is finally commercially available in the form of Intel's Optane DC Persistent Memory Module (DCPMM), can be the opportunity to mitigate the above problems. This paper describes practical experience with PMEM as a memory extension for a commercial in-memory storage-engine, under the Huawei openGauss database with its MOT memory engine Interestingly, we find that when our storage engine is embedded in a fully functional RDBMS it can exploit PMEM for all data and indexes and maintain DDR main memory performance. However, in micro benchmarks we needed to throttle the use of PMEM to prevent it from becoming a bottleneck. In this paper we present our hands-on experience with PMEM, its good results on RDBMS and its performance on a microbenchmark compared to DDR.

1. INTRODUCTION

GaussDB [5] was announced in 2019 as a distributed relational database management system, followed by July 2020 open-source release of openGauss [6, 4] as a non-distributed (single-machine) community version of the closed-source GaussDB. Influenced by emergence of memory-optimized databases, such as Microsoft Hekaton [2], MemSQL [11], SAP HANA [9] and works in the art such as [13, 17, 18], we developed a high-performance memory-optimized storage engine, that is pluggable into the openGauss envelope.

openGauss in-memory storage engine, named Memory Optimized Tables (MOT), is described in [1]. MOT was created to harness the increasingly larger amount of main memory and processing cores for online transaction processing (OLTP). This new storage engine is integrated seamlessly in GaussDB.

Limitations of memory channels and DIMM slots per CPU leads to over provisioning of CPUs (sockets) just to reach the required memory capacity. For example, with Optane DCPMM [7, 12, 15] a 2-socket server can reach a total of 12TB memory (6 TB per socket), previously only possible with 4 or 8 socket servers. While DDR capacity reached physical boundaries, PMEM size is expected to grow in the near future. DDR cost is already up to 3 times higher per GB of memory on ultra-large (128-256GB) DDR DIMMs over commodity-size (16-64GB) DDR DIMMs with up to 60% savings of total server cost.

PMEM is not yet a simple replacement to DDR, and it has weaknesses and Idiosyncrasies [3] which must be considered to maintain performance. It is expected that other PMEM technologies (such as STT-RAM, memristor, ReRAM, NVDIMM), with highly variable characteristics, will be commercially shipped as PMEM in the near future, and DCPMM is also improved frequently with new revisions.

Contribution: We present a comparison between the performance of PMEM and DDR, and apply the results to the use of PMEM as volatile memory for data and indexes in MOT. Failure atomicity is out of the scope of this paper.

2. BACKGROUND

Optane DCPMM is Intel's PMEM, and the first incarnation of persistent byte-addressable memory in commercial hardware with large capacity. Previous solutions, including battery attached were always limited by capacity.

The CPU memory controller uses the DDR-T protocol to communicate with DCPMM. DDR-T operates at cacheline (usually 64B) granularity and has the same interface as DDR4 but uses a different communication protocol to support asynchronous command and data timing. Access to the media (3DXPoint) is at a coarser granularity of a 256B XPLine, which results in a read-modify-write operation for stores, causing write amplification.

In [16] they focus on DCPMM characteristics, mainly in the DBMS perspective, under several PMEM configurations with micro-benchmark tools. Other, pre-hardware papers [14], are also using PMEM as a monolithic block of memory, possibly configured as volatile with DDR cache. Unfortunately, this monolithic approach does not yield performance on the current version of PMEM hardware [8]. The exclusive PMEM is too slow, having too low bandwidth, while memory-mode sounds good in theory, but when tested on hardware, shows to be slow as it introduces arbitrary NUMA

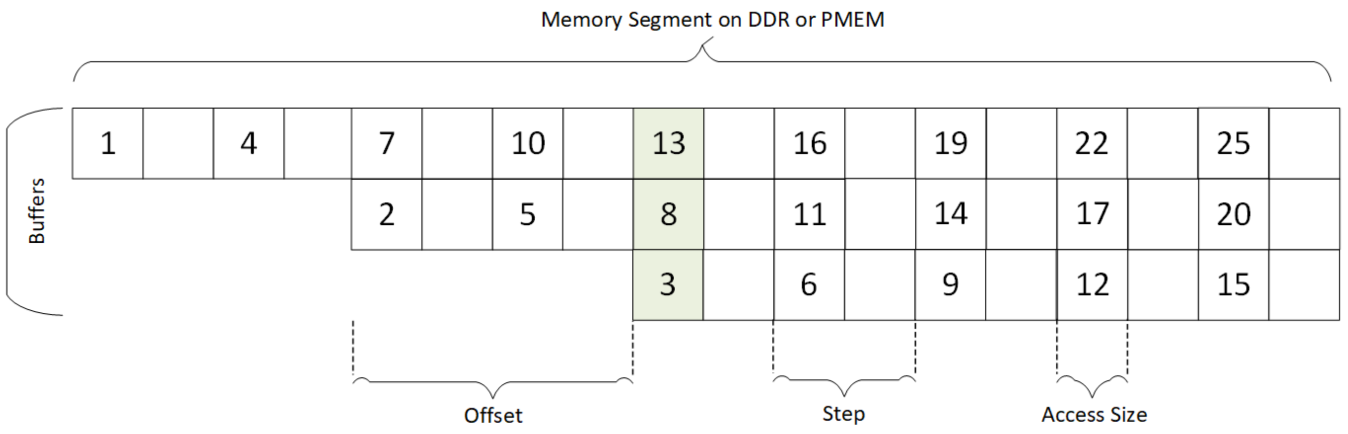


Figure 1: Testing the performance of PMEM, DDR and Cache interaction

accesses that are not in software control, and undermine performance.

When considering PMEM latency, it is important to note the effects of caching, which is the advantage of PMEM over block devices. Caching is accelerating access by orders of magnitude, but at the same time, it is unpredictable as evictions from cache to PMEM can cause arbitrary delays. In Section 3.1 we make an effort to understand the interaction of PMEM, DDR and cache.

Our goal: Maximal allocation of data and indexes on PMEM while maintaining acceptable performance.

3. PMEM AND DDR IN PRACTICE

First we experiment to see how DDR compares to PMEM on different access patterns, and how different mixes of PMEM and DDR compare to pure DDR. We do not invest in theoretical calculations of latency and bandwidth, and instead consider practical results.

3.1 PMEM and Cache

We conduct all experiments, both here and in Section 4, on a dual-socket server with Intel Xeon Platinum 8260L CPUs. Each socket has 22 physical cores, 22 hyperthreads, and is also populated with six 32GB Micron DDR4, and four interleaved 256GB Intel Optane DC Persistent Memory modules, combining to 1TB of PMEM. We run Centos 7.9 on the server.

To understand the actual latency of PMEM in OLTP and how it interacts with cache and DDR we devised the following test that is depicted in Figure 1:

- Two large memory segments of 10GB are allocated, one on local PMEM (S_p) and the other on DDR (S_d).
- Each segment hosts ten buffers, $B_0 ... B_9$ where PMEM B_k starts at $(S_p + k * Offset)$ and DDR B_k starts at $(S_d + k * Offset)$.
- In phase j of the test we access an $AccessSize$ in $B_0[j * Step] ... B_0[j * Step]$ for read or for write. After the access we place a delay of 0 or 20 cycles to simulate DBMS work.
- We try different mixes of PMEM and DDR by placing $B_0 ... B_m$ on PMEM and $B_{m+1} ... B_9$ on DDR.
- In the beginning we execute the test with all buffers in DDR and use it as the reference time T_{ddr}.
- Then we traverse different mixes of PMEM and DDR, i.e. k buffers from PMEM and $10 - k$ buffers from DDR, and measure the total time of traversal, T_{mix}.
- The graphs in Figure 2 presents T_{mix}/T_{ddr} for a specific workload

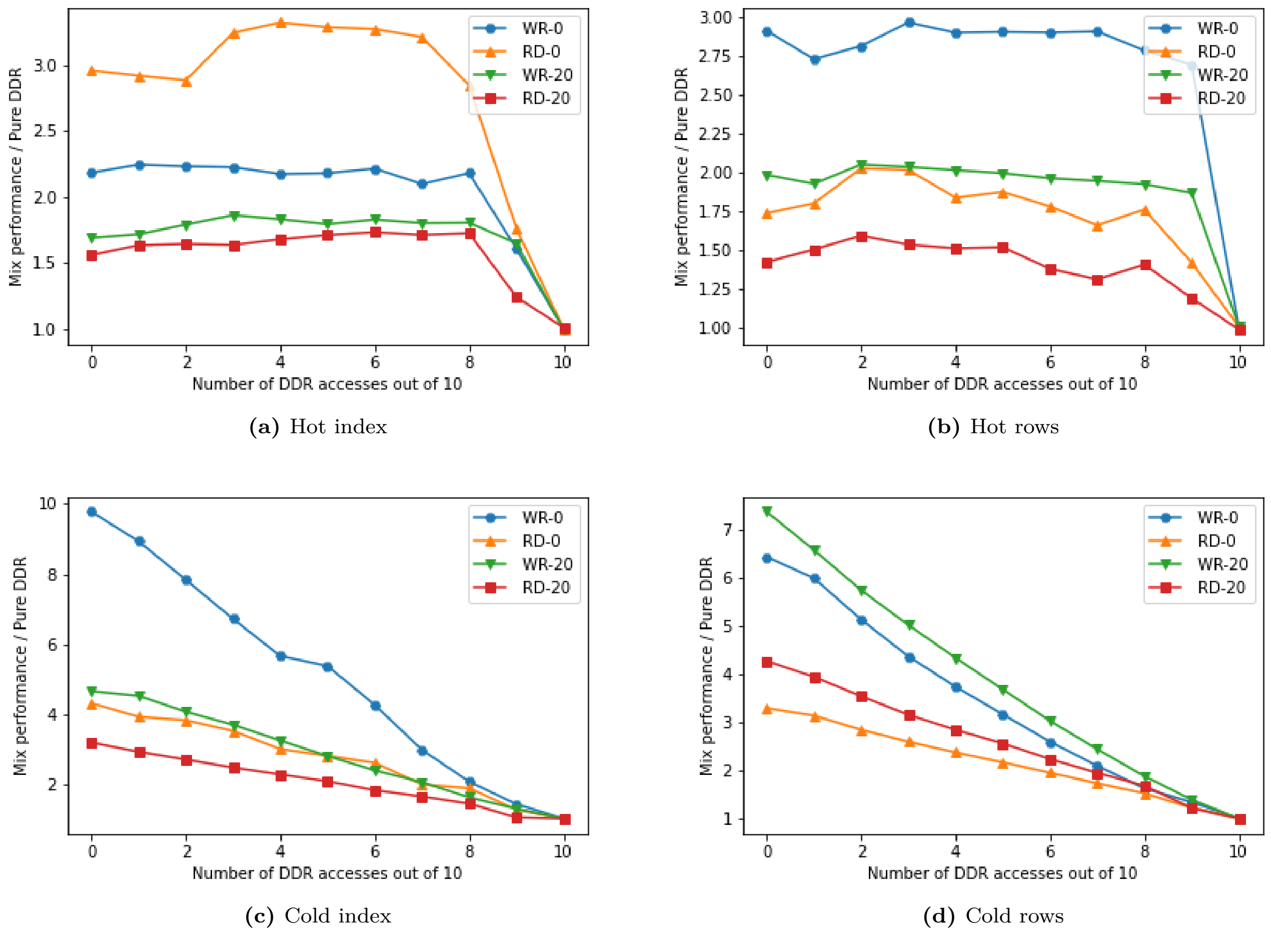
We use plain, no-flush writes, however we tried the copy also with non-temporal writes, as [3] suggests that not flushing the cache can in fact be detrimental to bandwidth when using PMEM with a built-in prefetcher. In our test, using the non-temporal writes made PMEM 3 times slower, and as we are not interested in persistence for this use-case, we removed it from the graph for readability.

The green squares (13, 8, 3) in Figure 1, represents accesses number A_{13}, A_8 and A_3 which access the same address. In fact, each address is accessed once per buffer that participates in the test. The number of cachelines accessed between two consecutive accesses to the same address can be calculated as follows:

$$\sigma = ((((Offset/Step) * Access\ Size) * Buffers) - 1) \quad (1)$$

The test is used to emulate non-sequential access (Offset between buffers and Step inside the buffers), with known number of accesses per address (Buffers) and known amount of memory accessed between consecutive accesses to the same address. The special property of the test compared to looping on the same address range is that L2 cache misses are distributed uniformly, and we control the amount of data scanned by the test and the rate of L2 cache hits vs. L2 cache misses.

The results of the test are presented in Figure 2. A line is RD (read) or WR (write) followed by the number of cycles we wait ("work")between copies. To eliminate noise we verify the executing thread and the PMEM reside on the same socket. This is a latency test and not a bandwidth test as we test the optimal performance per operation. If multiple threads will run concurrently, when bandwidth is saturated threads



(a) Hot index **(b)** Hot rows **(c)** Cold index **(d)** Cold rows

Figure 2: Index and rows access patterns for hot and cold usage types

will start waiting for each other. We see the bandwidth limit in later benchmarks.

We use 10 buffers and emulate four access patterns that are common in a database workload:

- Hot: When $\sigma < cache\ size$ where the L2 cache line size is $30.976MB$. in our test we set the offset to 64KB. In this access pattern, in steady state, i.e. after the cache is full, in each phase the last access to PMEM and the last access to DDR generate cache misses.
- Cold: In cold access patterns $\sigma > cache\ size$ so each access to PMEM or DDR generates a cache miss. For this test we use an offset of 32MB.
- Index: Access size is one cache line which is typical to index traversals. Specifically we use a Step of 1024 and access size of a single cache line.
- Row: Rows in TPC-C are few hundreds of bytes, so to emulate row access we use a step of 1024 and an access size of 512.

In Figure 2 we present the performance of workloads that mix the above access types, i.e. hot and cold accesses to indexes and rows. We assume these access types form any database workload so understanding their behavior will help us to analyze the performance of real workloads in Section 4.

Theoretically PMEM writes are 3x slower than PMEM reads while DDR reads and writes are symmetrical so we expect to see this in the results. However the actual results on real system are more involved and the test gives us the following insights:

- Hot accesses are not benefiting from mixing with DDR. This is expected as we pay a cache miss only for the first access to an address and the rest of the accesses are within the cache so they perform like DDR.
- In hot index, compared to DDR, writes perform better than reads, while in hot rows reads are closer to DDR performance than writes even when the writes are followed by a delay.
- Cold accesses are accelerated relative to the amount of DDR in the workload. in addition we see two phenomena:
  - Cold writes to indexes are slower than cold writes to rows.

– While in cold indexes delays make PMEM performance closer to DDR, in rows, without delays PMEM is closer to DDR performance.

Summary: We see that cold accesses can be an order of magnitude slower on PMEM. However accessing hot objects and adding a few cycles of work outside the access brings PMEM much closer to DDR performance. A commercial DBMS is performing work in DDR between rows and index accesses that mitigate PMEM higher latency.

3.2 Mixing PMEM and DDR

The concurrency control of MOT is OCC which means every accessed row is kept in a private copy until commit, and every uncommitted update is done on this copy. In commit, the row is locked, its version is validated and then it is copied back to the shared location. All the private copies are allocated from DDR.

For DDR allocation we are using jemalloc. For PMEM we use `memkind_create_pmem` to create memory pools in PMEM and than use `memkind_malloc` and `memkind_free` to allocate and release PMEM. In commit, after validation, we use `pmem_memcpy` to copy the memory to its shared destination with no flush.

4. EVALUATION

In this section we first evaluate MOT performance in PMEM when it is embedded in GaussDB, a fully functional SQL DBMS. Then we use microbenchmarks to analyze MOT stand alone, both when it is all in PMEM and with a combination of PMEM and DDR allocations.

4.1 DBMS Benchmark

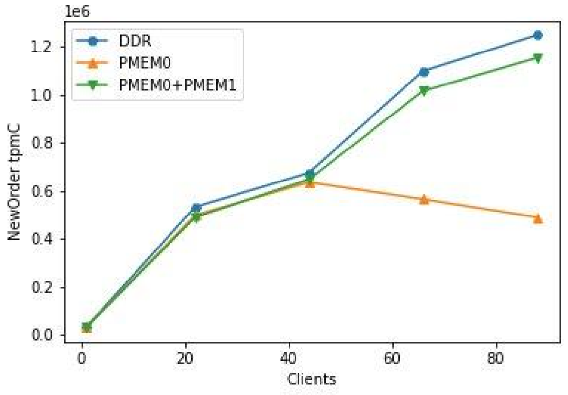


Figure 3: BenchmarkSQL Full TPC-C

To compare the performance of MOT inside GaussDB when all tables and indexes are deployed on PMEM to its performance on pure DDR we execute TPC-C by the standard BenchmarkSQL [10] tool. This benchmark includes five transactions which operate on nine different tables.

The results of our DBMS benchmark are presented in Figure 3. We add threads from the first socket and then from the second socket until we use all 88 hardware threads of the machine. When using only PMEM0(the 1TB which is located in the first socket), we get very close to DDR performance until thread 44 and than the second socket is not adding to the performance. However, when we add PMEM1, which is connected to the second socket, we manage to get almost DDR performance all the way to 88 clients. We make all even threads allocate from PMEM0 and all odd threads allocate from PMEM1, so, for example, the second thread which is on the first socket allocates from PMEM1. Figure 3 shows us that NUMA is not a factor here and when there is enough bandwidth the PMEM gets DDR performance in the full DBMS.

Summary: An industrial DBMS such as GaussDB, is incorporating an SQL engine and a logging and replication mechanism, and as can be expected from the results in Section 3.1, and as demonstrated in the DBMS results that are presented in 3, this allows PMEM to reach DDR speed. In fact, we see that the limitation of the scalability of a full DBMS workload on PMEM is not the latency, as when there is enough bandwidth, the DBMS is as fast on PMEM as it is on DDR.

4.2 Microbenchmark

All experiments in this section use the 22 cores, with 2 hyperthreads each, of one socket. As a full TPC-C microbenchmark is not scaling even to one socket, we focused on that case.

The microbenchmark we use to create Figure 4 is using MOT to perform all the accesses of standard TPC-C, and even pass the standard TPC-C consistency tests. However all transactions are prepared before execution and call MOT storage engine API directly to eliminate all overheads of the SQL engine and measure hardware potential. Each line name in a graphs in Figure 4 stands for the data that is maintains in PMEM, i.e. index, rows, both (all) or none. If the suffix is 1 only the 1TB PMEM0 is used, and if it is 2, all even threads use PMEM0 and all odd threads use PMEM1.

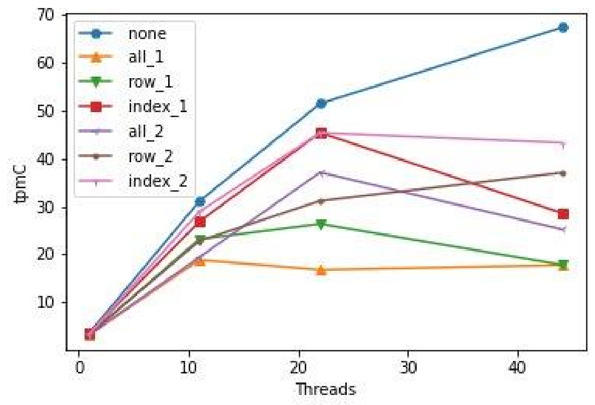
These lines are only one option of splitting the allocations and it is also possible to place only specific tables in PMEM or only index leafs or any other policy, as the mixed allocation is transparent to the transaction execution. All workloads are combined from the 5 TPC-C transactions:

- Payment which is mostly updating rows with one insert.
- NewOrder which performs a lot of inserts.
- StockLevel and OrderStatus which are read only.
- Delivery which is writing rows and deletes from indexes.

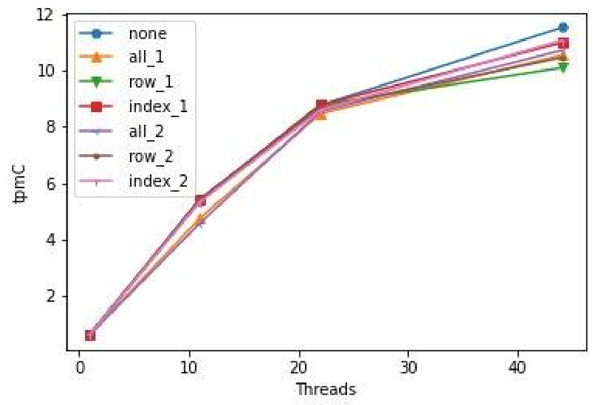
Each graph measures a different workload:

- Figure 4a: The standard TPC-C mix 0f 45% Payments, 43% NewOrders, 4% StockLevel, 4% OrderStatus and 4% Delivery.
- Figure 4b: More read oriented than the standard, with 25% Payments, 23% NewOrders, 24% StockLevel, 24% OrderStatus and 2% Delivery.
- Figure 4c: Only NewOrder transactions.
- Figure 4d: Only Payment transactions.

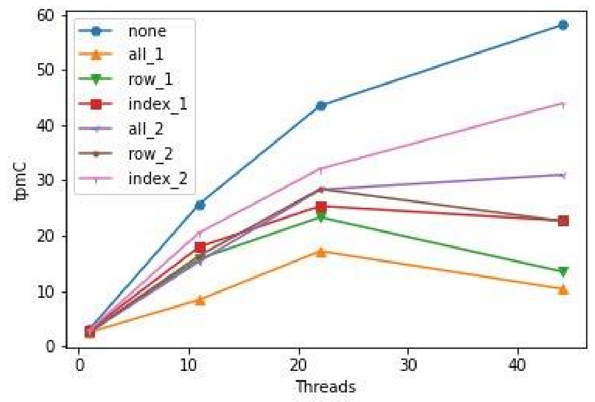
Here are some of the insights we can draw from the results in Figure 4:



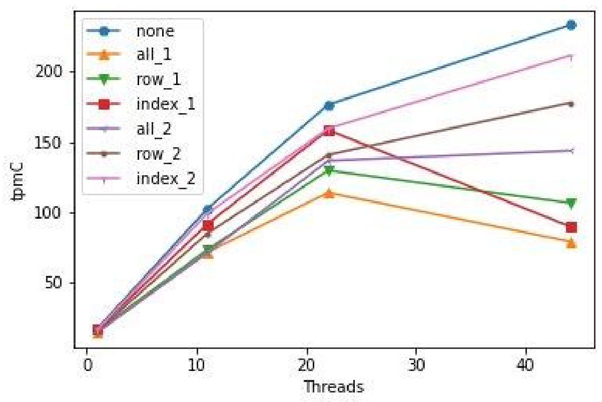
(a) Standard TPC-C mixture



(b) Read oriented TPC-C



(c) Only NewOrder



(d) Only Payment

Figure 4: TPC-C Microbenchmark Performance

- In all workloads using the two PMEM modules gives performance advantage already with 11 threads. Even though PMEM1 is on the remote socket, the added bandwidth is boosting execution speed, and the NUMA effect is not stopping scalability.
- Putting only the indexes on PMEM is faster than putting only the rows on PMEM. The reason is both that index accesses are shorter and that the internal nodes are hot and mitigate the cache misses from the leafs. Indexes are only 30% of the DB size, so if only the indexes are on PMEM we do not get a good utilization of PMEM.
- Figure 4b, is mostly read, and as there are fewer NewOrder transactions, i.e. less inserts, the working set is smaller. As a result even when all data and indexes reside on PMEM execution speed is comparable to DDR.
- Payment performance in Figure 4d shows that putting only the index on PMEM and putting only the rows on PMEM have the same performance with one or two PMEM modules up to 22 threads and than the second PMEM starts to give benefit. However, when both rows and indexes are on PMEM (all), the second module benefit is visible already with 11 threads, as bandwidth limitation is reached earlier. In the NewOrder benchmark at Figure 4c the second PMEM is improving performance already with 11 threads. The reason is that NewOrder is making a lot of inserts and manipulates much more memory then Payment, and as a result the bandwidth barrier is reached faster. Again we see that the bandwidth of the PMEM determines the performance.

Summary: When working in hardware speed, PMEM is not fast enough to replace DDR in all workloads. However, if there is enough locality or PMEM is blended with enough DDR, we can get close to DDR performance while allocating a significant amount of data on PMEM.

5. CONCLUSION

In this paper we evaluated the usage of the newly available persistent memory as a volatile extension to main memory for MOT, an in-memory storage engine of an OLTP database. When embedded in the fully functional GaussDB DBMS PMEM preserves DDR performance and will allow using PMEM, a much larger and cheaper memory, without a performance loss. However, in microbenchmarks we see that this generation of PMEM is not a transparent replacement for DDR as its lower bandwidth and higher latency can slow execution and limit scalability. However, when used in the right dosage it can extend memory size with only a small performance hit also there. In the future, as faster PMEM products will appear, we expect PMEM to be even more useful for in-memory databases.

6. REFERENCES

[1] H. Avni, A. Aliev, O. Amor, A. Avitzur, I. Bronshtein, E. Ginot, S. Goikhman, E. Levy, I. Levy, F. Lu, L. Mishali, Y. Mo, N. Pachter, D. Sivov, V. Veeraraghavan, V. Vexler, L. Wang, and P. Wang. Industrial strength OLTP using main memory and many cores. *Proc. VLDB Endow.*, 13(12):3099–3111, 2020.

[2] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254, 2013.

[3] S. Gugnani, A. Kashyap, and X. Lu. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.*, 14(4):626–639, 2020.

[4] Huawei. openGauss. `https://opengauss.org/en/`.

[5] Huawei. GaussDB Distributed Database. `https://e.huawei.com/en/solutions/cloud-computing/big-data/gaussdb-distributed-database`, 2019.

[6] Huawei. openGauss open source. `https://gitee.com/opengauss/openGauss-server`, 2020.

[7] Intel. Intel Optane DC Persistent Memory. `http://www.intel.com/optanedcpersistentmemory/`.

[8] A. A. R. Islam and D. Dai. Understand the overheads of storage data structures on persistent memory. In R. Gupta and X. Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, pages 435–436. ACM, 2020.

[9] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, and M. Grund. High-performance transaction processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[10] D. Lussier. Benchmarksql 5.0, 2019.

[11] N. Shamgunov. The MemSQL In-Memory Database System. In J. J. Levandoski and A. Pavlo, editors, *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014*, 2014.

[12] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden. Large-scale in-memory analytics on intel® optane™ DC persistent memory. In D. Porobic and T. Neumann, editors, *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 4:1–4:8. ACM, 2020.

[13] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[14] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1541–1555. ACM, 2018.

[15] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do. Lessons learned from the early performance evaluation of intel optane DC persistent memory in DBMS. *CoRR*, abs/2005.07658, 2020.

[16] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do. Lessons learned from the early performance evaluation of intel optane DC persistent memory in DBMS. In D. Porobic and T. Neumann, editors, *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 14:1–14:3. ACM, 2020.

[17] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[18] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1629–1642, 2016.