Improving Memory Utilization of Caracal

University of Toronto – ECE1724 Project Report

April 2021

Shirley Wang (1002154688) Corey Kirschbaum (997384165) Mohammed Baquir (1007489816)

ABSTRACT

The purpose of this project is to integrate persistent memory (PMem) into Caracal to increase the memory size available for the database without drastically reducing performance. Caracal is a multi-versioned deterministic in-memory DRAM database that serializes transactions and executes on rows and versions concurrently. The challenges with integrating a new type of memory system involved making efficient use of DRAM and PMem, modifying multi-versioning and garbage collection (GC) schemes to operate efficiently over the new memory layout design. Additionally, due to the lack of code documentation for Caracal there was a learning curve in understanding the code and original design.

The hypothesis at start of the project was performance would reduce due to higher latency of PMem relative to DRAM. However, by utilizing Caracal's deterministic epoch-based design the impact on performance might be less significant if GC could be performed dramatically faster.

Test results were obtained using the TPC-C benchmark to compare the performance of the original Caracal design to the new design. The final design results when using all-DRAM, instead of PMem, saw a 5% increase in throughput suggesting the design is successful. However, with PMem integrated there was a significant throughput drop of 77% from the original design. This was due to majority of epochs using two or less versions, such that they were stored in PMem for improved cache locality but at the cost of memory latency.

I. ORIGINAL CARACAL DESIGN

Caracal is a shared memory database using multi-versioning to achieve determinism and avoid the need for concurrency control related mechanisms. Multi-versioning enables parallelism so that transactions can read and write different versions of rows concurrently. To achieve this, Caracal batches transactions into epochs, and assigns transactions a serial order before execution. Each epoch is split into two phases: initialization and execution. These phases execute concurrently on all cores. Figure 1 illustrates a high-level view of the current Caracal design. In epoch *e*, all transactions are serially order and batched before execution. All transactions are then processed in the two-phase design concurrently with multiple cores. That is, the rows represent unique item is the database that will have the phases operate on them concurrently. However, the phases cannot start before all previous phase on all rows completes, as seen by the barriers. The initialize phase is further divided into two phases: insert and append. Insert phase creates new rows as specified by the transactions and inserts them in the database tables. Append phase inserts new versions for each row update (write) to the corresponding row version array, but the actual values will be written during the execution phase.

Minor GC runs during the append phase and frees unused versions before appending to a row. Minor GC has good cache locality since it only operates on the rows that are already being accessed during the append phase. However, rows that stop receiving updates will never have their unused versions reclaimed by the minor GC. Therefore, Caracal uses a major GC that is executed at the end of each epoch to free unused versions for rows that have not been updated for the last *k* epochs. As major GC operates on old and cold rows, it has poor locality, takes up CPU cycles, can pollute the cache, and affect performance in later epochs.

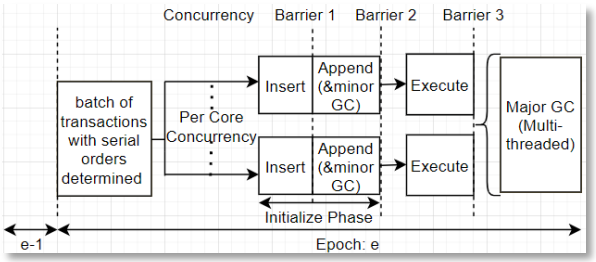

Fig. 1. Caracal Epoch Overview

Caracal stores a version array for each row in the database, as seen in Figure 2, and for each version it stores both the version ID (the serial ID of the transaction that created this version) ("sid" in Figure 3), and a pointer to the value of this version ("ptr" in Figure 3). During the initialization phase, Caracal identifies all the new rows that will be created by the transactions in this epoch and inserts them into the database with an initial version. Then, Caracal analyzes the write sets of the transactions, appending new empty versions to the version arrays for each row update. During the execution phase, transactions write their updates directly to the corresponding versions in the version array to avoid conflicts. Transactions that request a read can directly

access the version array and obtain the value from the largest element that has a smaller version number than the transaction's serial ID.

Figure 3 demonstrates the structure of a VHandle. Each VHandle is associated with and represents a row in the database where it contains a pointer to the version array as describe previously. Currently, 256 bytes are allocated for each VHandle. The first 64 bytes store metadata information on the row which includes a pointer to the row's version array. If the version array is smaller than 64 bytes, it can be inlined completely to the second 64 bytes of the VHandle, called the inlined version array. If the version values are small, they can be allocated from the inlined miniheap (last 128 bytes) of the VHandle. In this diagram we assume versions x, y, and z are larger than 128 bytes, so they are allocated from DRAM. This inlined design improves cache locality since if the version array is small it can fit in the inlined array, which is on the same cache line when reading a VHandle. If the version array grows and does not fit into the VHandle, it will be re-allocated from DRAM and the inlined version array will not be used in any future epochs.

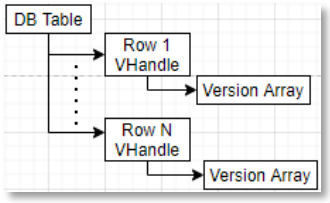

Fig. 2. Caracal Database Structure

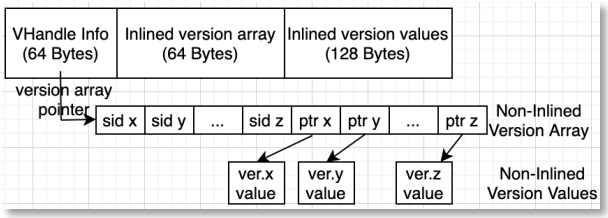

Fig. 3. Original VHandle Memory Layout

II. Caracal Design Concerns

Caracal has good performance and scales well. However, as Caracal is an in-memory database the size of the database is limited by the amount of memory (DRAM) of the system. Further concern on this limitation is made due to the multi-versioning requiring additional space. Storing data on disk would allow Caracal to support larger databases but would significantly impact performance. Adding more DRAM might be possible but requires much higher budget. Currently Caracal performs garbage collection to free unused, non-inlined version values. However, the size of the version array does not reduce, only the entries in the array become available for future versions of this row by overwriting older entries. As well, issues of poor locality and pollution of the cache exist due to major GC. The challenge is finding an alternate storage with comparable performance to DRAM at lower cost per gigabyte.

The approach section below describes the details of our design which addresses these concerns. The idea is to use PMem and for GC, only the last version of an epoch is necessary to be maintained after the end of the epoch, to act as the first version in the next epoch. Hence, all other versions can be discarded immediately at the end of the epoch. If a pool of memory is used such that we can simply reset an offset (similar to the brk system call) to "free" everything allocated from the pool, then it is fast to discard old version arrays and version values at the end of an epoch. This design is beneficial because only the final version of each epoch needs to be written to persistent memory. The reduced garbage collection helps with performance and memory availability.

III. Persistent Memory

Intel has recently introduced Optane Persistent Memory (PMem) that has high storage densities and is byte-addressable like DRAM. PMem has much lower cost per gigabyte than DRAM, but with a higher latency [1]. Currently Intel's Optane PMem line comes in 128-512 GB [5, 6] modules in DIMM form. As an example, for 128 GB PMem cost 29% cheaper at $4/GB (at $512) instead of $13.67GB (at $1,750) [7]. To get an idea of latency, DRAM latency can take 80-100 nanoseconds, PMem latency takes up to 350 nanoseconds, and 10-100 microseconds for NAND SSDs [1]. Caracal may benefit from using PMem to support larger databases while maintaining good performance.

IV. Background

The progression of databases (DB) first began with the introduction of database management systems (DBMS), becoming popular with Relational Database Model (IBM SQL). DB's were targeting execution on single machines with storage on disk. However, as more disk space was needed, databases were sharded across multiple machines. Sharding also allowed for more concurrent requests to be handled. To further improve performance (higher throughput, lower latency), caching and memory were better utilized. BigTable [16] and GFS [17] which stored data permanently on disk, utilized DRAM caching by locality and read frequency to reduce network communication cost and memory reads for better performance while supporting large files and databases. RAMCloud [18], Memcached [19], Redis [21], and others were completely in-memory to provide even better performance, only using non-volatile memory (NVM) like SSD's for asynchronous persistence. As an example, RAMCloud only keeps a single copy of data in DRAM and replicas on disk, distributed over multiple machines for high availability

and recovery. However, due to capacity limitation of DRAM systems like RAMCloud they still have expenses due to DRAM cost and as data is split across machines, network latencies will be higher as opposed to having more memory per machine which would reduce the amount of data split across machine's DRAM. Some systems like Memcached, Redis, and MemVerge [15] are now looking to integrate PMem to handle larger database size while maintaining their fast speeds and provide faster recovery.

A. Related Work in PMem

Assise [8] is a distributed file system designed to maximize the use of client-local NVM (PMem, SSD) as a cache between DRAM and slower, possible remote, storage. The design targets improving traditional systems that use remote storage servers, where clients cache state in DRAM and persistent state in NVM on remote servers (ie. RAMCloud). Assise avoids remote servers by caching state in local NVM, that is updates/writes are stored on NVM while DRAM caches reads. A three level memory hierarchy is used using various memory types based on hot-cold data; local DRAM and NVM, followed by remote NVM utilizing RDMA. The idea is by using fast persistent memory for caching it can help to reduce network communication latency and bandwidth. The paper states that to realize the performance benefit of NVM there is a need to not use fixed block sizes and instead track IO at original operation granularity. The original Caracal design allocates version data at fixed 32-byte granularity from DRAM. When updating the allocation method for the new PMem design, using a finer granularity level should then be considered. Assise was found to improve write latency by 22x, throughput by 56x, and scaled to 6x due to the reduce network communication for various local NVM levels of caching. However, the design of Assise is different than our goal since we are targeting Caracal on a single machine with no replication and in-memory. Assise utilizes various types of local NVM to improve performance in specific systems that store persistent data on remote servers. The new Caracal design is focusing on utilizing PMem to increase memory size, that is making use of DRAM and PMem, which is different than Assise.

Recipe [4] is an approach for converting concurrent DRAM indexes into crash-consistent indexes for PMem. Designing such fault tolerant PMem indexes from scratch is challenging; they must provide cache efficiency, concurrency, and crash consistency. DRAM indexes on the other hand already provide the first two. DRAM indexes are used to efficiently lookup data items in storage. The main insight behind Recipe is that isolation provided by a certain class of concurrent DRAM indexes can be translated with small changes to crash-consistency when the same index is used in PMem. PMem indexes can achieve crash consistency from DRAM indexes by following a recipe. Recipe-converted indexes outperform state-of-the-art handcrafted PMem indexes. They are optimized for cache efficiency and concurrency as they are built from mature DRAM indexes. Recipe-converted indexes encounter fewer cache misses as compared to hand-crafted PMem indexes. In our project with Caracal we will be using PMem but not for persistency, hence such indexes are not needed now. However, for future work to enable fault tolerance utilizing PMem's persistency, Recipe might be a potential exploration.

Persimmon [9] introduces a new building block called Persistent State Machine (PSM) for converting distributed in-memory replicated state machine storage systems into persistent, crash-consistent version with low overhead and minimal code changes. Persimmon aims to provide a design that general in-memory storage systems can use to integrate PMem, unlike Recipe and MOD [22] which targets specific data structure properties that are not suited to all in-memory systems. However, Persimmon does not support multi-threaded applications that apply operations concurrently. The system keeps 2 copies of state, one in DRAM and one in PMem. Reads and writes are performed using the state stored in DRAM, with updates/writes also adding an entry to a PMem log such that a background process executes them asynchronously on the PMem state. PMem is used to provide lower latency overhead on recovery instead of using SSD or disk. In all, Persimmon uses PMem for persistency and recovery which is different than the goal of using PMem to increase memory size (DRAM + PMem). Utilizing PMem's persistency is future work for Caracal.

MemVerge [7, 15] provides a virtualization (hypervisor) layer called Memory Machine that runs as a user-space application on Linux and by-passes the kernel to support generic applications to ease the integration of PMem by making DRAM and PMem appear the same with no development overhead. The design creates memory pools consisting of some ratio of DRAM and PMem in a single pool on or across different NUMA nodes (various configurations). Then allows applications to use the memory through a custom allocator to allow API calls like malloc and mmap, without modification. The design still requires clients to configure their design to make best use of DRAM and PMem through the Memory Machine interface. In terms of Caracal design, if it was fully understood how PMem virtualization was occurring by the Memory Machine it might be a possible integration choice for Caracal, or a comparison against could be performed in future work.

Figure 4 presents Sysbench test results done with MemVerge Memory Machine [7, 15]. It is important to note that the exact test setup and use of memory was not detailed making the results questionable. However, their

results show that with PMem they obtained only 21.5% less tps. However, the results with both DRAM and PMem shows better performance then purely DRAM solution, which again not understood how the result was obtained. Figure 5 presents results performed with MemVerge and kdb+ column series database from Kx Systems for financial trading [7, 15]. Both results demonstrate that it is possible to obtain as good results or better, though it doesn't provide enough understanding on how.

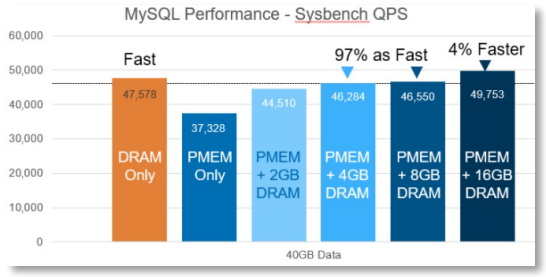

Fig. 4. MemVerge Sysbench DRAM vs PMEM Results [7, 15]

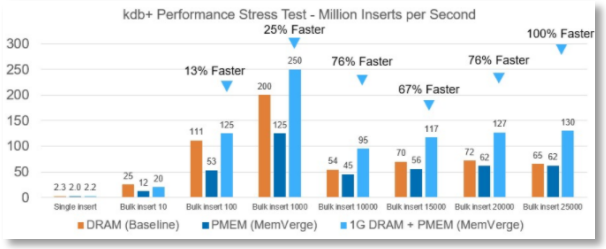

Fig. 5. MemVerge kdb+ DB Test [7, 15]

Redis [21] presents a design that utilizes persistent memory. In the traditional Redis with persistence the entire KVstore is placed in DRAM and replicated to NVM such as SSD. In order to integrate PMem they only store keys in DRAM, to save memory, and the values are stored in PMem. For persistence, the keys and a pointer to the values in PMem are replicated to NVM such as SSD. This is interesting and different from our purposed solution in the Approach section. Such a separation between keys and values might be something to try in the future to support persistency.

B. Related work in GC

Databases like Caracal generally rely on Multi-Version Concurrency Control (MVCC). While multi-version concurrency control (MVCC) supports fast and robust performance in in-memory, relational databases, it has the potential problem of a growing number of versions over time due to obsolete versions. Thus, to maintain the necessary number of versions in MVCC, versions which will no longer be used need to be deleted. This process is called garbage collection. In in-memory databases, garbage collection can often become the performance bottleneck. Traditional garbage collectors have several fundamental limitations. They have scalability issues due to global synchronization, vulnerability to long-living transactions caused by its inaccuracy in garbage identification. The general high watermark approach cannot clean in-between versions in long version chains.

Steam [2] is a lock-free garbage collection design that eagerly collects all versions that are not used by any active transactions on each traversal of a version chain. Steam's design is most beneficial for mixed workloads that have long running transactions such as OLAP, but comes at an overhead of maintaining lists of active transactions sorted by their timestamps. Caracal's focus is on OLTP workloads where transactions lengths are mostly similar. Caracal is also deterministic and epoch-based, which means all but the last version can be discarded at the end of each epoch. Caracal might benefit from Steam's design since it reduces memory footprint during the epoch, but we noticed that most version arrays in Caracal are very short, so the benefit might not be greater than the overhead of maintaining lists of active transactions.

Cicada [3] is a single node multi-core multi-version in-memory serialized DB. Cicada presents a design similar to Caracal; multi-version, concurrency, using inlining to embed small versions alongside record metadata. Cicada has four phases: "read", "write", "validate", maintenance", with GC performed at end of phases (major GC) similar to Caracal. However, Cicada performs validation to ensure serializability, unlike how Caracal serializes transactions at the beginning of an epoch. Cicada's GC uses a variant of epoch based reclamation (EBR) and quiescent state based reclamation (QSBR), detecting versions to free using fine-grained timestamps. Caracal's deterministic epoch only requires storing a record's last version at the end of an epoch. GC performing reclamation, no matter on how recent the rows' versions have been accessed, if performed between epochs will not have any contention. Drawback of this is poorer cache locality at the end of epoch.

There are other systems similar to Caracal that perform garbage collection. BOHM [12] guarantees serializable execution while ensuring that reads never block writes. BOHM does not require reads to perform any bookkeeping whatsoever, thereby avoiding the overhead of tracking reads via contended writes to shared memory. Hekaton [11] is a new database engine optimized for memory resident data and OLTP workloads. It is integrated with the SQL Server. ERMIA [14] adopts snapshot isolation concurrency control to coordinate heterogeneous transactions and provides serializability when desired. Its physical layer supports the concurrency control schemes in a scalable way.

These systems have various frequencies, tracking level and identifications for their garbage collection. Frequency and precision indicate how quickly and

thoroughly a GC identifies and cleans obsolete versions. If a GC is not triggered regularly or does not work precisely, it keeps versions longer than necessary. Some systems like HANA [10] and Hekaton [11] run GC on a background thread and thus the frequency at which GC is called is based on how often that thread is run. If the thread is run too infrequently, then GC is run based on outdated information. Systems like BOHM [12] organizes and executes its transactions in batches. GC is done at the end of a batch or epoch to ensure all its transactions have finished executing. Caracal is taking a similar epoch based approach, where GC runs at the end of each epoch. In Caracal, we keep a record of which rows to clean in GC, and thus at the end of each epoch we may clean those rows.

Database systems use different granularities to track versions for garbage collection. The most fine-grained approach is GC on a tuple-level. The GC identifies obsolete versions by scanning over individual tuples. Systems can collect versions based on transactions. Multiple obsolete versions of the same transaction can be identified and cleaned at once. Alternatively, Epoch-based systems can group multiple transactions into one epoch. Systems like Deuteronomy [13] and ERMIA [14] tracks versions on an epoch based level. Caracal tracks versions on a row-level. During regular transaction processing, we keep a track of individual rows. Hekaton cleans up all obsolete versions they see during query processing.

V. Approach

A. Overview of New Design

The goal of this project is to integrate persistent memory into Caracal while limiting its impact on performance. Figure 6 shows the new data structure design. VHandles are stored in persistent memory to achieve storage scalability. For version values, Caracal's deterministic and epoch-based design means that only the final version of an epoch will be needed in the future, and all the previous versions can be discarded at the end of an epoch. Based on this observation, per-core transient pools similar to the brk system call are used to store all intermediate version values of an epoch in DRAM, and discard them at the end of each epoch by simply resetting each core's transient pool offset without actually traversing the version array or deallocating memory. The final version values of each epoch will be stored in PMem, inlined to the VHandle miniheap if space allows. If new final versions are created in future epochs, the old final versions of previous epochs will be discarded through garbage collection. Similarly, version arrays are allocated from the transient pool, but the serial ID and pointer of the final version values of each epoch are stored in PMem.

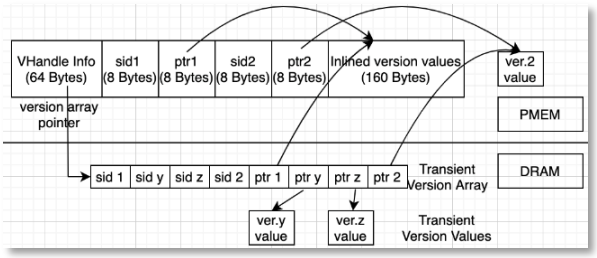

Fig. 6. Modified VHandle data structure in our design

Caracal's design has existing inlined space in VHandles. The serial ID and pointer of final versions are inlined to the VHandle as sid1, ptr1, sid2, and ptr2. The final version values are inlined to the miniheap in the VHandle if space allows, otherwise they will be allocated from PMem instead. The inlined space will store at most two versions at any time, which means the last 32 bytes of the inlined version array from the original Caracal design can be used by the miniheap instead. This will allow more version values to fit in the miniheap. The reason why there are at most two versions is because a row insert will populate the first version, the final append of an epoch will populate the second version, and garbage collection will run at the end of the epoch to remove the first version, leaving the final version in the VHandle as the first and only version. It is also possible to have another design that only stores one inlined version that is updated during garbage collection to the final version of the transient version array. However, this will not work for an optimization that we describe in the future work section that minimizes major GC, so our design stores two inlined versions to allow for future optimizations.

B. Details of New Design During Database Execution

Figure 7 shows the data structure changes during each phase of the database execution.

When initializing a row in the database (Figure 7a), during benchmark setup or during the insert phase of any epoch, the initial version is written to (sid1, ptr1) and the initial value is written to the inlined miniheap if space allows.

During the append phase (Figure 7b,c) of any epoch, the first thread to modify this row will observe that the version array pointer is null. This thread will allocate a new transient version array, set the version array pointer, and copy the existing version information (sid1, ptr1) to the transient version array. We don't copy the existing version value to the transient pool to avoid incurring overheads and storing duplicates. The thread also adds this row to the garbage collection list since it has been modified. There are no data races during these updates since Caracal protects modifications to the VHandles with mutexes during the append phase. After the above steps are completed, the first thread and any future

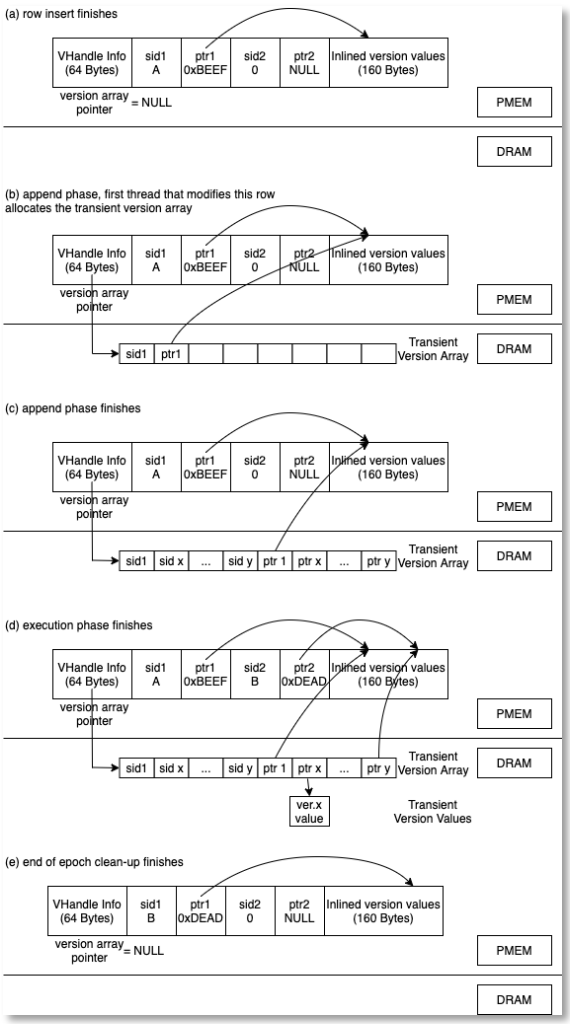

Fig. 7. Data structure changes during each phase of the database

threads can append new versions to the transient version array like how it is currently done in Caracal.

During execution phase (Figure 7d), the thread that writes the final version will write the version value to the inlined miniheap if space allows, otherwise it allocates PMem to store the version value. It will update the corresponding pointer in the transient version array, as well as (sid2, ptr2) in the VHandle. Other intermediate version values are allocated from the transient pool.

At the end of each epoch (Figure 7e), we run garbage collection to clean up every row that was appended to in this epoch. In garbage collection, we set their version array pointer to NULL, free the first (i.e. old) version pointed by ptr1, copy (sid2, ptr2) to (sid1, ptr1), and clear (sid2, ptr2) to (0, null). Then, we discard the transient version arrays and values by resetting the transient pool offset. After these steps, the database stores a single latest version for every row in PMem, inlined in the VHandle as (sid1, ptr1), and all stale values are cleared. One drawback of our design is the increased frequency of garbage collection compared to the original major GC in Caracal which only collects from rows that haven't been updated for K epochs. We address this concern in our future work section.

VI. Experiments

Experiments were performed on a machine with Intel Xeon Silver 4215 CPU with 8 cores. It has 96GB of DRAM and 256GB of persistent memory. We evaluate our design with the TPC-C benchmark and measure the throughput of the database using transactions per second.

Figure 8 compares the throughput of the original Caracal design, the new design using DRAM, and the new design using persistent memory. When using DRAM the design has a 5% higher throughput than the original Caracal design, demonstrating the benefit of reducing garbage collection by using the transient pool. However, when using persistent memory for our design, the throughput drops to only 22% of the throughput when using DRAM. This hit in performance can be explained by Figure 9, which shows that more than 81% of the versions are stored in persistent memory. In fact, more than 96% of the rows have less than 2 versions and don't benefit from the transient pool. Benchmarks with more frequent row updates might observe a greater benefit from using our design. Figure 10 shows that with a VHandle size of 256 bytes, 99% of the persistent versions are inlined. This gives good locality since 256 bytes is the internal access granularity of persistent memory [5]. Figure 11 shows that most version values are 30 bytes or less, which explains why 99% of the persistent versions are inlined.

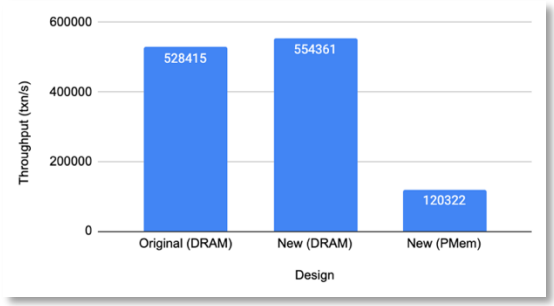

Fig. 8. Throughput of Original and New Design

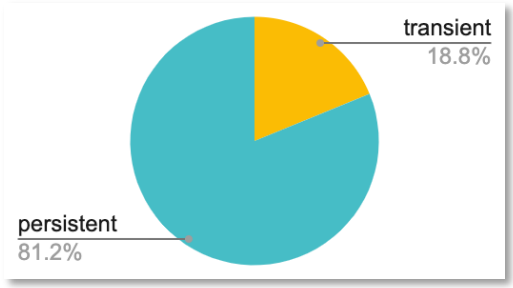

Fig. 9. Percentage of Transient v.s. Persistent versions

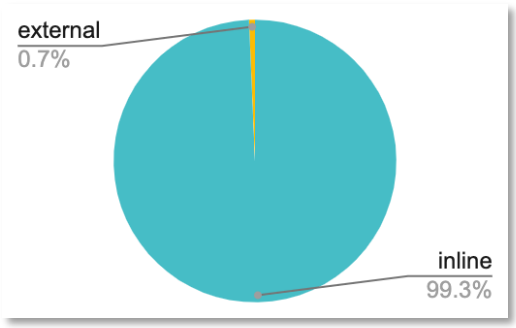

Fig. 10. Percentage of Inlined and External Persistent Versions

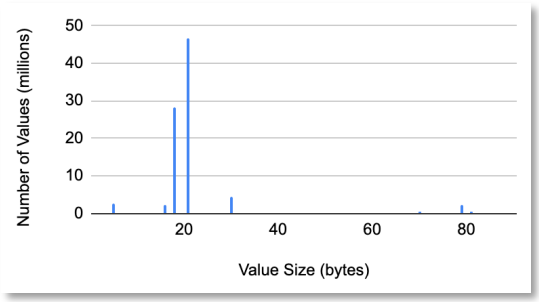

Fig. 11. Size of Version Values in Bytes

Figure 12 shows the total storage used for storing the database tables. The VHandles and the external persistent memory versions use more than 9 GB of PMem, while the transient pool only uses 73 MB of DRAM. The size of the transient pool depends on the length of each epoch and is not affected by the size of the database, thus our design makes Caracal scalable in terms of memory usage. One thing to note is that we store indexes in DRAM and they can grow when the database grows. We believe this is not an issue because indexes are small and storing them in DRAM can give us a performance benefit.

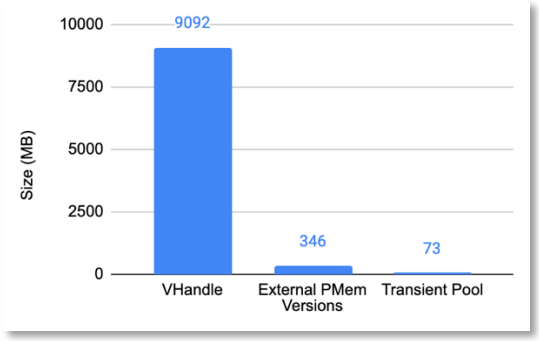

Fig. 12. Total Size of VHandles, External PMem Versions, and Transient Pool

Figure 13 shows the increase in execution time for the main components of Caracal when our design switches from all DRAM to integrating PMem. Most components are 2-6 times slower when using PMem, which is close to the difference of DRAM and PMem latency. On the other hand, garbage collection is 13 times slower when using persistent memory. It becomes a large overhead and takes up 11% of the total execution time of Caracal. Our GC is similar to the major GC in the original Caracal design, which can suffer from bad locality. By the end of the epoch, most rows in the GC list are likely not in the cache and GC must access them from persistent memory. We propose an optimization to reduce this issue in the next section.

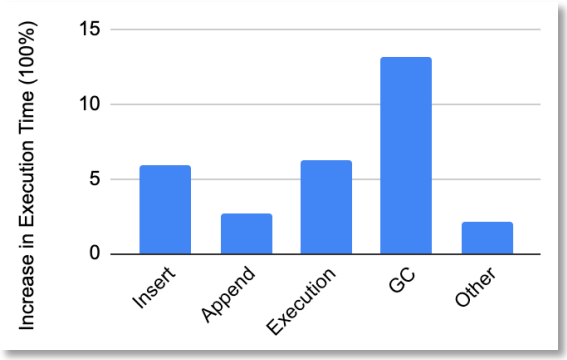

Fig. 13. Increase in Execution Time on PMem Against DRAM

VII. Future Work

A. Transition to Minor Garbage Collection

Our design performs garbage collection at the end of each epoch on all rows that were updated in the epoch. This major garbage collection is a large overhead, especially when running on persistent memory because it has a bad temporal locality, requiring reads and writes to persistent memory for every row that was updated. The major garbage collection is used by our design mainly to clear the version array pointer so that future transactions will not observe a stale version array pointer. One alternative solution that we propose for future work is to use minor garbage collection, which delays the garbage collection of each row until their next update, for better cache locality. We still need the major garbage collector to collect from rows that stop receiving updates, but its frequency can be reduced, and it only collects from rows that allocated non-inlined persistent memory for their version values. We expect this approach to perform better than our current design because of the reduced major garbage collection and the good cache locality for the minor garbage collection.

Modifications required for new GC approach:

- In each VHandle, store an epoch number associated with the version array to indicate if the version array is current or stale. This is required because it is possible that some rows will not go through any garbage collection if they are read but not updated in an epoch.
- During the append phase, the first thread to append to a row will perform garbage collection

for that row and update the epoch number to the current epoch after allocating a new version array.

- During the execution phase, if a transaction read finds a non-null version array, it will check the epoch number stored in the VHandle to decide whether it should read from the version array or from the last inlined version.
- To maintain the major garbage collection list, rows will be added to the list if a thread writes to their last inlined version and their first inlined version does not point to the inlined space. Rows will be removed from the list when the minor garbage collection is performed on them.
- Major garbage collection can be performed less frequently. For example, it can be performed when the memory is low or performed every epoch as in the Caracal design.

B. *Supporting Snapshots of the Database*

Persistent memory is non-volatile. For future work, we can use this property to store snapshots of the database after each epoch. When combined with logging the input transactions, we can recover the database after failures and replay the transactions. In order to support snapshots, we simply store a global "largest persisted sid" in persistent memory. This number indicates that modifications made by transactions with an equal or smaller serial ID are crash-safe. The largest persisted sid will be updated at the end of each epoch, before the major garbage collection, to have the value of the largest sid processed in this epoch. When recovering from a failure, for each row of the database we check if the second inlined version is non-null and smaller than the largest persisted sid. If yes, then we recover the second inlined version. If no, then we check and recover the first inlined version if it meets the criteria. If the first version also does not meet the criteria, then we discard the row because it was inserted during the epoch that crashed and the corresponding transaction will be replayed.

Other modifications are also required to support snapshots. First, the table indexes need to be stored in the persistent memory, which can hurt performance. Second, we need to ensure that modifications to the rows are flushed to the persistent memory before we update the largest persisted sid, to avoid inconsistencies in the persistent memory. One solution would be to flush after every row insert and after each update to the inlined versions. Finally, we need to ensure that the persistent memory will always be mapped to the same address across failures, so that we can correctly interpret the pointers that are recovered.

C. *Measure Persistent Memory Performance*

We plan to perform experiments to measure the actual PMem latency and the DRAM latency on the test machine. Experiments will include both sequential accesses and random accesses. We will also test with accesses of various sizes since the internal block size of persistent memory is 256 bytes. Combined with an analysis of the number of PMem and DRAM accesses in the different phases of Caracal, we want to see the correlations between the PMem latencies and the increase in execution time of the different phases in Figure 13.

CONCLUSION

In conclusion, we have integrated persistent memory into Caracal to support larger databases and our new design makes Caracal scalable in terms of storage size. By introducing the transient pools and persistent pools, we reduce the number of versions written to persistent memory and reduce garbage collection. Experiments using the TPC-C benchmark show that our design outperforms the original Caracal design when completely in DRAM. However, performance drops significantly when persistent memory is used, partly because many rows have less than 2 versions when running the TPC-C benchmark, hence most versions are stored in PMem. Garbage collection is a major overhead when using persistent memory due accesses to persistent memory with bad locality. We propose for future work to transition to minor garbage collection to reduce the overhead. We also propose for future work to utilize the non-volatile characteristic of persistent memory to support snapshots for failure recovery.

REFERENCES

[1] Intel, "Affordably Accommodate the Next Wave of Data Demands", [Online], April 2021. Available: https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-dc-selection-guide.html

[2] J. Böttcher, V. Leis, T. Neumann, and A. Kemper, "Scalable garbage collection for in-memory MVCC systems," Proc. VLDB Endowment, 2019, pp. 128–141

[3] H. Lim, M. Kaminsky, and D. Andersen, "Cicada: Dependably Fast Multi-Core In-Memory Transactions," SIGMOD '17 ACM International Conference on Management of Data, 2017, pp. 21–35

[4] S. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent DRAM indexes to persistent-memory indexes," SOSP'19 27th ACM Symposium on Operating Systems Principles, 2019, pp. 462–477

[5] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," 18th USENIX on File and Storage Technology (FAST'20), 2019, pp. 169-182

[6] Intel,"Intel Optane Persistent Memory," [Online], April 2021. Avaliable: https://www.intel.ca/content/www/ca/en/products/memory-storage/optane-dc-persistent-memory.html

[7] T. Morgan. 2020. THE ERA OF BIG MEMORY IS UPON US. Retrieved April 18, 2021 from https://www.nextplatform.com/2020/09/23/the-era-of-big-memory-is-upon-us

[8] T. Anderson et al., “Assise: Performance and Availability via Client-local NVM in a Distributed File System,” 14th USENIX on Operating Systems Design and Implementation, 2020, pp. 1011-1027

[9] W. Zhang, S. Shenker, and I. Zhang, “Persistent State Machines for Recoverable In-memory Storage Systems with NVRam” 14th USENIX on Operating Systems Design and Implementation, 2020, pp. 1029-1046

[10] J. Lee et al., “Hybrid garbage collection for Multi-Version Concurrency Control in SAP HANA,” SIGMOD’16 Internatioanl Conference on Management of Data, 2016, pp. 1307-1318.

[11] P. Larson, M. Zwilling, and K. Farlee, “The Hekaton memory-optimized OLTP engine,” IEEE Data Eng. Bull., 36(2), 2013.

[12] J. Faleiro and D. Abadi, “Rethinking serializable multiversion concurrency control,” PVLDB, 8(11), 2015.

[13] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang, “High performance transactions in deuteronomy,” In CIDR, 2015.

[14] K. Kim, T. Wang, R. Johnson, and I. Pandis, “ERMIA: Fast memory-optimized database system for heterogeneous workloads,” In SIGMOD. ACM, 2016

[15] MemVerge, “The Skinny on Memory Machine,” [Online], April 2021. Avaliable: https://memverge.com/the-skinny-on-memory-machine/

[16] F. Chang et al., “Bigtable: A Distributed Storage System for Structured Data”, 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006, pp. 205-218

[17] S. Ghemawat, H. Gobioff, and S. Leung, “The Google File System,” Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003, pp. 20-43

[18] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, “Fast crash recovery in RAMCloud,” SOSP’11 Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 29-41

[19] V. Marathe, M. Seltzer, S. Byan, and T.Harris, “Persistent memcached: bringing legacy code to byte-addressable persistent memory,” HotStorage'17: Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems, 2017

[20] Redis, “Introduction to Redis,” [Online], April 2021. Avaliable: https://redis.io/topics/introduction

[21] Redis, “Accelerate Redis in Kubernetes with PMEM-CSI and Intel Optane Persistent Memory,” 2020, [Online], April 2020. Avaliable:https://www.youtube.com/watch?v=qNcUWbZoNKI&t=167s

[22] S. Haria, M. Hill, and M. Swift, “MOD: Minimally Ordered Durable Datastructures for Persistent Memory,” In Proc. of ASPLOS (2020), 2020, pp.775-788