Caracal Database - Remote Access of Distributed Shared Memory

University of Toronto – ECE1747 Project Report

Alexander Boritz (1003132503) Ling Feng Zhou (1002739012) Corey Kirschbaum (997384165)

Abstract – *The paper presents the Caracal database and its current design and behavior. Through discussion, it is found Caracal fully utilizes shared memory access per cores on a single machine when operating in a distributed cluster but does not allow for cross node access of shared memory. The paper's goal is to extend Caracal to support such behaviour by allowing transactions to read from remote machines during execution, which we are calling remote-read-dependencies (RRD). TPC-C benchmark is used during evaluation to compare the behaviour of the original Caracal to the new implementation that supports remote access of shared memory. The results demonstrate that when running Caracal with remote read dependencies (RRD), by modifying the TPC-C benchmark to include such transactions, network latency and lack of work on the spinning wait receive side substantially reduces performance by increasing the run time and reducing the throughput of the system. Some future work is suggested to reduce the amount of idle cycle time by performing future work while receive side waits or utilizing a faster form of network transfer such as RDMA.*

I. Introduction

Caracal is an in-memory database utilizing shared memory and a deterministic transaction ordering. Caracal was initially created by Mike Qin and was further developed by Mike and members of Professor Goel's research team at the University of Toronto. Being deterministic, Caracal extends serializability. Serializability is a concurrency scheme that ensures the schedule of concurrent execution of transactions is equivalent to one that executes transactions serially in some order [2]. However, serializability only guarantees a serial order, allowing for transactions to be processed in any serial order [17]. Hence, deterministic extends serializability by guaranteeing a single specific predetermined serial order of transaction execution. Serial ordering is established before transactions are executed, ensuring that a transaction's outcome is uniquely determined by the database's initial state and an ordered set of known previous transactions. Deterministic databases are beneficial since predetermined ordering of transactions ensures serializable execution while avoiding concurrency-control related deadlocks and aborts. Most deterministic databases use partitioning to scale and avoid contention by partitioning data across cores and machines. Transactions execute in parallel across cores and machines, while executing serially in deterministic order on each partition. Caracal partitions by use of multi-versioned array of data and per-core queue of transaction pieces.

Caracal batches transactions in epochs and makes use of multi-versioned arrays of database table's data to avoid lock contention. Caracal can operate in a distributed manner, however only with transaction pieces operating independently on a machine, with machine communication not occurring during transaction execution. This means transactions cannot make use of the entire available memory in a cluster and can only access the shared memory on their own machine. The purpose of this project is to extend the Caracal database to support remote-read-dependencies (RRD) transactions. Implementing RRD will allow for transactions running on a machine to access shared memory on any other machine in the cluster during execution.

The hypothesis at the start of the project was implementing RRD functionality would result in a drop in performance due to the overhead of having network latency. As Caracal is an in-memory database, performing reads over a network would add additional delay into the system that would be higher than accessing RAM. Test results were obtained using the TPC-C benchmark to compare the performance of the original Caracal design without RRD transaction to the new design. The performance results for a single node cluster, where no RRD transaction existed due to using only a single node, saw the original and new design have comparable performance. However, once RRD transactions were introduced by increasing the cluster size and percent of RRD occurring, the runtime and throughput of the new design were substantially reduced due to nodes waiting to receive data.

Various challenges were faced due to the lack of code documentation of Caracal, resulting in a large learning curve in understanding the code and original design. As Caracal is a deterministic database, with synchronized phases across all cores and machines, deadlocks could and did occur during the development and integration step of the project, requiring debugging many aspects of the database in depth. Additionally, challenges involved modifying the current TPC-C test bench to add RRD transactions and getting a test setup working. During integration of the new design, we additionally had to ensure current features and behaviour of Caracal were not altered and function properly (ex. Scheduler, memory, single node and multi-node systems, regular transaction processing, etc.).

II. Background

A. Caracal Design

Caracal is a shared in-memory database using multi-versioning to achieve deterministic behavior and avoid concurrency control overheads. Multi-versioning enables parallelism so that transactions can read and write different versions of rows concurrently. Figure 1 below illustrates the epoch design of Caracal. Caracal's deterministic scheme batches transactions in epochs and executes them in a predetermined order by assigning a serial order before execution using a form of Lamport timestamp. Each epoch is split into two phases: initialization and execution. Phases execute concurrently on all cores, as well as across machines, with epochs synchronized with use of barriers. In the figure, a row represents a row in a table in the database. Each row entry will then have its own multi-versioned array and multiple rows will be processed within an epoch.

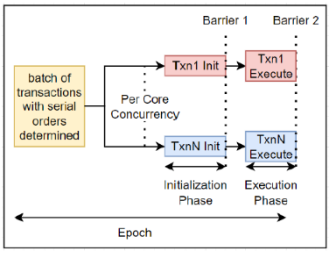


Fig. 1. Caracal Epoch Design

The initialization phase preprocess the transactions in an epoch to set up the versioning feature and identify where transactions should execute based data partition and core workload. By preprocessing transactions during initialization, Caracal allows splitting transactions into contended and uncontended pieces and running them in parallel, without concern about conflict.

B. *Caracal Schedular and Pieces*

Caracal executes transactions in threads per-core, with the schedular dispatching and executing transactions on a core. The scheduler maintains a piece queue per-core consisting of transaction pieces that worker threads execute on respective core. Transactions are split into multiple independent pieces and added to the per-core piece queue. Each time a worker thread chooses to run a piece, it will pick the one with the smallest timestamp which is the highest priority from its piece queue.

The scheduler also allows for pre-emption to avoid deadlocks and better utilize CPU cycles. Pre-emption is performed while an executing transaction is waiting for a data result to be ready. Figure 2 below demonstrates an example scenario where pre-emption is performed. A worker thread on core 2 is running a piece of transaction 2, since it has the smallest timestamp. While this transaction is running, the dispatch thread on core 1 could add a piece of transaction 1 (serialized before transaction 2) to core 2's piece queue. If transaction 2's read depends on transaction 1's write, then transaction 2 will wait forever even though transaction 1 is in the piece queue. Hence, while a worker thread waits for a read during execution of a piece, it spins and periodically checks the queue for a piece with a smaller timestamp. If such a piece exists, then Caracal suspends the current worker thread and creates a new one to execute the piece with the smaller timestamp. When finished, it resumes the suspended thread.

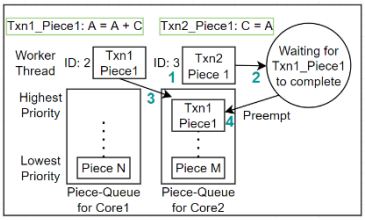


Fig. 2. Caracal Piece-Queue Schedular

C. *Remote-Read-Dependency (RDD)*

Figure 3 below provides a simple visualization of a remote-read-dependency. In this example, a simple transaction A=A+B needs to be executed. Based on how Caracal processes transaction, each transaction will get executed by the node that holds/stores the data for the write value. That is, during the initialization phase Caracal preprocesses the transactions and generates a write-set of all transactions in the epoch. That way during execution each transaction can get processed on the node that stores the data entry on which the transaction writes to. In this case, Node 1 holds the data for A, which is what the transaction writes to. Hence, the transaction will be executed on Node 1. However, this transaction also requires B which is held by a different node, Node 2. Therefore, a RRD exists and requires, at a point during execution, that Node 1 read the appropriate value for B from Node 2 based on the expected deterministic serial order.

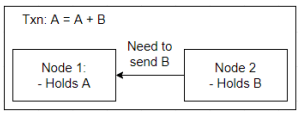


Fig. 3. RRD Example

Currently, Caracal will transfer transaction pieces to the appropriate machine that stores the partition of the database that the transaction piece will work on, as already described in the example above. However, transactions in Caracal do not support accessing data across machines as the TPC-C benchmark does not contain data dependencies across warehouses.

D. *Original TPC-C Benchmark*

The benchmark used for testing is called TPC (Transaction Processing Performance Council) version C, TPC-C. TPC-C is used to compare the performance of online transaction processing systems and simulates the activities of a wholesale supplier. In such a simulation, suppliers stock items in multiple warehouses, customers place new orders and pay for orders, with items eventually delivered to customers. The TPC-C models this process with various transaction types, however for the purposes of this project only the payment transaction is of concern, as it will be modified to introduce RRD's. The payments make up 43% of the total number of transactions in the system, with 100k transactions occurring per a warehouse.

Payment transactions are split into 3 pieces by Caracal that act on the Warehouse, District, and Customer tables which are partition across nodes. The relationship between these tables is in relation to warehouses. That is, each warehouse will have multiple districts and a district will have customers. Additionally, 15% is used at the number of payment transactions that will have their piece operate on a customer table partition located on another warehouse than the one used in the current payment transaction warehouse piece. That is, from 100k transactions, 43% are payment transactions.

100k x 0.43 = 43k payment transactions

And 15% will have their customer transaction piece operate on another warehouse.

43k x 0.15 = 6.45k customer transaction pieces on another warehouse

Figure 4 below illustrates the TPC-C payment transaction. The example presents a 2-node cluster, with the number of warehouses split across the 2 nodes. Hence, if 32 warehouses are used, then each node will have 16 warehouses operating on them with ideally a core-to-warehouse relation. Since transactions are in relation to warehouses, these 6.45k customer transaction pieces that are operating on a different warehouse or (# of warehouses – 1) and have their warehouses located on the other node is around:

(6.45k x 16/31) = ~3329 customers transaction pieces will execute on a different warehouse located on a different node.

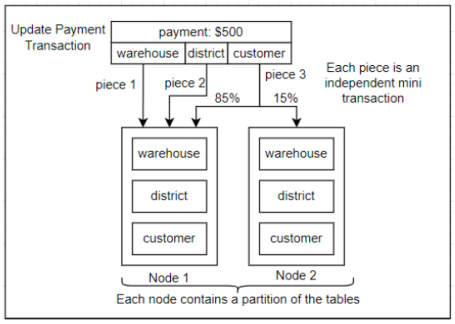


Fig. 4. Original TPC-C Payment Transaction

E. Original Caracal Behaviour

Currently Caracal can execute in a distributed fashion by running transaction pieces independently on multiple nodes, dependent on only the data stored on their own machine. Transactions are split into pieces based on their write sets and sent to the correct node at the beginning of the execution phase. That is, as mentioned previously, transactions pieces currently execute independently, only modifying local data, and only requiring the transaction inputs and data from the local node. This is possible due to the design of the TPC-C benchmark, as no data dependencies exist across warehouse partitions, allowing Caracal to split and distribute transaction pieces across nodes while using this partitioning scheme.

The feature currently not supported are transactions that have remote-read-dependencies (RRD) to another machine in the cluster. RRDs would allow for alternative partitioning schemes and load balancing options.

III. New Caracal Design

A. RRD Implmentation

The new Caracal design involved code changes to the TPC-C benchmark as well as Caracal's internals to support the new RRD feature. It is implemented generically to allow for other uses, though in the context of this project, it has been applied to a modified Payment transaction from the TPC-C benchmark. Figure 5 below provides a visual that can be compared to the original Payment transaction from Figure 4 to provide a high-level understanding of the changes. The main change to the TPC-C benchmark is that customers pay tax. The tax amount is retrieved by the warehouse piece and is used in the customer transaction piece to update the customer payment to include paying a tax amount. Refer to appendix A and B for further details on the payment transaction and implementation changes.

At a high-level sense, extra memory is allocated to store a FutureValue<int32_t> class in an epoch's memory area. This future area is used to hold the data required by transaction read dependencies, like the RRD for tax amount. That is, during execution of a payment transaction, when executing the warehouse piece of the transaction it will store the current tax value in the transaction's future value area on its local node, setting its local ready flag. The future area on a node is accessible by all threads on all cores operating on the same node due to the shared memory. Hence, any other transaction pieces executing on the same node that requires reading the warehouse tax value can do so by polling until the ready flag is set to true and then reading the value. As a percentage of customer transaction pieces will execute on the same node as their warehouse partition that the customer piece accesses for tax amount, the added code should not add any major overheads besides acceding shared memory.

For the case when a customer piece needs to access the warehouse tax amount from a warehouse not located on the same node, this will result in an RRD. In this scenario, a TCP packet is created with a header identifying it as FutureValue data, containing the value being sent, and details identifying the associated FutureValue object on the node. This is sent to the subscribed receiving nodes. This subscription is possible due to Caracals deterministic nature. Extra visualization to this process is given in Figures 6 and 7. Once the receiving side processes the incoming future value packet it sets its own future value to the decoded packet value and then sets its local ready flag to true. At this point the waiting customer transaction piece will see the ready flag is set to true, finish processing the piece, and then move onto the next transaction.

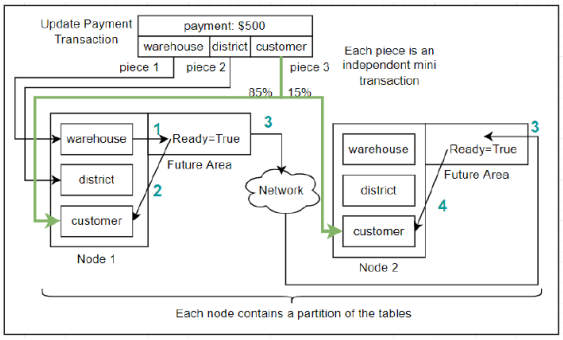


Fig. 5. New TPC-C Payment Transaction

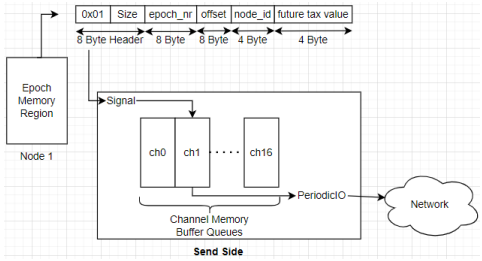


Fig. 6. Future Value Send Side

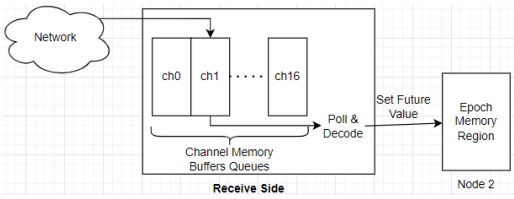


Fig. 7. Future Value Receive Side

This network communication implementation for RRD only requires one TCP network message to transfer the future value. In typical RPC systems two messages are required; one for the request of data, and one to send the data. However, due to the deterministic nature of Caracal and the fact that an epoch is split into two phases, the write sets are known during execution allowing for nodes to know where they must send future values to. This means that the number of network messages can be reduced from two to one message.

IV. Evaluation Results

A. Evaluation on node count

The test configurations used during evaluation were 1, 2, and 4 node clusters. A total of 32 warehouses were used for the TPC-C benchmark which were evenly divided among the nodes. 16 cores per node were used and execution was for 50 epochs. A total of 100k transactions occurred per node each epoch.

Table I and II below show the results, plotted in Figures 8 and 9. For the single node experiment, the performance remained mostly unchanged between the original design and the new design. Although the remote read dependency (RRD) code was introduced here, it caused minimal impact, as all data dependencies remained on the same node. However, as expected, for the two-node experiment, more time was spent in processing the epochs and the transaction throughput was decreased significantly. The overhead was introduced by RDD's. In the new design, the receiver required information from the sender to process some transactions in its queue. Therefore, the extra latency introduced by the network caused the receiver to spend time spinning, waiting for the future value. Additionally, the overhead became more severe when run on four nodes as each node had to manage and wait on future values for three senders compared to only one other node in the two-node experiments. The extra latency caused by waiting for the future values was about double in this case.

B. Evaluation on varying remote read dependency

By default, 15% of the payments have customers on a separate partition, requiring information from the host node. The following evaluation was performed with two nodes and adjusting the remote dependencies from 15% to 3.75%, 7.5%, 30%, 60% to observe how the total epoch time and throughput responded to the change in number of RRDs. The results are shown in Figure 10 and Figure 11. The total epoch time and execute time grew linearly with the remote dependencies due to more waiting being introduced while the transaction though decreased quadratically as a result. The insert and initialize time remained unchanged being unaffected by the remote dependency changes.

TABLE I. Original Design

Number of Nodes	**1**	**2**	**3**
50 Epoch time (ms)	7128	7476	7412
Throughput (txn/s)	687399	655409	661055

TABLE II. New Design

Number of Nodes	**1**	**2**	**3**
50 Epoch time (ms)	7223	25925	78655
Throughput (txn/s)	678327	189003	62296

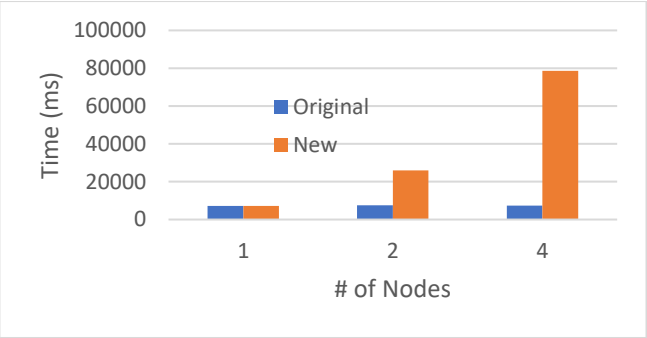


Fig. 8. 50 Epoch Time Evaluation Comparison

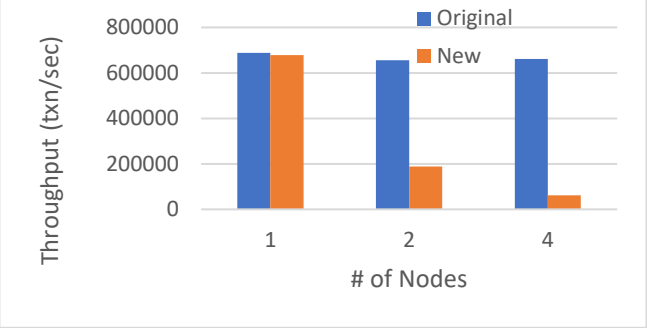


Fig. 9. Throughput Evaluation Comparison

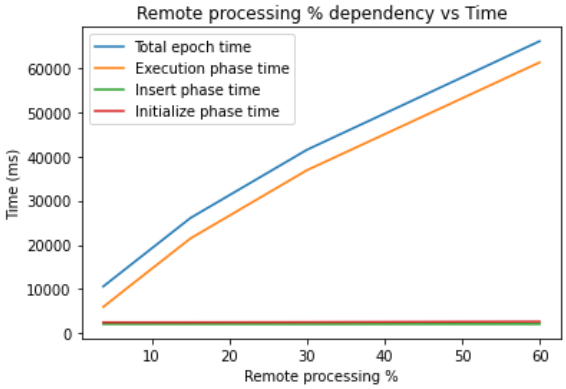


Fig. 10. Epoch Phase Time vs # of RRD Transactions

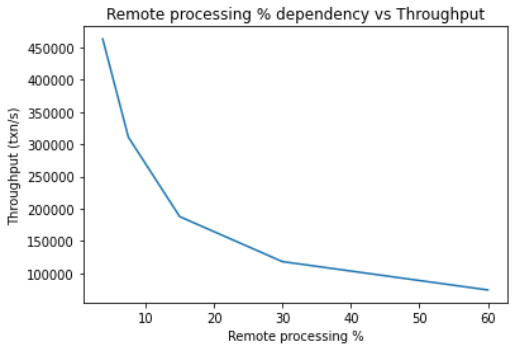


Fig. 11. Throughput vs # of RRD Transactions

C. Profiling Code

As can be seen in the evaluation plots, there is a lot of overhead due to RRD transactions. Profiling the code with perf, approximately 41% of the time was seen spent spinning waiting for future values. That is, 41% of the time, receiving nodes are waiting to receive the appropriate future warehouse tax value and have the ready flag set to true.

V. RELEATED WORK

With modern web services scaling to large number of users and handling large amounts of data, systems like BigTable [3], GFS [4], Hadoop [5] came about. Database begin handling large number of transactions and large amounts of data by smart partitioning for data locality like, databases have wanted to provide maximum performance by running completely in-memory (RAM) such as RAMCloud [6], Redis [7] Memcached, and Resilient Distributed Datasets (RDD) [8]. Database systems then allowed for serializability and then deterministic scheduling to provide even greater performance benefits and reduced complexity.

Caracal operates in-memory to maximize performance and builds off many past systems. Previous deterministic schemes performed epoch initialization in serial resulting in reduced scalability (ex. Calvin [10], Bohm [11]). Caracal on the other hand runs operations in the initialization phase in parallel for scalability. It also builds off Calven's write-sets using renaissance transactions, by performing transactions in read-only first to infer write sets. Calven was an early single-version shared-memory deterministic database that establishes a lock order for each row accessed by a transaction within an epoch during execution, in predefined serial order, by using a centralized lock manager during initialization, which can be a bottleneck. Granola [9] is also a single-versioned partitioned database that splits transactions into pieces based on partitioning scheme, with pieces executed serially on different cores in a predefined serial order. Bohm is a multi-versioned deterministic database. Bohm partitions the keys that need to be initialized across cores during initialization, eliminating the synchronization costs during initialization. The Caracal execution model is similar to Bohm and implements early write visibility similar to PWV [12].

SEDA [13] present a scalable web service by using event-queues at the connecting ends to the network and having threads pulling a batch of events off the incoming queue and operating in stages. Each stage can run independently and in parallel. The paper on SEDA also discussed about schedular fairness and locality, and how lock contention can cause poor performance. Caracal similarly sends transaction pieces to per-core queues to reduce contention and allow for parallel independent execution. However, regarding RRD implementation, it introduces dependencies cross nodes, and as Caracal is deterministic with a specific serial ordering it's hard to process other transactions in the queue while the current thread is waiting for a result from a transaction piece running on another node. It would require processing the read-set of the batched transactions during the initialize phase to identify uncontended read transactions to learn which could execute concurrently or in a different order without affecting the deterministic nature of Caracal.

Shared memory across machines has been looked at previously, an example using virtual memory [14]. However, as Caracal is already operating with shared memory per a node and cross machine virtualization should be equivalent to message passing, the current limitation with RRD implementation might be improved on with RDMA so that receive and send side don't have to wait for CPU to process. DMA could take care of network sending when ready flag is true. Hence, ideas from distributed shared memory (DSM) systems like Munin [15] and TreadMarks [16] don't quite apply to this project, unless a lazy approach was taken to RRD data sending/waiting so that it occurred in batches. However, if it did occur in batches, due to the deterministic order of transactions it might result in deadlocks due to multiple nodes waiting for data and can't progress. This could be performed once no progress is being made, nodes could then batch send RRD data and set ready flags, though how would one node know when another is blocking? This might require a whole redesign or a full investigation in the complete transaction ordering produced by the schedular.

H-Store [17] is a transaction database that partitions across cores and machines. A thread executes per a core and processes transactions that require accessing data only available to that cores partition. As transactions are independent and do not have any dependencies cross-cores it is able to mitigate lock contention or thread coordination and get maximal performance. However, transactions that do require cross core/partition data access or coordination results in serial execution of transactions, limiting concurrency and reducing performance. Caracal has a similar behaviour; however, Caracal is a shared-memory database. As well, in the H-Store paper [18] it states that is does nothing to mitigate overhead of cross core/partition transaction dependencies as the performances gains from avoiding having such transactions, and having majority transactions being independent, offset the performance loss. However, from the results in the evaluation section, in the final design of Caracal such approach is not enough as RRD transactions cause high performance loss in Caracal.

VI. CONCLUSIONS

Remote-read-dependencies (RRD) support was implemented and added to Caracal. The evaluation results matched the hypothesis, that adding RRD would reduce performance due to the overhead of network latencies. However, the affect on the throughput and epoch time were much more significant then expected, with the cluster size further reducing performance by introducing more dependencies. In all, the results showed a linear scaling with number of remote data dependencies. This is a result of the wait function on the receiving side spinning while waiting for data from RRD's.

VII. FUTURE WORK

Originally, after talking with professor Goel, the goal of the project was to modify the schedular to enable lower independent priority transactions to pre-empt a current waiting worker thread when an executing transaction depends on data not currently available. Data could not be available due to transaction read-after-write dependencies on pieces of transactions that have not executed yet, or when a transaction is waiting for data to be received from a remote machine (RRD). However, the group later found out after discussing with the lead architect of Caracal, PhD. Mike Qin, that the RRD feature was not implemented. Hence, the project goal shifted from the original project proposal to the implementation of the RRD feature.

Therefore, the primary area for future work in this project is to reduce the time spent by cores on spinning threads waiting for remote data. Specifically, attempting to extend pre-emption by additionally executing queued transactions of lower priority during a RRD while the thread is waiting/spinning, after attempting to pre-empt with higher priority transactions. This would allow cores to progress should subsequent lower priority (larger timestamp and serial

ordering) transactions not share table rows used by the remote dependency, potentially mitigating the overhead resulting from network delays.

Another area for future work would be utilizing remote direct memory access (RDMA) hardware. Bypassing the CPU when sending and receiving data has the potential to reduce the network overheads of data dependencies

ACKNOWLEDGMENT

Special acknowledgement to PhD. Mike Qin (also goes by Dai Qin) and Professor Ashvin Goel at the University of Toronto. The Caracal database was originally created by Mike Qin and members of Professor Goel's research team have further extended and developed Caracal. Additionally, Mike and Professor Goel closely worked with our group to come up with our project idea, and to help with debugging and understanding the source code (as nothing is commented and not a lot or documentation exists). Mike was very instrumental in working with us to spec out the remote read dependency algorithm, and in the analysis of various deadlocks and bugs within the source code.

CODE REPOSITORY

A. *Original Git Repository*

All code related to this project is contained within the "felis" repository under the "distributed-dataflow" branch.

- git clone git@fs.csl.utoronto.ca:felis.git --recursive

The controller used to initiate and control tests is in the "felis-controller" repository. Not code changes were made to this.

- git clone git@fs.csl.utoronto.ca:felis-controller.git –recursive

B. *UG-Machine – Code Location*

In case access to the private git repo shown above is not possible, the code was pushed to UG machines under "Corey Kirschbaum" project member's account.

- /nfs/ug/homes-4/k/kirschb3/ECE1747ParallelProgramming/Project

The above location has the original and modified code versions so that a diff can be performed between the two.

- felis_orig: This directory contains the original Caracal code.
- felis: This directory contains the implementation code for this project for RRD support.

C. *Code Test Setup Extra Details*

The figure below is a diagram of how the test setup looks like. Each node/machine runs the felis code (contains Caracal database and TPC-C benchmark). The controller is aware of which nodes are in the clusters and starts the TPC-C benchmark on all of them. They each generate transactions and then they are all ready the user tells the controller to notify all nodes to start the test.

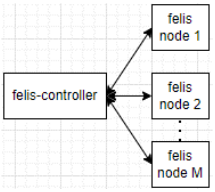


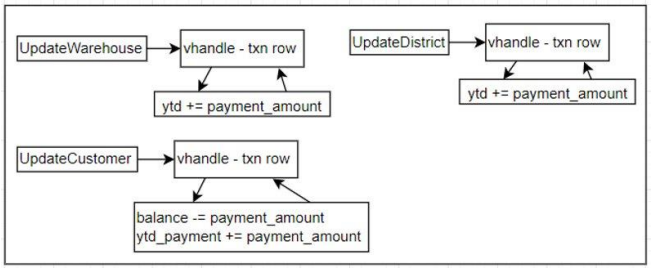
Fig. 12. Test Setup – Code View

The commands used on the felis nodes is seen below. lldb was used as the debugger.

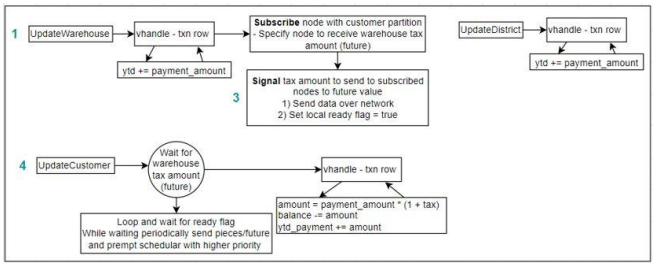
- buck-out/gen/db#<release or debug> -c <controller-ip> -n host<#> -w tpcc -Xcpu16 -Xmem14G -XTpccWarehouses32

APPENDIX

A. *Original Payment Transaction*



B. *New Payment Transaction*



REFERENCES

[1] D. Qin, A. Brown, A. Goel, "Caracal: Contention Management with Deterministic Concurrency Control", SOSP '21: Proceedings of the ACM SIGOPS 28th, 2021, pp. 180-194. https://doi.org/10.1145/3477132.3483591

[2] P. Dewan. 2001. Serializability. Retrieved December 14, 2021 from https://www.cs.unc.edu/~dewan/242/s01/notes/trans/node3.html

[3] 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

[4] 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

[5] J. Dean, S. Ghemawat, "MapReduce: simplified data processing on large clusters," in Communications of the ACM Volume 51, Issue 1, 2008, pp. 107 113

[6] 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

[7] Redis, "Introduction to Redis," [Online], April 2021. Avaliable: https://redis.io/topics/introduction

[8] M. Zaharia et al., "Resilient distributed datasets: A Fault Tolerant Abstraction for In Memory Cluster Computing," in NSDI'12, 2012

[9] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In USENIX Annual Technical Conference. USENIX, 13.

[10] Alexander Thomson, Thaddeus Diamond, Shu-ChunWeng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: Fast Distributed

Transactions for Partitioned Database Systems. In SIGMOD '12. 1–12. https://doi.org/10.1145/2213836.2213838

[11] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. Proceedings of the VLDB Endowment 8, 11 (July 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[12] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via EarlyWrite Visibility. Proceedings of the VLDB Endowment 10, 5 (Jan. 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[13] M. Welsh, D. Culler, E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," SIGOPS Volume 35, Issue 5, 2001, pp.230-243. https://doi.org/10.1145/502059.502057

[14] K. Li, P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," ACM Transactions on Computer SystemsVolume 7, Issue 4, 1989, pp.321-359. https://doi.org/10.1145/75104.75105

[15] J. Carter, J. Bennett, W. Zwaenepoel, "Implementation and performance of Munin," SIGOPS Volume 25, Issue 5, 1991, pp.152-164. https://doi.org/10.1145/121133.121159

[16] P. Keleher, A. Cox, S. Dwarkadas, W. Zwaenepoel, "TreadMarks: distributed shared memory on standard workstations and operating systems," WTEC'94, 1994

[17] D. Abadi, J. Faleiro, "An Overview of Determinisitic Database Systems," Communications of the ACM Volume 61, Issue 9, 2018, pp.78-88. Retrieved December 14, 2021 from https://cacm.acm.org/magazines/2018/9/230601-an-overview-of-deterministic-database-systems/fulltext

[18] R. Kallman et al., "H-store: a high-performance, distributed main memory transaction processing system," VLDB Volume 1, Issue 2, 2008, pp.1496-1499. https://doi.org/10.14778/1454159.1454211