Remote Direct Memory Access as Communication Tier for Disaggregated Database Systems

Andreas Geyer, Alexander Krause, Dirk Habich, Wolfgang Lehner
Technische Universität Dresden
first.last@tu-dresden.de

ABSTRACT

Disaggregated systems form a new trend of composable hardware. Efficiently leveraging this kind of volatile hardware requires a database system to be flexible on all layers, especially in data transfer. Within this paper, we present our view on database systems using disaggregated hardware. Further, we will show how the fast and reliable *Remote Direct Memory Access (RDMA)* interconnect can be used to define an efficient communication layer and how multi buffering can solve performance problems in contrast to traditional single buffer messaging.

Keywords

RDMA, Disaggregated Systems, Message passing

1. INTRODUCTION

The amount of data that needs to be highly available grows larger every day. Therefore, many companies tend to store their data in any kind of database system. While the amount of data grows, the access time should stay as small as possible. To achieve this, many database systems keep the highly available data completely in main memory. These systems are also called in-memory database systems.

Fulfilling this ever-growing need for main memory with a reasonable cost in energy and money becomes increasingly difficult. It is possible to have scale-up systems with several Terabytes of main memory, but these are very expensive to buy and operate [5]. However, traditional scale-up systems suffer from two core problems: (i) they have to be highly utilized most of the time to amortize their cost and (ii) they can only scale to the initially bought hardware. If actual workload peaks exceed this limit, there are no spare resources to utilize. On the other hand, the classical scale-out system is more favorable in terms of upfront costs and scalability, which makes it a more suitable approach for data centers. This approach allows every worker node to consist of different hardware components, which allows for a bit more flexibility, let alone the possibility to enable or disable additional machines, according to the current workload requirements. However, the scale-out systems can suffer from underutilized hardware as well [3]. It is possible to shutdown unused nodes, but if a node is running, it generates cost in any form. Thus, scale-out systems provide a higher degree of freedom in terms of customizability.

The principle of *disaggregated systems* takes this approach to a new level, by improving on the available degree of freedom but keeping the cost proportional to the used resources. The basic idea is to move away from the traditional Von-Neumann-Architecture in a way, that hardware components are stored in disjunct units. Thus, there are server racks solely consisting of CPUs, GPUs, RAM, SSDs, etc., which allows a system admin to freely assign resources, based on the current needs. With this configurability in mind, it is possible to e.g. dynamically allocate or deallocate CPUs, if more or less compute power is needed for a given workload.

One approach for such a separation of components is proposed in [7], where CPU, RAM and storage are separated and connected via network. The authors proposed an own operating system to manage the disaggregated hardware. However, this disaggregation approach brings several challenges with it. First, there has to be a fast and reliable communication interface between the components to achieve a high overall performance. After all, the common PCI interconnect is considerably faster than Ethernet and every other method will be compared to it. Second, the widespread resources have to be managed by some governing instance. This includes the distribution of resources to the processes, the logical pairing of components for processes and the management of the components themselves.

Within this paper, we propose our vision of a database system architecture, based on disaggregated hardware. To achieve this, we present our core contributions as follows:

1. First, we present our envisioned system architecture with its necessary components in Section 2.
2. Further, we tackle the first challenge by providing our implementation ideas for a fast and reliable communication layer. Thus, we explain the benefits of RDMA[1] and our (multi-)buffer messaging interface in Section 3.
3. Third, we prove the viability of our approach by thoroughly evaluating different communication buffer combinations in Section 4.

This paper is then concluded with a discussion of related work and a summary in Sections 5 and 6, respectively.

33th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 08.06.2022, Dresden, Germany.

[1]`https://docs.nvidia.com/networking/display/RDMAAwareProgrammingv17/RDMA+Aware+Networks+Programming+User+Manual`

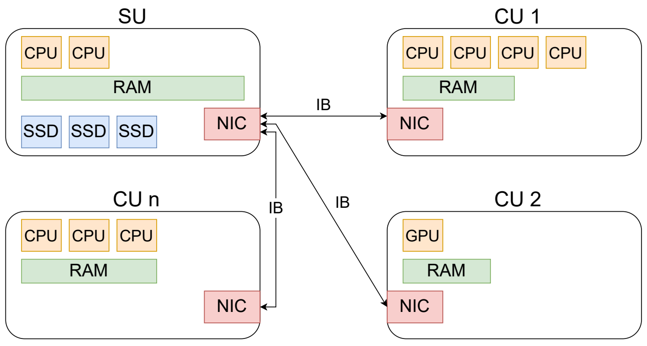


Figure 1: Draft of our disaggregated system architecture.

2. SYSTEM ARCHITECTURE

In this section, we introduce our vision of a composable data processing system, based on disaggregated hardware. As outlined in Section 1, we target a hardware environment, in which we can allocate and deallocate components like additional CPUs or storage units. Without loss of generality, we assume the existence of two distinct unit types, namely a *compute unit* (CU) and a *storage unit* (SU), with the overall architecture being sketched in Figure 1. We understand a CU as a piece of hardware, that consists of a number of CPUs and some attached main memory, i.e. DRAM. The CU is responsible for all the heavy lifting, e.g. query processing for a database system. Contrary, the SU mainly consists of storage media, such as SSDs. The SU holds all the data and provides requesting CUs with the necessary information. Obviously, the SU is also equipped with a CPU and some DRAM, however the computational power of an SU is very limited, compared to the CU.

A composable system consists of at least one CU and at least one SU, however usually multiple CUs and only very few SUs are present. The units are connected through a high performance interconnect, such as Infiniband (IB) via *Network Interface Controllers (NICs)*. CUs and SUs communicate via this interconnect using RDMA, which allows for high bandwidth data exchange and the eponymous remote direct memory access. Generally, additional CUs can be enabled or disabled, if the current workload requires additional or fewer resources, respectively.

Data processing systems require a multitude of components or layers, e.g. storage, communication or processing. One of the most crucial tasks is getting the data from the storage to the actual computation. Scale-Up and Scale-Out systems usually employ the near-memory processing paradigm, to reduce inter-socket or inter-machine data transfer. However, a disaggregated system featuring our envisioned topology does not exhibit the luxury of completely avoiding data transfer between units. This implies, that a CU has to fetch data from the SU, for every query that is being processed, if no particular caching optimizations are applied. We thus conclude that an efficient inter-unit communication is crucial for the overall system performance. Therefore, as a general and crucial building block, we focus on the development of a communication strategy via RDMA for data systems in this paper.

2.1 RDMA Basics

As previously mentioned, RDMA is a way of accessing the main memory of another unit via the IB connection. There are two possibilities on how to use RDMA. The first one is a Send-Receive type of communication, described as two-sided-communication. It involves the CPU of both sides to communicate data, since both sender and receiver need to issue a send or receive request, respectively. The primitives for that are RDMA SEND / RECEIVE. In contrast, the one-sided-communication uses the RDMA WRITE / READ instructions, which only involve the CPU of one side, i.e. the one issuing the instruction. Both have their own advantages and disadvantages.

To reduce the overall influence of the communication on the CU, we first implemented the one-sided communication scheme and leave the two-sided version for future work. The basic principle of RDMA is, that every participant has to create a local buffer and register it as memory region. The involved machines will then have to exchange some metadata, e.g. the buffer pointer, identifiers, keys, etc. to be able to read from or write to the memory region of the other side. This initial handshake is usually performed via a TCP connection, which requires some sort of a client/server architecture. Conceptually, both CU and SU could play either role, however we figured that the SU is more suitable as a *server*. That is, since the SU would wait for incoming requests of newly enabled CUs to register their RDMA buffers accordingly. After successfully exchanging the meta information, the TCP connection is closed, and all following communication is done via RDMA through the created buffers.

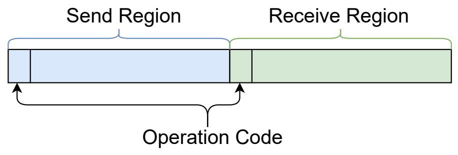


Figure 2: Single RDMA communication buffer layout.

2.2 Buffer management

Intuitively, we would create a single zero-initialized buffer, that is used for both sending and receiving data. For our initial testing, we employed the buffer layout as shown in Figure 2. To allow a two-way access, the buffer is split in half, yielding a sending and receiving part. A receiving unit would have to read its receive buffer part and probe, if it contains any usable information. As soon as the first 8 Bytes of the buffer, i.e. a processor word of a 64-bit system, contain something other than a zero-value, we can start reading that message and consume it. However, we could then potentially read an in-flight message, which is still in the process of being written, because the local read bandwidth is obviously higher than the RDMA write bandwidth. To circumvent this issue, we introduced the *operation code* as the very first byte of both the send and receive region. The local reader only has to check that first byte to realize the reception of data, since this byte can be atomically written after the whole message has been transferred.

However, the single buffer approach inhibits the potential data transfer performance. If we only write one package to the receiver, no issue arises. As soon as multiple packages have to be transferred, the sender has to wait for the receiver to process it, clear the receive buffer and notify the sender, that a new package can be sent. To avoid the implied

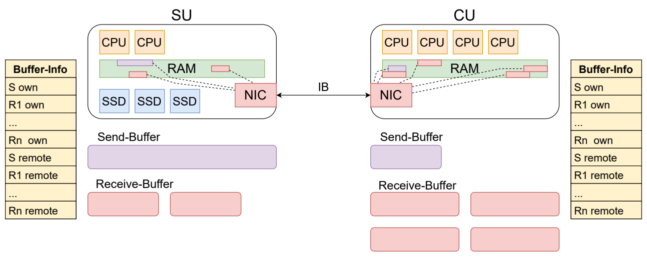


Figure 3: Multi buffered RDMA based communication.

downtime, we follow the well-known double or multi buffer approach, with concrete details being given in the following Section 3.

3. IMPLEMENTATION DETAILS

In this section, we will elaborate on how we facilitate the actual multi buffering with RDMA. First, we will explain the communication architecture. After that, we will give an outlook of the integration possibilities with existing systems or prototypes.

3.1 Multi Buffer

The multi buffer approach, as the name indicates, includes more than one RDMA buffer for each unit. Figure 3 depicts a schematic overview of the components of our multi buffer RDMA communication tier. In contrary to the mentioned naive approach of a single buffer on each unit, the multi buffer approach holds three types of buffers on each site. It is important to mention that every connected pair of SU and CU has their own buffers. That is, the amount of allocated buffers can vary between SU and CU.

On each unit there is exactly one *Send-Buffer (SB)*. The name indicates that the task of this buffer is to send data to the remote machine. Its size can vary from unit to unit. Its working principle is, that the data which has to be sent, is copied into the SB. Everything that is inside the SB can be sent to the connected unit to which connection this SB is registered. As the name multi buffer indicates, we additionally allocate several *Receive Buffers (RB)*. In Figure 3 this is shown, as the SU has 2 RB and the CU has 4 RB. Eponymously, the RB is the memory region where the remote SB writes its data. When the transmission from the SB to any RB is complete, the data inside the RB can be consumed by the receiver. If the data is larger than 1 remote RB, it is sent as multiple packages iteratively.

The third type of buffer is different from SB and RB. While the former two are responsible for the actual data transmission, the *Buffer-Info (BI)* is responsible for controlling the interaction of the SB and RB. It is a fixed-sized array with one entry for every buffer on the local and the remote size. This buffer is also registered as an RDMA-accessible memory region, which is readable and writable by the connected units. For now, we limit ourselves to a BI size of 16, which means we can allocate up to seven RB on every side plus the mandatory SB. However, this could lead to unused entries, if there are less than seven RB on one side. The BI is used to store information about the current status of every registered RDMA buffer of its connection, e.g. if an RB is currently free to use. The size of an RB is arbitrary, as long as there is enough main memory. However, every RB at a machine is of the same size. This size can vary between the machines. Similar to the RB, the size of the SB can be freely chosen. Nonetheless, we propose and use an integral multiple of the remote RB size. That is, for 4 RBs with, e.g. 1 KiB on the CU side we would allocate a SB of 4 KiB on the SU side. Figure 3 illustrates the usage of different buffer sizes per unit. We chose a larger SB size for the SU than the CU, because we see a higher transfer directed from the SU towards the CU than vice versa.

The transmission process starts with checking whether the own SB is ready, by looking at the corresponding BI entry. If it is ready, it is possible to send and the entry in the own BI is changed to indicate, that this SB is currently unavailable for other data sending requests. Data transfer is then performed as message passing. A message consists of the operation code, some meta data and the actual payload. For now, the meta data contains the size of the current payload as well as the total size that will be transferred. Therefore, this meta information is calculated and inserted into the SB together with its corresponding piece of data. The SB will contain as many messages as it can fit, based on its size and the amount of remotely available RB. After preparing the message, we scan the local BI for any available RB on the remote side. Upon finding an entry, its status will be set to unavailable and the actual data transfer, i.e. through using the *IBV_WR_RDMA_WRITE* instruction, begins. Afterwards, the local and remote BI entry for the RB is set to a status that indicates that the transfer is done. Due to the multi buffer approach it is not necessary to wait until the chosen RB is ready again. The next data package can be sent to the next free remote RB, which can be identified through the local BI.

When awaiting an incoming data transfer, the respective unit polls its local BI to check for any RB, that has the corresponding status set to *data ready* and proceeds to consume that message. One possibility would be to copy the data from the RB to a memory region that is not allocated for RDMA purposes. After the data is consumed, the local and remote BI entry for this RB is set to ready again.

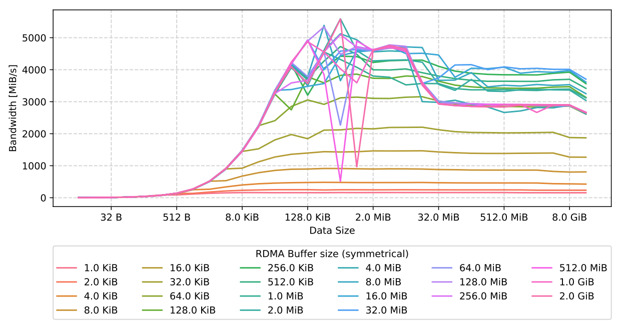
3.2 Integration

To make this communication tier as easily usable as possible it is planned to release it as a self contained C++ library with a well defined API. This API is designed to take away the low-level networking, buffer creation and management topics from the user by abstract methods in a send/receive manner. However, it is still possible for the user to configure for example the number and size of the RB. This offers the flexibility of adapting the communication tier to the own workloads, but also the transparency of not having to deal with the pure RDMA handling for example.

This library approach aims at creating a simple way of using this RDMA based communication tier in a variety of applications. It is desired that it is as easy as possible to exchange the existing network communication tier of a database system with our RDMA based one as long as there is an existing IB connection between the systems.

4. EVALUATION

To show the overall applicability of our approach, we conducted several experiments for both the single buffer and the multi buffer approach. For our experiments, we connected two machines equipped with an Intel Xeon Gold 6130 (*Ser-*


(a) Throughput test for the single buffer approach.

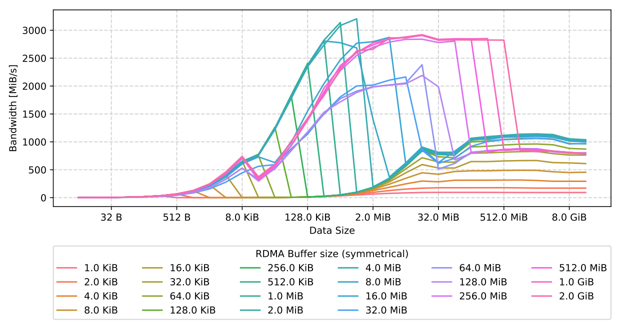

(b) Consume test for the single buffer approach.

Figure 4: Evaluation of the single buffer approach.

ver1) and Intel Xeon Gold 5130 (*Server2*) with 3.7 GHz and 3.2 GHz peak frequency, respectively. Both servers are equipped with 384 GB of main memory. The infiniband connection is realized through two Mellanox ConnectX-4 cards with up to 100 GBit/s. Both servers run with Ubuntu 20.04 and employ opensm version 5.8 as subnetmanager. We repeated every experiment 10 times and averaged the results. Note that we calculate the reported effective bandwidth based on the wall clock time. That is, we do not account for additional copy operations that effectively increase the overall data that has to be processed but divide the original requested data by the overall time spent for processing. Thus, our reported bandwidth could never reach the theoretical maximum of 100 GBit/s provided by our infiniband cards.

4.1 Throughput Test

The throughput test aims on measuring the raw writing performance of our implementation in combination with the available hardware. This provides a theoretical peak performance. For this experiment the data is written from DRAM to the SB and then to an available remote RB without waiting on the consumption or acknowledgement of the CU.

Figure 4a shows the results for the single buffer approach as defined in Figure 2. The x-axis shows the requested data size in bytes, the y-axis denotes the effective bandwidth. Every line corresponds to a total buffer size, however since the buffer is split into a send and receive region, each of them only features half of the reported size. A general take-away is that very small buffer sizes of less than 64 KiB perform poorly, due to a high number of `memcpy` operations for larger data sizes. For this experiment, we achieve a peak bandwidth of 5615.4 MiB/s using a buffer size of 2 MiB. As the requested data is copied from DRAM to the RDMA buffer, and then sent to the remote side, we actually process twice the size of the original data. Thus, if we multiply the achieved bandwidth with this factor 2, we get an approximated RDMA bandwidth of 11 230.8 MiB/s, which is relatively close to the specification of the network cards.

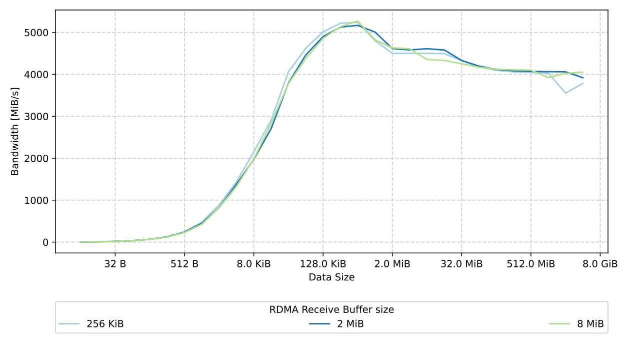
Figure 5a shows the results for our multi buffer implementation. As shown earlier, plenty of the observed buffer sizes perform poorly. This issue is further enhanced by the necessity of finding a good SB-to-RB-ratio in terms of number and size. Thus, to find the best possible configuration we iterated over 1 through 7 RBs and varied the size of the remote RB. The size of the SB is calculated by the number of remote RB multiplied with the size of the remote RB. As buffer size for every remote RB all powers of two, ranging from 2^{10} to 2^{29}, i.e. 1 KiB up to 512 MiB, were tested. In this experiment, we vary the data size from 8 Byte up to 4 GiB for every combination of RB count and size.

To reduce the visual noise, we show the best 3 curves according to the integral under the curve in Figure 5a. We have chosen the integral over the overall highest peak, since it better represents the more robust configurations, and it is less prone to a single high performing outlier. The figure shows the results for the throughput test with 1 SB and 1 RB. On the x-axis the total size of the data is shown, while the y-axis indicates the speed of the transmission in MiB/s.

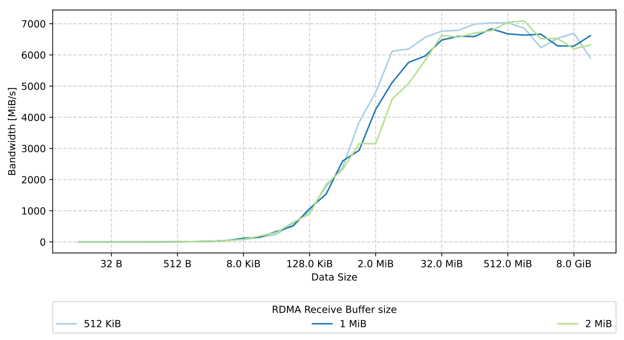
This experiment has shown that the maximum achievable transmission speed with our configuration is at around 5250 MiB/s for all three depicted curves. In this case, this configuration is also the one with the highest peak. Most of the other RB sizes lead to a curve with a significantly lower peak performance. It is recognizable that the curves rise quickly up to 128 KiB of total requested data size and the optimum is reached by all shown RB sizes at a total requested data size of 512 KiB. With total package sizes greater than the optimal 512 KiB the performance decreases. This is due to two major facts. Firstly, only the size of the actually requested data is taken into account when calculating the performance, but for every message that is sent, there are several bytes of meta data added. Therefore, if the total data size is greater than the RB size, it is split into several smaller messages, each with its own block of meta data. Hence the more packages are sent, the more untracked meta data is transmitted, too. The second reason is, that with growing size of a single message, the overhead of the RDMA layer, e.g. its latency, gains impact. It is worth noting, that an RB size in between these three best ones was not significantly worse in performance. Significant differences were seen for RB sizes smaller than 128 KiB or larger than 8 MiB.

4.1.1 *Multi-threaded throughput test*

In this experiment, we want to check if we can fully saturate the theoretical bandwidth and thus we employ multiple threads for sending data to the requesting unit. To achieve this, we spawn an individual thread for every existing remote RB. Hence the SB size is again the number of remote RB times the size of the remote RB. This allows every sending thread to fully use its own part of the (logically partitioned) SB to write without interfering with the other sending



(a) Throughput test with 1 RB and three best performing RB sizes according to the area under the curve.



(b) Multi threaded throughput test with 3 RB and three best performing RB sizes according to the area under the curve.

Figure 5: Evaluation of the experiments on the performance of our RDMA communication tier.

threads. Again, the receiving unit does not acknowledge or consume the received data in any way. All threads work on the same requested data, but each thread is responsible for its own part, if the total requested data size has to be split to fit the remote RB. Again, we iterated over the size and the number of RB and thus also the number of sending threads.

Figure 5b shows the results for the best performing buffer sizes using 3 RB. The results are that the optimum of around 7100 MiB/s is significantly higher than the optimum of the single threaded experiment. However, this experiment also revealed, that this advantage is only given at large message sizes of at least 64 MiB. Otherwise, the transmission speed is worse than the single threaded case. For a total data size of 512 KiB, which was optimal for the single threaded case, the multi threaded version with 3 RB achieved around 3000 MiB/s and therefore significantly less. We argue that the general threading overhead impedes the performance for such small messages. Further increasing the amount of RB and sending threads accordingly, we can observe a flatter curve with less dents. However, all of them approximately achieve the 7100 MiB/s bandwidth.

4.2 Consuming Test

We designed the consuming test to better reflect actual database workloads. In this experiment, every RB has to be cleared by its owner, before the sender can write to it again. Implementation-wise, the difference is that the SU signals the CU via the BI that an RB is ready to be consumed after writing to it. For this test *consuming* means, that the data is copied to a location in the main memory of the CU, which is not registered as RDMA buffer. However, it is also possible to use the data from the RB directly without copying it to any other location. Subsequently the RB is `memset` with 0 and signaled as ready to the SU to be used again. This does also imply, that this RB is blocked for further writes until the CU resets the BI status to ready again.

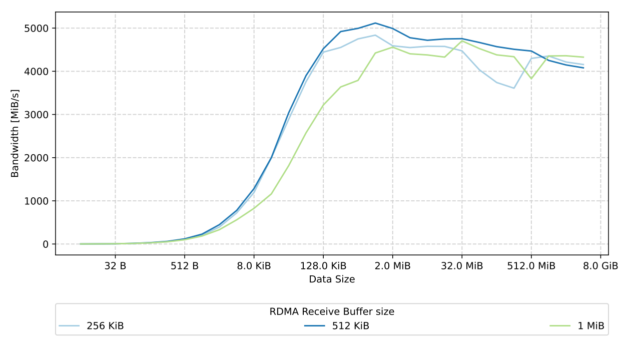
The single buffer experiment is shown in Figure 4b. Remarkably, there is a huge performance drop for every curve just as the total requested data size equals half the RDMA buffer size. At this point, we have to send at least two messages to transfer all the data, because of the required meta data, that is added to every message. Thus, the sender has to wait for the receiver to make the receive region of the single buffer available again, before sending the next message. This is where the multi buffer approach can yield its benefits, since it can already use the next free buffer.

Figure 6a depicts the three best curves for the multi buffer version of this experiment using 2 RB. Other than for the throughput test, the curves differ more from each other and the RB sizes smaller than 256 KiB or larger than 1 MiB perform significantly worse. The optimum is around 5100 MiB/s and therefore slightly lower than in the throughput test and it is reached at a total data size of 1 MiB instead of 512 KiB. For 3 or 4 RB the results are slightly, but not significantly better. For 5 up to 7 RB there is virtually no difference to 4 RB. This result is achieved and only slightly different from the throughput results because the consumption method of our implementation is at least as fast as writing to an RB. This means every time the sender is done with writing, another RB may have just become ready again and there is near to no idle time for the sender.

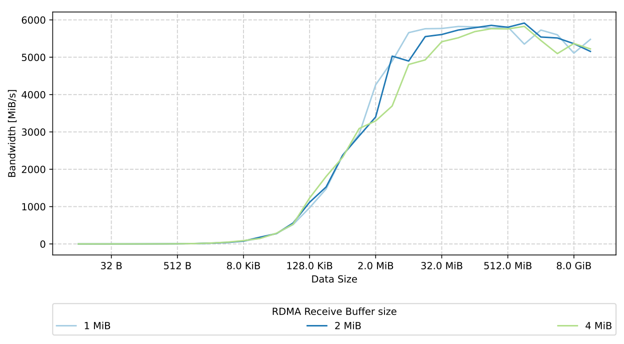
4.2.1 *Multi-Thread*

Again, we executed this experiment with multiple threads and report selected results in Figure 6b. However, the implementation of this multi-threaded test is different than the multi-threaded throughput test. For this test every thread is sending to at least 2 remote RB and every receiving thread works on at least 2 RB. This is because the checking whether an RB is ready is done by polling on the corresponding BI entry. To achieve the maximally possible performance, this is implemented as busy waiting using a lock on the entry to avoid read-write-conflicts. However, busy waiting on only one entry means that this entry is never lockable by the sender, due to the larger read latency of the remote unit. Thus, the remote side is not able to write this entry and indicate that the data transfer is done. According to this, there are at least 2 RB for each thread to force the unlocking of the BI entry. The same issue occurs on the sender side where the receiving unit would not be able to indicate that this RB is ready to receive data again. This means for the sending side that the SB size does not have to be $\text{num}_{RB} * \text{size}_{RB}$ as before, but instead $\text{num}_{\text{threads}} * \text{size}_{RB}$.

Figure 6b depicts the best performing configuration. We use 6 RB on the remote side. This implies 3 sending threads on the local side and 3 receiving threads on the remote side. The best achieved result is a bandwidth of around 5900 MiB/s. This is approximately 14 % more than the 5100 MiB/s of the single threaded consuming test, but signi-



(a) Multi buffer Consume test, single threaded, 2 RB.



(b) Multi buffer Consume test, multi threaded, 6 RB.

Figure 6: Evaluation of the experiments on the performance of our RDMA communication tier.

ficantly less than the maximum of around 7100 MiB/s from the multi threaded throughput test. Similar to the multi threaded throughput test, the curve grows slower and reaches its maximum at a larger data size. We argue that the management overhead for the multi threading on sender as well as on receiver side impacts the result and prevents us from achieving the theoretical maximum.

5. RELATED WORK

Protobuf [2] is a language-neutral and platform-neutral way of serializing structured data. However, its portability comes with high abstraction and thus also overhead for simple operations. We copy data columns with some meta information and thus a more hand-tailored approach is necessary to preserve performance.

In [4] the authors show an approach similar to our own work. While they did not focus on a certain use case and evaluated their approach on smaller message sizes, we built our tests with the database use case in mind and therefore, with larger possible message sizes, buffer sizes and combinations of send and receive buffers.

While we worked with the standard RDMA implementation in our experiments, D-RDMA [6] has recently offered a way to achieve higher throughput and less CPU usage with an RMDA extension. However, our use case foresees the usage of RDMA as transport layer from a data source to a data sink. Thus, we do not face message fragmentation as e.g. expectable for distributed joins and the like.

Our approach shares several similarities with L5 [1]. However, our system model follows a client-server design, and we rely on RDMA to exchange contiguous data, such as data columns and intermediates. Thus, no online switching between communications is necessary, which could harm the overall performance.

6. SUMMARY

We have introduced our vision on how disaggregated system components can be composed to be usable for a database system. In this paper, we presented our RDMA based communication layer for such disaggregated database systems. We have shown the viability of our implementation and that we are able to efficiently use the RDMA interconnect. Further, our evaluation identified potential pitfalls, such as idle times, if only a single communication buffer is used. Our multi buffer concept is able to alleviate the performance drops, which arise when sending more than one package through a single buffer.

From here on, we want to pursue multiple paths. First, we will develop our communication layer into a shared library and make the code publicly available, e.g. on github. Following that, our vision is to have a deeper look into leveraging the DRAM of the CU. That is to efficiently cache already requested data and make it available to other local workers, e.g. other CPUs that are executing queries on the same CU, to save on the limited RDMA bandwidth.

7. REFERENCES

[1] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-latency communication for fast DBMS using RDMA and shared memory. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE.

[2] Google, Inc. Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[3] B. Grot, D. Hardy, P. Lotfi-Kamran, B. Falsafi, C. Nicopoulos, and Y. Sazeides. Optimizing data-center TCO with scale-out processors. *IEEE Micro*, 32, 2012.

[4] P. MacArthur and R. D. Russell. A Performance Study to Guide RDMA Programming Decisions. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012.

[5] M. Mahloo, J. M. Soares, and A. Roozbeh. Techno-economic framework for cloud infrastructure: a cost study of resource disaggregation. In M. Ganzha, L. A. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017, Prague, Czech Republic, September 3-6, 2017*, volume 11 of *Annals of Computer Science and Information Systems*, pages 733–742, 2017.

[6] A. Ryser, A. Lerner, A. Forencich, and P. Cudré-Mauroux. D-RDMA: Bringing zero-copy RDMA to database systems. In *CIDR*, 2022.

[7] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, Oct. 2018.