D-RDMA: Bringing Zero-Copy RDMA to Database Systems

André Ryser[1] Alberto Lerner[1] Alex Forencich[2] Philippe Cudré-Mauroux[1]

[1] eXascale Infolab – University of Fribourg, Switzerland
[2] Dept of Electrical and Computer Engineering – University of California San Diego

ABSTRACT

The DMA part of RDMA stands for *Direct Memory Access*. It refers to the ability of a network card (among other devices) to read and write data from a host's memory without CPU assistance. RDMA's performance depends on efficient DMAs in the initiating and target hosts. In turn, a DMA's cost is almost always proportional to the length of the data transfer. The exception is small DMAs, which suffer from high overheads.

In this paper, we show that database systems often generate small DMA operations when using RDMA canonically. The reason is that the data they transmit is seldom contiguous by the time transmissions occur. Modern databases avoid this problem by copying data into large transmission buffers and issuing RDMAs over these buffers instead. However, doing this requires a substantial amount of CPU cycles and memory bandwidth, forfeiting RDMA's benefits: its *zero-copy* feature. To solve this issue, we introduce D-RDMA, a *declarative* extension to RDMA. D-RDMA is declarative in that it specifies what data to transmit but not the DMA schedule to do so. The approach leverages a smart NIC to group data fragments into larger DMAs and produce the same packet stream as regular RDMA.

Our experiments show that the network throughput can increase from 18 Gbps per CPU core to up to 98 Gbps (on a 100 Gbps card) with virtually zero CPU usage when replacing RDMA with D-RDMA in a typical data shuffle scenario. We believe that D-RDMA can enable a new generation of high-performance systems to take full advantage of fast networking without incurring the usual CPU penalties.

1. INTRODUCTION

Databases can significantly benefit from fast networking [6], and RDMA-based networks such as Infiniband/RoCE have arguably been the most common way to deliver such performance [12, 13]. The central premise of RDMA is that the network card can autonomously move data in and out of an application's memory without involving the host's CPU. For instance, to send query results back to a client, a database points to the data that answers the query and asks the card to fetch and transmit that data. This arrangement does not involve copying the data into the kernel, as a traditional TCP/IP stack would have done, which is why it is said to be a *zero-copy* protocol.

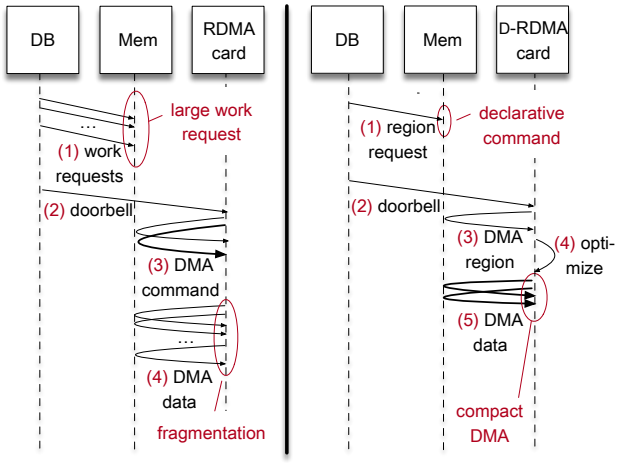

Figure 1: Left: standard RDMA forces the database to enqueue work requests for every fragment to be transmitted (1). Once enqueued, the card gets notified that new commands are waiting through a mechanism called a *doorbell* (2). The card proceeds by pulling the large commands from the submission queue (3). It then transfers one fragment at a time, regardless of optimization opportunities (4). Right: D-RDMA compactly declares the regions to transmit (1) and notifies the card the same way (2). The card pulls much shorter commands from the queue (3). In turn, the card looks for fragment coalescing opportunities and performs much larger DMAs should the opportunity arises (4, 5). The packets produced by the two approaches are identical.

The problem. In practice, however, RDMA can be very CPU intensive if the data to transmit is scattered in many non-adjacent, small chunks. In that case, the database has to point to each chunk by filling some RDMA control data structures as follows: for each message, it has to create a Work Request (WR), and for each data chunk within that message, a Scatter-Gather Element (SGE). If the chunks are relatively small, filling these control structures, and transferring the small chunks which they refer to, can dominate the cost of an RDMA. Figure 1 (left) depicts this scenario.

12th Annual Conference on Innovative Data Systems Research (CIDR '22) January 10-13, 2022, Chaminade, USA.

Unfortunately, fragmentation is a common phenomenon occurring in both OLAP and OLTP database systems. Let us consider OLAP databases first. A common operation in those databases is the data shuffle operation [18]. It distributes a potentially large number of rows across a set of servers. By definition, each row is destined to a different server than the previous one and, therefore, every row requires a unique WR/SGE set. The resulting WR/SGE list size is likely the same as the number of rows to transmit, which requires substantial CPU cycles to build for large shuffles.

Fragmentation is just as common in OLTP databases. These databases transmit data at the end of a query, when usually a small number of resulting rows are selected. These rows, however, may be composed of attributes that are not adjacent. For instance, queries may project out some of a base table's attributes. This leaves gaps from a transmission point of view in what could have been otherwise a contiguous row. The individual OLTP responses are not heavy to transmit, even if the gaps are present, but these kind of systems deal in volume. If each response requires additional control structures, the system ends up bleeding performance in the aggregate.

We will explore those scenarios in detail shortly in Section 2 but can already state that they are an inescapable consequence of the current RDMA API. Ultimately, forcing an application to describe chunks of data individually is the reason why high-speed networking typically entails high CPU costs for data-intensive applications.

Database systems deal with fragmented data by using transmission buffers [16, 22]. These are contiguous areas into which the database copies data fragments before transmitting them as one WR/SGE. The rationale is that it may be worth copying data to large contiguous regions rather than building extensive WR/SGE lists. However, such copying also causes CPU overhead—different than filling WR/SGEs but overhead nonetheless—and consumes significant memory bandwidth, which is considered the main bottleneck of high-performance databases [1, 7, 9].

The problem worsens with increasing network speeds, as it takes more CPU cycles to fill transmission buffers at faster rates. Networking speeds are only increasing, with 100 Gbps links being standard in data centers at the time of writing, and with 400 Gbps equipment being already available off-the-shelf [4]. Furthermore, the 800 Gbps Ethernet standard was ratified in 2020, and ongoing discussions over the 1.6 Tbps standard are expected to complete before 2025. At this pace, increasingly more CPU cycles will be diverted from database processing to keep up with networking. We argue that we can address this problem by rethinking the RDMA interface.

The D-RDMA Extension. The starting point of this work is an investigation of RDMA's performance on fragmented data scenarios typical in database systems. We use a PCIe logical analyzer to instrument how the network card and the host interact[1]. The analyzer allows us to capture PCIe traffic traces—the live DMA—between the card and the host without affecting speed.

These traces give us information about the impact of fragmentation. For instance, they indicate that the card strictly executes one DMA for each SGE, regardless of how small the SGE area is and whether two or more SGEs areas could be coalesced in a single DMA. This implies that the WR/SGE list serves as a *de facto* DMA schedule. For another instance, the traces reveal that some data fragments are often adjacent or that the gaps between these fragments can be small and occur in patterns.

Armed with these insights, we propose a method to *declare* larger areas that contain both data to be transmitted and gaps. We do so by using alternative structures to WR/SGEs that represent these areas in a much more compact way. The first benefit of our approach is that these more compact control structures save the CPU time that would otherwise go into building and sending large requests to the card. Figure 1 (right) depicts this scenario.

However, our biggest gains come from another feature. In contrast to WR/SGEs, our proposed control structures do not mandate a given DMA schedule. The card is free to devise an optimal DMA strategy to bring data (and gaps!) from the host. For example, if the card finds two adjacent fragments within the large declared area, it can pull their data in a single DMA operation. We call our extended version D-RDMA to reflect its *d*eclarative approach vis-a-vis the DMA schedule.

Preliminary Results and Contributions. We experiment with the key component of our solution, varying DMA sizes, and observe the effects of our techniques under the analyzer. We found, for instance, that a naive shuffle can take 100% of one CPU core and generate a meager 18 Gbps worth of data on a 100 Gbps network. Using our method, the CPU cost to initiate the transmission drops to virtually zero, while the throughput rises to 98 Gbps. We note that the packet sequence generated in both cases is the same, showing how central the DMA schedule is in terms of performance. We obtained similarly encouraging results with different scenarios of data layouts and query operations.

This paper is an early report on the design and implementation of D-RDMA. In summary, our main contributions and the rest of the paper are as follows. We characterize the DMA schedule between a host and a network card and identify fragmented data transmissions as a key bottleneck (§ 2). We propose an extension to RDMA that declares the data to be transmitted in a compact, optimization-friendly way (§ 3). We continue by presenting the architectural changes a traditional NIC requires to support our extensions (§ 4). We validate our proposal's viability through several experiments (§ 5). Next, we lay out the research agenda necessary to fulfill our vision (§ 6) and position our work with respect to other related efforts (§ 7). Finally, we close the paper by presenting some conclusions (§ 8).

2. CHARACTERIZING FRAGMENTATION

Database Systems strive to represent data in memory in ways that foster performance. They do so by using different data layouts, each adapted to a different kind of queries [2, 7]. However, by the time databases transmit (intermediate) results, the data may contain gaps. We posit that these gaps occur frequently at transmission time. We verify this assumption by instrumenting typical OLAP and OLTP operations using special hardware.

[1]We use a Teledyne-Lecroy PCI Protocol Analyzer with a PCIe Interposer Card.

2.1 Shuffle in an OLAP System

Consider an OLAP database in a columnar format that uses a set of servers, each storing a horizontal partition of the data. OLAP queries often reshuffle the data, redistributing tuples across a set of servers. From a sender's point of view, this means building one message (WR) per row. Each message would point to as many values (SGEs) as there are columns in the table to be shuffled. The top of Figure 2 depicts this scenario.

We run this scenario using a state-of-the-art 100 Gbps network card and obtain traces with the PCI analyzer we mentioned above. The relevant part of the trace appears as a screen capture at the bottom of Figure 2. Some explanations are necessary to interpret the trace information.

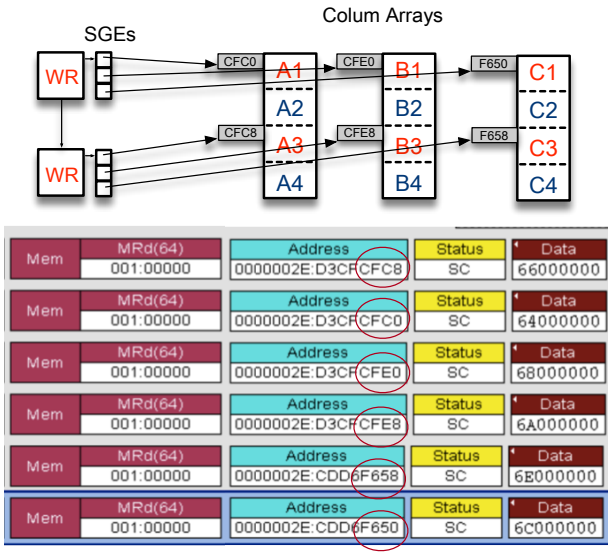

Figure 2: A typical OLAP work request. Top: the tuples $\{A_1, B_1, C_1\}$ and $\{A_3, B_3, C_3\}$ should go to the red server, while $\{A_2, B_2, C_2\}$ and $\{A_4, B_4, C_4\}$, to the blue one. The red server's WR chain is shown here. Bottom: a screenshot of the analyzer showing that the card issued six MRds (first column), as the WR specified. The data was transferred 4 bytes at a time (last column).

Applications use RDMA through an API called *Infiniband Verbs* [13]. This API exposes the WR and SGE data structures to applications, which use them to express what they wish to transmit. When the application hands in a WR to the verbs API, the WR and the referenced SGEs are turned, in user space, into a device-specific work queue element (WQE) [15]. This WQE is then placed into a work queue, where the WQE will be fetched by the card via the PCIe system [14]. Upon receiving a WQE, the card fetches, again via the PCIe system, the data to which SGEs in the WR pointed to. It is those back-and-forth messages that we observe using the analyzer.

The PCIe system is itself a networking system. The addresses in this network depend on the position (the slot) in which a card sits. The host has a privileged address called *Root Complex*. Using such addresses, hosts and peripherals can exchange network packets, called Transaction Level Packets (TLPs). There is one peculiarity of PCIe that is relevant for our purposes. TLP packets have *operations* associated with them. The two operations that we are interested in here are *MemoryRead* (MRd) and *CompletionWithData* (CplD). The card issues the former against the root complex when it wishes to read portions of the memory host. The host—or more precisely, the host's memory controller—responds with the latter message type, sending the data requested to the card. To put it simply, a DMA Read operation, at a high level, *is* this exchange of MRds and CplDs. What we see at the bottom of Figure 2 are the pairs of read requests (MRds) along with the data the server sent. The analyzer pairs the CplD packets (the data) with the originating MRd for presentation purposes.

As could be expected, the analyzer shows that the card interpreted the WR literally and issued one DMA per SGE. (The DMAs are issued in parallel and, as a result, the responses may be out of order.) Each SGE is pointing to a 4-byte value in the table (e.g., an Integer) and, therefore, an entire MRd/CplD cycle is set up to transfer only 4-bytes at a time. As we show in Section 5, such small DMAs incur high overheads.

As previously mentioned, database systems typically avoid this penalty by resorting to serializing the data before transmitting it. They copy the data fragments into a contiguous area—the transmission buffer—and issue an RDMA SEND or WRITE off of that area instead. We instrument that scenario as well. The analyzer shows, once again, that the buffer contents are sent to the card in one DMA (spanning several large TLP packets) in this case. This shows that the technique is effective in increasing throughput. Unfortunately, it presents drawbacks we discuss in Section 5.2.

2.2 Projection in an OLTP System

Fragmentation also occurs in OLTP scenarios. Consider a database stored in a row-oriented format and a select/project query that returns a subset of the columns for rows in which a given predicate holds. Figure 3 depicts this scenario. From a server's point of view, sending that result over the network means assembling one message per row, consisting of the columns specified in the select list.

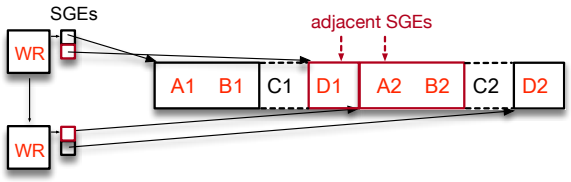

Figure 3: A typical OLTP work request for a query that returns two rows, $\{A_1, B_1, D_1\}$ and $\{A_2, B_2, D_2\}$. A projection eliminates the C column from the rows. The database assembles a chain of two WRs with two SGEs each. The analyzer shows that four DMAs were performed, following the WR's SGEs strictly, even when two consecutive SGEs are adjacent.

The instrumentation of this scenario was also telling. The DMA schedule the card used followed exactly the SGE list: a transfer of the first WR' data and then the following WR's. We note that the card could have issued a single DMA operation that would join the $\{D_1\}$ and $\{A_2, B_2\}$ chunks, since they are contiguous. It did not. The reason could have been that these SGEs belong to different WRs.

To verify that this optimization could be possible, we replaced the SGE that point to $\{A_1, B_1\}$ by two SGEs pointing to $\{A_1\}$ alone and $\{B_1\}$ alone, respectively. We did the same to the next tuple. We assumed that the card would coalesce these two SGEs into one DMA since they are adjacent and belong to the same WR. Once again, it did not. Ultimately, the card performs strictly what it is told, i.e., it issues one DMA per entry in the SGE list irrespective of optimization opportunities.

These observations indicate that data fragmentation is a natural phenomenon in OLAP and OLTP database systems. Note that each scenario presents a different kind of gap. The gaps in the OLTP case appear regularly across the rows because of the projections. They can be considered as physical gaps. The gaps in the shuffle case are due to the distribution of rows. As such, they can be seen as logical gaps. Either way, because of the apparently *imperative* DMA schedule imposed by SGEs, traditional RDMA cards miss several optimization opportunities. To unlock them, we need more flexible control structures than WR and SGEs and the freedom to tailor different DMA schedules.

3. THE D-RDMA EXTENSION

We extend RDMA with control structures that point to larger regions than a single message at a time and that contain both data and gaps. We call these structures *Non-Contiguous Regions* (NCRs). The D-RDMA extension encompasses (a) these new structures, which we claim to be compatible with the verbs API, (b) a new verb we think is missing from standard RDMA, and (c) a runtime to support them. We describe the new structures and interface in this section and go over the runtime details in Section 4.

3.1 The Strided Regions

We introduce NCRs by presenting a use-call we call *Strided Region* (SR). The rationale for this region is the following. RDMA's Work Request can be seen as a rudimentary language that can only describe *Contiguous Regions*, the SGEs. To make it more expressive, we propose an NCR type that captures whole regions that present data and gaps in regular patterns.

We call this region *strided* as it is useful when there are gaps in the area to be transmitted that occur in strides. Strides Regions in particular, and NCRs in general, are represented by data structures that replace the combination of WR and SGEs with richer data descriptions. In other words, they are intended to be used instead of WR on certain RDMA verbs, e.g., in `ibv_post_send`.

A Strided Region can be defined using a base pointer, a period made of one or more elements, the width of the elements, and a stride, as Figure 4 illustrates. The stride is described by a frequency, e.g., 1 every 2 elements, and an optional start position, if different than the base address.

Strided Regions are expressive enough to handle the OLTP example from Section 2, as Figure 5 illustrates. One SR instance suffices to locate the start of both rows in that example, because the rows correspond to the SR's periods, and to identify the gaps generated by the projection of the C column.

In general, SRs are more compact than WRs when describing the same regions. For instance, an OLTP result with adjacent rows, similar to the one we refer to here, requires $(1 + P) * R$ SGEs, where P is the number of non-

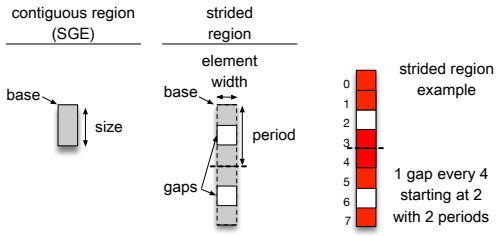

Figure 4: Contiguous Regions are insufficient to capture data patterns. Non-Contiguous Regions, such as a Strided Region, can describe data and gaps in a compact way.

adjacent projected columns and R is the number of rows in the result. In contrast, we can describe the same result *topology* with only one Strided Region. (If the rows were not adjacent, we could still compactly describe them but using a different NCR that we present in Section 3.3).

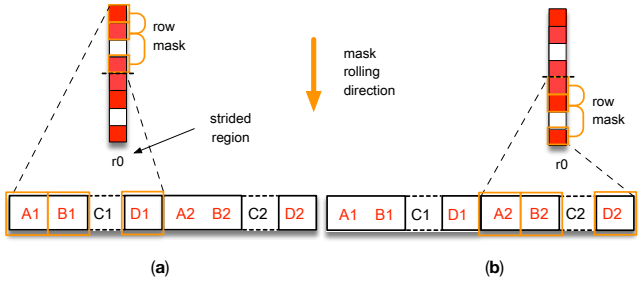

Figure 5: Example of a Strided Region, r_0, that captures an OLTP query's result. The latter is comprised of two adjacent records that had the C column projected out. The period of the Strided Region corresponds to a row in the result. (a) The first period's *row mask* generates the payload $\{A_1, B_1, D_1\}$. (b) The second period's row mask generates the payload $\{A_2, B_2, D_2\}$.

Figure 5 introduces a new concept, the *Row Mask* (RM), which has the following purpose. In RDMA, each WR is treated as a different message and, if the message is smaller than the network's MTU, it is transmitted as a single packet. In the NCRs, however, the message boundaries are derived implicitly with the help of RMs using simple rules. When using a single Strided Region to represent the data, D-RDMA assumes that the period *is* the row mask. In other words, to find the $i-th$ row/message, one should look at the $i-th$ Strided Region's period. The components of the $i-th$ message are the contiguous areas of that period, as illustrated in Figure 5. Ultimately, NCRs/RM are for D-RDMA what WR/SGEs are for RDMA.

3.2 An Additional Verb

As discussed in Section 2, another source of fragmentation is data shuffles. In general, the data in any *all-to-all* communication appears fragmented on the initiating side. The reason is that, by definition, these operations send potentially contiguous values to different servers. To exacerbate the problem, RDMA does not offer any *verb* that would send data to many servers at once, even though it has one that can receive data from anyone, i.e., the `ibv_post_srq_recv`

verb. The `srq` in the verb's name stems from *Shared Receive Queue*. This verb acts as the *gather side* of an all-to-all communication, allowing a server to receive data from any number of queue pairs (connected servers).

D-RDMA introduces a new verb that would act as the *scatter* side of the communication. In contrast to the `ibv_post_send` verb, which can only address one server at a time, the new verb can generate messages destined to a set of target servers at once. We call the verb `ibv_post_ssq_send`, where the `ssq` stems from *Shared Send Queue*.

Figure 6 shows an example of how to overlay NCRs in a shuffle. It extends the use of an NCR in Figure 5 (the OLTP example) in at least two ways. First, the data in this example is organized in columnar format. Therefore, a set of NCRs—three in this case—overlays the data, each NCR representing a single column. Second, each destination server has a different set of NCRs. Since this is a shuffle, the gaps on one server correspond to the data in another server. Figure 6 represents this with the blue and red colored NCRs.

Most importantly, the use of Strided Regions here is, once again, more compact than the WR/SGE alternative. The shuffle example in Section 2 required $R * C$ SGEs, where R is the number of rows and C is the number of columns in the table. To describe the same data with Strided Regions, we only need $N * C$ regions, where N is the number of servers. Note that $R \gg N$, since the number of rows will be much larger than the number of servers that receive the shuffled data.

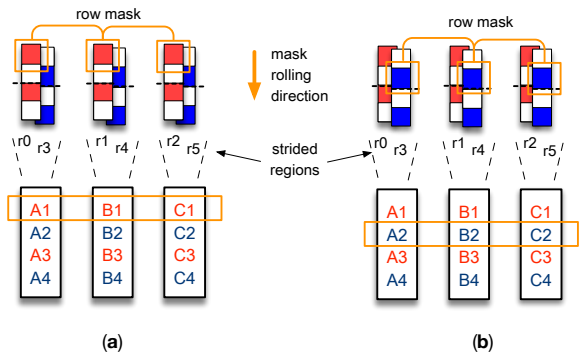

Figure 6: Example of a set of Strided Regions in a data shuffle scenario. The regions $[r_0, r_1, r_2]$ and $[r_3, r_4, r_5]$ describe the red and the blue server's data, respectively. (a) The initial row of a shuffle is determined by positioning the row mask across its NCR set. Note that the regions have the same size, and the pattern of each of them determines how to distribute data. In a shuffle, the patterns of different servers are complementary. (b) Advancing the row mask is done by moving by one position down the regions instead of advancing to the next period, as in Figure 5.

The new verb imposes one significant constraint about the NCR sets: the number of elements in each NCR must be the same irrespective of its target server. This allows the verb to set a single row mask for all the servers. The row mask assumes data is being transposed for transmission, i.e., advancing the mask moves to the next element of each NCR. (This mechanism can optionally be switched to a columnar one, in which case the row mask will advance in periods, as in the OLTP case, traversing one NCR/column at a time.)

3.3 Additional Regions

We showed how expressive the Strided Regions can be to capture both OLTP and OLAP communication patterns, but they are by no means the only region types possible. For instance, Figure 7 depicts another region type called **-Mapped Regions*. This type of region aims to capture scenarios where results are not regular or contiguous.

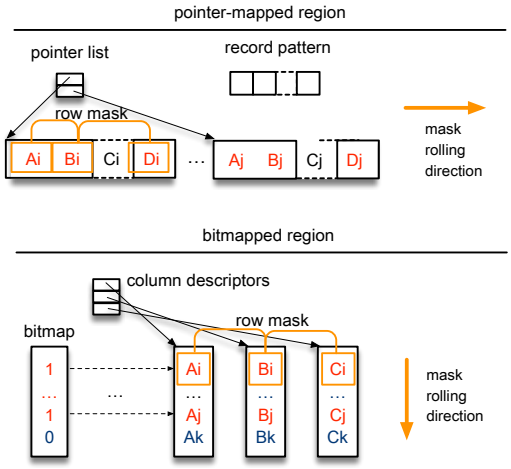

Figure 7: A new family of region types can capture scenarios where the data does not show regular patterns. Top: The Pointer-Mapped Region uses an array of pointers to the resulting rows. It assumes the data is in a row-oriented format. The row mask of a Pointer-Mapped Region follows the record pattern. Bottom: The Bitmapped Region uses a bitmap vector to represent the rows that should be transmitted. It assumes the data is in a columnar format and uses an auxiliary column descriptor array to locate the columns. The row mask of a Bitmapped Region cuts across all the columns.

The *-Mapped Regions have two variations. A Bitmapped Region assumes data is in a columnar format and uses a bitmap instead of a pattern to determine which rows should be sent to one server and a vector of column descriptors to point to the columns. A Pointer-Mapped Region assumes the data is in a row-oriented format and uses a pointer array to refer to the records and a record pattern to determine the portions of the records to be transmitted.

We assume that different systems may need a different set of regions, and since the overhead of supporting each region type is small, we plan on carrying several types in a D-RDMA NIC. We discuss how we define such a set in Section 6 but note that all the regions must have the following characteristics:

- A region type representation must be significantly more compact than the equivalent WR/SGE representation of the messages that must be transmitted.
- A region type must come with a precise semantics of row mask. The mechanics of obtaining the next packet from a region must derive naturally and unambiguously from the semantics.
- A region type must support *Gap Analysis* via algorithms that allow the runtime to locate all gaps, determine their contiguity to one another, and the ratio of gaps per data unit within a portion of the region.

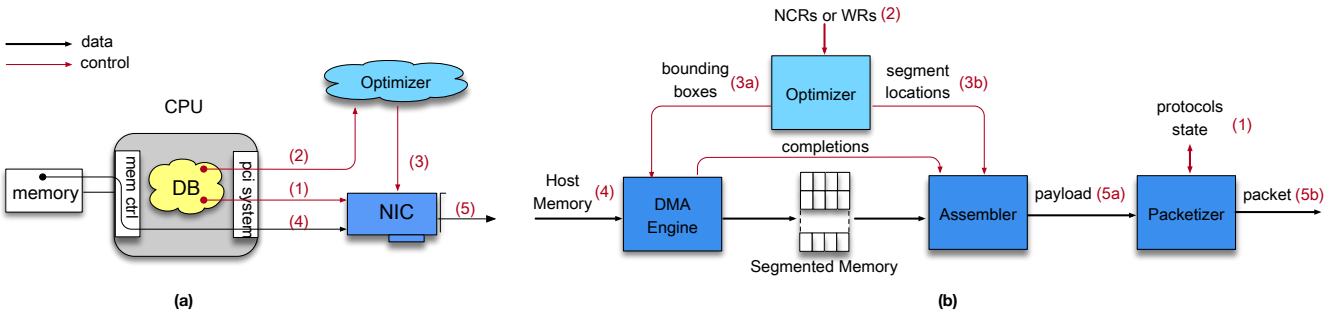

Figure 8: The life of an operation in the D-RDMA runtime from the system's perspective (a) and from the NIC's (b). The application sets up a connection as usual (1). It uses NCRs instead of SGEs to post work to the card (2). The card determines a DMA schedule upon receiving the NCR list (3,3a,3b). For more details on 3a and 3b, please refer to Figures 9 and 10, respectively. The card issues the DMAs (4). The card uses the row window for that request to find and packetize the data (5,5a,5b). We note that the optimizer can be implemented in software (e.g., at the driver level) or in hardware.

4. NIC EXTENSIONS

A D-RDMA request containing NCRs is handled by a runtime on the card. Figure 8(a) depicts the workflow from a system's point of view. First, the application sets up the connections (queue pairs) to the remote hosts as it would in an RDMA scenario. It can then use the Infiniband Verbs API to send transmission instructions to the card. Certain verbs would take Non-Contiguous Regions to describe the requests. Upon receiving an NCR-based request, the runtime in the card forwards it to an optimizer, which determines the fastest DMA schedule to bring the data from the host. The runtime then executes this DMA schedule, and, as data arrives, it assembles the payloads contained in the NCRs before forming and sending out the packets. We will comment on the optimization and packetization processes shortly.

Internally, the runtime comprises five components, shown in Figure 8(b). Two of these components are similar to those we would encounter in a regular NIC: the DMA Engine is responsible for transferring data from the host's memory into the card's; and the Packetizer envelopes payload data with headers and trailers for the network protocol the card is handling. The third component, the Segmented Memory, is also present in regular cards but it is implemented slightly differently in D-RDMA supporting cards. The remaining two components, the Optimizer and the (Payload) Assembler, are extensions required to process D-RDMA. We describe the modified and new components next.

4.1 The Optimizer

This component receives and analyzes the NCRs and determines bounding boxes on which the DMA schedule will be based. Figure 9 depicts this process by showing how alternative bounding boxes can cover a given Non-Contiguous Region. The Optimizer considers the relative speed of transferring each data chunk separately or together and decides whether to merge chunks through their gaps. In other words, a gap may create some overhead by adding bytes to the transfer or it may alleviate it by reducing the number of DMAs. We determine the cost of transferring different chunk sizes experimentally and elaborate on the results in Section 5.

The Optimizer needs to know where the gaps are before it can calculate bounding boxes. In Strided Regions, the gaps are explicitly declared; the optimizer can locate them and know their relative sizes. However, future regions types may deal with gaps differently. As we stated before, one of the requirements for a region type to work in our scheme is to allow what we call *gap analysis algorithms*. As the name implies, the region must support methods that locate contiguous areas in a set of non-contiguous regions and determine the size and distance among their gaps. For example, the bitmapped region we presented in Section 3.3 supports gap analysis via bit counting operations.

Gap analysis also applies when the NIC is dealing with a set of NCRs, as is the case for the shuffle scenario. The algorithm, in this case, needs to consider two additional features: NCR overlaps and inter-chunk space. Regarding overlaps, an NCR must be amenable to such reasoning. In the Strided Region case, it is easy to compare base addresses and the number of periods across every pair of regions to determine whether they overlap. Regarding inter-NCR space, it can be treated simply as intra-NCR gaps. This means that the NIC is not limited by the boundaries of an NCR when planning a DMA. If the NIC deems the space between one or more NCR to be relatively small, it is free to bring data from several NCRs at once[2].

4.2 The Segmented Memory

The DMA engine moves data from the host into this special area in the card. The salient feature of this area is that it is comprised of smaller, independent memory buffers, hence its name. The DMA engine writes to this area in a stripped way: the 4 initial bytes are written to a destination base address that the optimizer determined. The following 4 bytes are written to the next memory buffer—and so on. If the DMA engine writes to the last memory buffer, it can loop to the first one and continue. In other words, each buffer takes 4 bytes at a time, but the combined area can take a much larger write bandwidth.

The DMA engine writes the data of a single bounding box to the segmented buffers in parallel. Moreover, the engine can mediate the transfer of several bounding boxes simul-

[2]Some restrictions need to be considered here. The NCRs must be under the same Protection Domain and Memory Region for the NIC to perform these calculations.

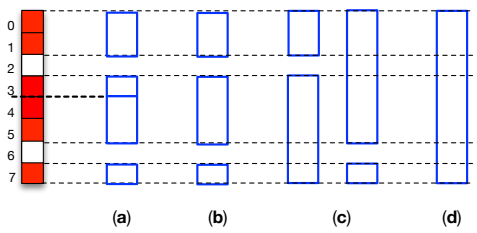

Figure 9: Different sets of bounding boxes to transfer the two-period strided region from Figure 5. (a) The configuration that regular RDMA would use following the WR/SGE in that figure. (b) The areas [3] and [4,5] could be merged because the card can transfer adjacent regions. (c) The card may decide to transfer either the gaps at [2] or at [6], generating larger bounding boxes. (d) The card may transfer both gaps using a single bounding box spanning [0,7].

taneously[3]. This and the segmentation technique allow to achieve very high ingress bandwidths. For example, when running the segmented area at 322 MHz, which is common in 100 Gbps interfaces, it suffices to write 8 segments of 4 bytes per cycle to reach line speed. We expect the NIC to have a ×16 Gen 3 PCIe connection or an ×8 Gen 4 one. Such PCIe bandwidth is typical among 100 Gbps NICs. In practice, we have many more than 8 segments for reasons we discuss next.

4.3 The (Payload) Assembler

This component connects to the read side of the segmented area. Note, however, that not all data written to that area will be transmitted to the network because D-RDMA allows the non-contiguous regions to contain gaps. Therefore, we need to allow more ingress data than egress: the area allows at least 8 segments to be read in parallel, but many more can be written. For this reason, the Assembler is built internally with a rectangular cross-bar, e.g., 16×8, where the ingress side connects to the segmented area, and the egress one connects to a payload buffer.

The Assembler receives information about the incoming packets and their destination addresses from the Optimizer. Once the DMA engine completes one of these transfers, the Assembler also gets notified. It then uses the row mask information sent by the Optimizer to program the crossbar at every cycle. Figure 10 depicts this interaction.

5. PRELIMINARY EXPERIMENTS

We quantify the potential benefits of D-RDMA by running three sets of experiments. The first set evaluates the baseline DMA performance of the shuffle and the OLTP scenarios from Section 2 (§ 5.1 and § 5.2). The second set characterizes the efficiency levels for DMAs depending on their relative size (§ 5.3, § 5.4, and § 5.5). The third set evaluates how a D-RDMA card's internal memory needs to be structured to produce RDMA packets at line speed (§ 5.6). Lastly, we discuss the design constraints that the experiments uncovered (§ 5.7).

Hardware and Platform. We performed all our experiments on a server with a Intel Xeon Silver 4216 (2.10GHz) CPU.

[3]We experiment with 64 simultaneous transfers but can increase it to up to 256.

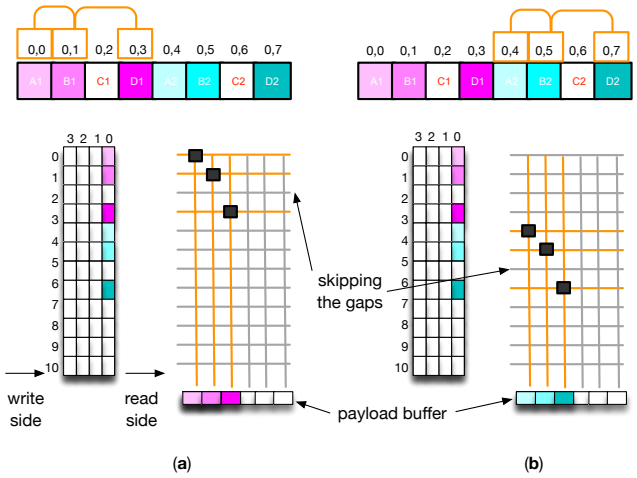

Figure 10: Top: the NCR from Figure 5 with color codes corresponding to the assigned locations on the segmented area is written to the Segmented Area. Bottom: (a) The initial row mask is used to program the crossbar, which produces the payload without the gap in position 2, and (b) the following row mask is used similarly to produce the second payload. We assumed that the DMA schedule chosen was that of Figure 9(d), i.e., two periods at once. Note that the figure uses dimensions for the Segmented Area and for the crossbar that facilitate visualization. The actual dimensions are different.

The experiments involving RDMA were carried out with a 100 Gbps Mellanox ConnectX-5 card. The DMA and packetization experiments used a Xilinx Alveo U50 FPGA with a 100 Gbps port and a ×16 PCIe connection.

5.1 Zero-Copy Experiment

In this experiment, we evaluate the efficiency of a shuffle operation when using traditional RDMA calls. We deploy a table in a columnar format containing five integer attributes (4 bytes each) and 64K rows. We varied the table size but obtained consistent results for tables from that size and up. The shuffle operation targets two remote servers (cf. Section 2). For simplicity, we confine the experiment to a single core, but the results would extend to a multi-threaded scenario just as well. To focus on the cost of the RDMA calls, we precalculate all the WRs and group them in batches of 1024 in a single RDMA WRITE request. We send batches as fast as the card would take them. The resulting performance is shown on Figure 11 under "Zero-Copy."

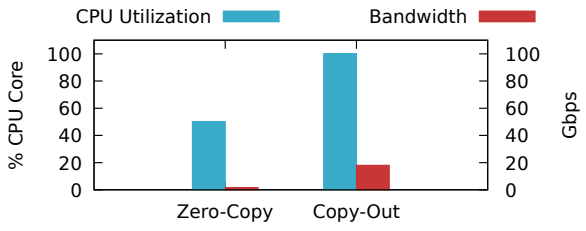

Figure 11: CPU utilization and bandwidth when using zero-copy (left) or with a transmission buffer (right) for the OLAP operation described in Section 2.

The shuffle operation spends 50% of a CPU's time issuing `ibv_post_send` calls and only achieves 1.5 Gbps of throughput. As expected, small DMA transfers over the PCIe system incur substantial overhead, resulting in a decrease in throughput [20]. We also run the OLTP projection scenario from Section 2, once again, using traditional RDMA calls. From an RDMA perspective, the shuffle and OLTP cases are very similar—chains of small data chunks—, and so are the performance results, which we omit for brevity.

5.2 Copy-Out Experiment

In this experiment, we evaluate the benefits of copying data into a 4 MB buffer before transmission. We use the same table setup and RDMA configuration from the Zero-Copy experiment (§ 5.1). The results appear on Figure 11 under "Copy-Out."

The shuffle operation uses 100% of a CPU core this time, twice as above, because the data copying exerts a toll. However, it achieves 18 Gbps of throughput. The results indicate that this method requires at least 5 CPU cores to reach 100 Gbps (line speed). Once again, we run the equivalent OLTP projection from Section 2 using the copy-out technique with the same results. The throughput improvement notwithstanding, the problem with this approach is scalability. As discussed above, network speeds are increasing faster than CPU speeds. If five cores are needed for 100 Gbps traffic, twenty would be required for 400 Gbps, and forty for 800 Gbps. D-RDMA avoids such overheads by creating larger DMA transfers, as we discuss next.

5.3 Impact of Transfer Sizes

In this experiment, we programmed an FPGA-based NIC to initiate DMA operations and evaluate the latter's efficiency under varying transfer sizes. We used a repurposed version of Corundum [11], a high-performance network card logic that supports several FPGA-based platforms. The card uses 64 PCIe tags, i.e., there are 64 transfers in-flight at any given time. The reads are sequential and target a large 1GB memory region. The rationale is to force the reads to be served by main memory by not reading any address twice. Figure 12 shows the results in terms of latency and throughput obtained.

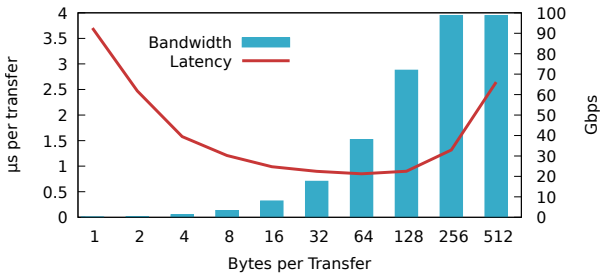

Figure 12: Latency and bandwidth of DMA reading increasingly large chunks of data from a network card.

Surprisingly, we can observe that the latency of very small reads can be higher than that of bigger ones. We thought initially the reason would lie in DRAM's nature; it is not designed to support low latency access to small ranges [25]. We explore this phenomenon in more detail in the next section.

In turn, the throughput results were as expected. The larger the transfer, the better the throughput. To reach the peak throughput, a DMA operation needs to move at least 256 bytes, which is also the maximum payload size of the PCIe link. The additional packets are the reason why the latency starts growing back as the transfer sizes increase.

5.4 Impact of Alignment

We rerun the same experiment as in Section 5.3, but this time we transfer regions increasingly farther apart from one another. For instance, when transferring 1 byte at a time, we experiment skipping 1 byte between transfers (contiguous transfers), then 2 bytes, then 4, and so on. The experiment's goal is to assess the performance impact of repeated accesses to the same DRAM regions. Note that we do not include numbers for strides that are smaller than the read size, as this would lead to overlapping reads. Figure 13 shows the results of this experiment.

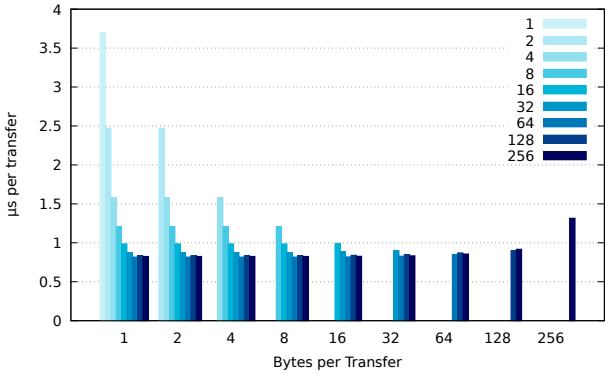

Figure 13: Latency of DMA reading increasingly large chunks of data with different strides.

The most telling result occurs on smaller transfers. We observe that when we skip less than 16 bytes between transfers, the latency times remain high. Once the stride size hits 64 bytes, the cache line size, the latency stays constant for reads below 256 bytes. The results when using small strides explain the increased latency we have seen in Section 5.3.

5.5 Impact of Caching

In the previous experiments, we forced the card to fetch data from main memory[4]. We did so by reading from a 1 GB host memory area and never hitting the same address twice within the same experiment run. In this experiment, we evaluate how caching can affect transfer performance.

We reduce the host memory area to 16 KB, forcing different transfers to access the same addresses repeatedly and therefore using caching. We investigate four types of access: "Cached Sequential" uses sequential reads, i.e., a stride equal to the read size, and the 16 KB area; "Cache Aligned" uses a stride such that two sequential reads do not access the same cache line and the 16 KB area; "Memory sequential" uses sequential reads and the 1 GB area; and "Memory Aligned" uses large strides and the 1 GB area. Figure 14 shows the results of the experiment.

[4]We used the *PCIRdCur* performance counter to confirm that almost all reads from the device result in LLC misses.

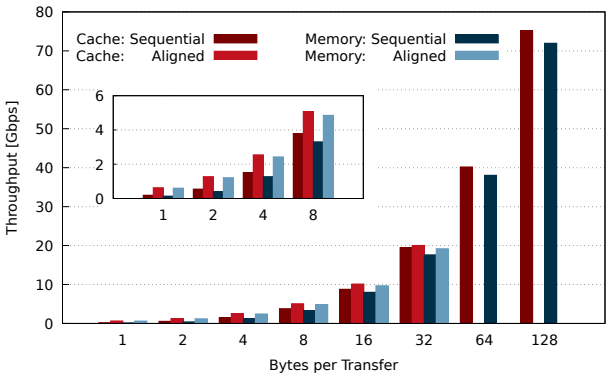

Figure 14: Throughput of the DMA engine reading increasingly large chunks of data from either the LLC (red) or from main memory(blue). We plot both sequential reads and cache line aligned reads, omitting the latter once the size becomes a multiple of the cache line (64B).

Much to our surprise, the effect of caching in the transfer was very small. For aligned reads, the difference is always below 5%. The biggest differences in transfer times are, once again, for very small reads. For instance, for 1-byte reads, the throughput increases from 137 Mbps to 190 Mbps, or 39%. However, the caching benefit decreases rapidly as the transfer size increases.

5.6 Packetization Microbenchmark

In this experiment, we determine the minimal number of segments the Assembler has to read per cycle to reach line speed. We implement segments of 4 bytes width in our FPGA-based NIC and run the logic at 322MHz, a typical rate for 100 Gbps cards. We assemble RoCE v2 packets with a payload size of 128 bytes, resulting in 198-byte packets after the protocols' overhead. Figure 15 shows the results of the experiment.

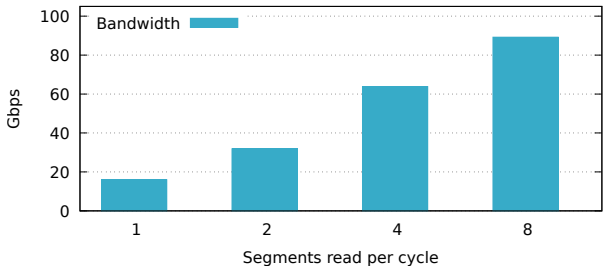

Figure 15: Networking throughput with an increasing number of parallel segments when producing RDMA packets.

As expected, the throughput obtained increases as we increase the number of parallel segments. In this configuration, reading from 8 segments suffices to reach line speed.

5.7 Discussion

The experiments in this section can help us determine the conditions necessary for D-RDMA to achieve line speed. These conditions are (a) it should be able to transfer close to 100 Gbps of data into the card and (b) it should be able to ship the same rate out the wire.

Regarding the transfer, we observed that, for latency purposes, it might be worth transferring gaps as large as a cache line if that would help connect two small regions. We also observed that small reads are very costly, even when properly aligned. For throughput proposes, the considerations are different. Transferring 256-byte chunks, or 64 integer-wide values at once, yields a throughput of 98 Gbps. However, we are giving up part of the throughput when transferring gaps. For instance, in the OLTP scenario, we can theoretically achieve 3/4 of 98 Gbps, or 74 Gbps, because only 1 in 4 columns is a gap. The benefit is that what we give up in throughput comes back in terms of freeing CPU cycles that would otherwise be used in the "copy out" approach. These considerations should be transformed into constraints that the Optimizer tries to satisfy.

Regarding shipping data at line rate, the crux of the problem is whether the machinery can push a certain number of bytes per cycle through the runtime in a pipelined fashion. The critical component is the Segmented Area. We determined that 8 4-byte segments is the absolute minimum number. The machinery around the Segmented Area and the Assembler must be carefully designed not to introduce bubbles in this pipeline.

In summary, our experiments, albeit preliminary, show that it is possible to design a card that can calculate and take advantage of DMA optimizations. The performance benefits that such a card could bring are sizeable.

6. RESEARCH AGENDA

We believe that D-RDMA can become the cornerstone of low-overhead, high-speed network communication for database systems. To reach its full potential, several fundamental research directions should be explored, including:

Non-Contiguous Regions. We introduced the Strided Regions and showed that they are particularly suited to the regular data patterns present in our motivation examples. We also briefly mentioned how *-Mapped Regions can capture more irregular data placement scenarios. These are hardly the only NCRs that the D-RDMA language can offer. We are looking into additional region types to support features such as variable-length, compressed data, and even data structures that require pointer chasing logic for tree-like traversals. Ultimately, we seek an expressive set of NCRs that can navigate the most common transmission scenarios used in database systems.

We also mentioned that NCRs can replace WR/SGEs in selected Infiniband verbs. The verbs API is implemented by an open-source library, `libibverbs`, that applications link to when using RDMA. We are considering the different alternatives to integrate NCRs into this context. One possibility is to extend `libibverbs` itself and introduce the new data structures.

Shared Send Queues. Our motivation with this new queue is to allow the card to process several NCRs at once that are destined to different servers. This creates a powerful optimization opportunity for larger bounding boxes to encompass multiple—overlapping or not—NCRs. However, creating a new type of queue has profound implications. The behavior of a new queue type should be considered in light of the different transport types, i.e., RC, UC, UD, and even the more esoteric XRC. In RDMA, queue types are hardware objects, making them challenging to change once specified.

The alternative to creating a queue type is to create a stand-alone verb that supports group communications. From a syntax point of view, it would be an unusual verb because, instead of taking a "compound" queue pair parameter as the shared send queue, the verb would be given a list of "individual" queue pairs. Ultimately, choosing between a hardware queue + verb construct versus a verb alone depends on what benefits the hardware implementation can bring. This requires further investigation.

DMA Optimization Algorithm. A brute-force algorithm to find the best bounding box for an NCR (or a set thereof) could be as follows. One would find the power-set among all the NCRs' data chunks and establish the cost of each possible combination to find the best one. We are currently working on pruning techniques that would make this approach feasible and are also considering a dynamic programming-based solution.

Regardless of the exact approach considered, we can use faster, more straightforward algorithms that can be easily implemented in hardware or more sophisticated ones that can be executed in software, e.g., at the driver level. The software-implemented algorithms may incur more latency, whereas the hardware ones may not be optimal. We are investigating the different tradeoffs in this context.

The Segmented Area. This area is of utmost importance for the performance of D-RDMA. It has to allow for fast parallel writes to adjacent segments. It also has to allow for parallel reads from random segments, which is why we deploy a crossbar on the read side. The dimensions of this area—numbers of segments, and depth and width of each segment—are still an object of study. We determined their minimal sizes here but are still considering what the optimal dimensions should be.

We plan on implementing the segmented area on an FPGA. Modern FPGA chips provide several types of SRAM configurations, such as Ultra-RAM, Block-RAM, and LUT-RAM. These variations present different options in terms of size and width of memory. The choice of depth and number of segments should consider these options.

Payload Assembly. This component requires other elements than just the crossbar and the payload buffer. It must also store state information of every ongoing and planned DMA operation and access it efficiently when the DMA engine notifies it of a DMA completion. Furthermore, this component should support the arithmetic operations that rolling a row mask entails. These operations are simple, most likely additions to jump to the next period on an NCR, but may require multiple arithmetic units to work in parallel, one for each attribute on the row mask. The design of this component should incorporate all these details.

7. RELATED WORK

The issue of high CPU consumption and data copying associated with fast networking has been documented before [15, 16, 20, 28]. Our effort, however, uses a protocol analyzer, equipment capable of precisely observing the behavior of the network card directly, to reach this conclusion. To the best of our knowledge, this is the first work to provide analysis at this level of detail.

There have been other attempts to improve RDMA efficiency in database communications. We divide such work into two categories. First, we consider efforts that rely on software optimizations. Second, we consider approaches that rely on hardware modifications.

Software-based approaches. We can further subdivide the software approaches by the layer at which they deploy their solution. Starting from the layer closer to the networking API, some pieces of work noted that the de/serialization of data before transmission prevents zero-copy transfers [21]. The authors suggest having the applications directly use *scatter-gather* units that exist in RDMA cards. The resulting performance is better but the card still operates at a fraction of its full bandwidth. Moving further up on the stack, some suggest shielding the applications from low-level interfaces such as RDMA, offering higher level abstractions instead [3, 10, 17, 26], or designing communicating subsystems specifically for RDMA [18, 22], or even giving some database query operators, such as distributed join and aggregation, the ability to optimize for fast network transmissions [5, 6, 23, 29]. We see our approach as orthogonal to these, as they could be re-implemented to leverage our RDMA extension and attenuate the demand for CPU and memory bandwidth in fast communications.

Hardware-based approaches. There are several approaches that, similarly to ours, propose shifting some tasks to smart network cards. Some of them design very specific hardware, for instance, to accelerate de/serialization [27]. These cards have very limited functionality, which is contrary to the generic approach we propose. Our work is more aligned with generic smart NIC works in that D-RDMA can be implemented on top of them. There are two broad classes of smart NICs for our purpose: those running application logic on local cores, called network processing units (NPUs), e.g., iPipe [19], sPin [8] or cards that do so via extensible hardware, e.g., Corundum [11] or StRoM [24]. We believe the hardware-based approach is more suited to support D-RDMA since NPU-based cards often present high latencies.

8. CONCLUSION

This paper introduced D-RDMA, an extension to RDMA that allows a database to benefit from zero-copy data transfers. The key conceptual differences between RDMA and D-RDMA are that (a) D-RDMA allows the application to describe data transmissions in a more compact way, and (b) it shifts the responsibility of devising a DMA schedule from the host to the network card.

As a result, a D-RDMA card can transfer data much closer to line speed, even fragmented data in database scenarios. In the process, a D-RDMA card gives CPUs cores back to the database that would otherwise be used to interact with the network. While realizing our vision's potential will take more research work, we are excited by the prospects that D-RDMA opens for the next generation of database systems to communicate at increasing network speeds.

Acknowledgments

We would like to thank the reviewers for their insightful comments and suggestions. This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement 683253/GraphInt) and from the Department of Energy through grant ARPA-E DE-AR000084.

REFERENCES

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreaos, and S. Madden. The design and implementation of modern column-oriented database systems. 2013.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[3] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. Dpi: the data processing interface for modern networks. *CIDR 2019 Online Proceedings*, page 11, 2019.

[4] Arista. 400g solutions. https://www.arista.com/en/products/400g-solutions.

[5] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1463–1475, New York, NY, USA, 2015. Association for Computing Machinery.

[6] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, Mar. 2016.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[8] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beránek, L. Benini, and T. Hoefler. A risc-v in-network accelerator for flexible high-performance low-power packet processing. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 958–971. IEEE Press, 2021.

[9] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems, 2017.

[10] P. Fent, A. v. Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-latency communication for fast dbms using rdma and shared memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1477–1488, 2020.

[11] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.

[12] Infiniband architecture specification–annex a16: Roce. https://cw.infinibandta.org/document/dl/8567.

[13] Infiniband architecture specification. https://www.infinibandta.org/ibta-specifications-download/.

[14] M. Jackson, R. Budruk, J. Winkles, and D. Anderson. *PCI Express Technology 3.0*. Mindshare press, 2012.

[15] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 437–450, USA, 2016. USENIX Association.

[16] A. Kesavan, R. Ricci, and R. Stutsman. To copy or not to copy: Making in-memory databases fast on modern NICs. In *Data Management on New Hardware*, pages 79–94, Cham, 2017. Springer International Publishing.

[17] F. Liu, C. Barthels, S. Blanas, H. Kimura, and G. Swart. Beyond mpi: New communication interfaces for database systems and data-intensive applications. *SIGMOD Rec.*, 49(4):12–17, Mar. 2021.

[18] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 48–63, New York, NY, USA, 2017. Association for Computing Machinery.

[19] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[20] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[21] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang. Breakfast of champions: Towards zero-copy serialization with nic scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 199–205, New York, NY, USA, 2021. Association for Computing Machinery.

[22] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4):228–239, Dec. 2015.

[23] A. Salama, C. Binnig, T. Kraska, A. Scherp, and T. Ziegler. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.*, 40(1):27–37, 2017.

[24] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[25] S. Srikanth, L. Subramanian, S. Subramoney, T. M. Conte, and H. Wang. Tackling memory access latency through dram row management. In *Proceedings of the International Symposium on Memory Systems*, pages 137–147, 2018.

[26] L. Thostrup, J. Skrzypczak, M. Jasny, T. Ziegler, and C. Binnig. *DFI: The Data Flow Interface for High-Speed Networks*, page 1825–1837. Association for Computing Machinery, New York, NY, USA, 2021.

[27] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

[28] T. Ziegler, V. Leis, and C. Binnig. Rdma communciation patterns. *Datenbank-Spektrum*, 20(3):199–210, 2020.

[29] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 741–758, New York, NY, USA, 2019. Association for Computing Machinery.