Farview: Disaggregated Memory with Operator Off-loading for Database Engines

Dario Korolija
dario.korolija@inf.ethz.ch
ETH Zurich
Switzerland

Dimitrios Koutsoukos
dkoutsou@inf.ethz.ch
ETH Zurich
Switzerland

Kimberly Keeton*
kimberlykeeton@acm.org
Hewlett Packard Labs
USA

Konstantin Taranov
konstantin.taranov@inf.ethz.ch
ETH Zurich
Switzerland

Dejan Milojičić
dejan.milojicic@hpe.com
Hewlett Packard Labs
USA

Gustavo Alonso
alonso@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

Cloud deployments disaggregate storage from compute, providing more flexibility to both the storage and compute layers. In this paper, we explore disaggregation by taking it one step further and applying it to memory (DRAM). Disaggregated memory uses network attached DRAM as a way to decouple memory from CPU. In the context of databases, such a design offers significant advantages in terms of making a larger memory capacity available as a central pool to a collection of smaller processing nodes. To explore these possibilities, we have implemented *Farview*, a disaggregated memory solution for databases, operating as a remote buffer cache with operator offloading capabilities. Farview is implemented as an FPGA-based smart NIC making DRAM available as a disaggregated, network attached memory pool capable of performing data processing at line rate over data streams to/from disaggregated memory. Farview supports query offloading using operators such as selection, projection, aggregation, regular expression matching and encryption. In this paper we focus on analytical queries and demonstrate the viability of the idea through an extensive experimental evaluation of Farview under different workloads. Farview is competitive with a local buffer cache solution for all the workloads and outperforms it in a number of cases, proving that a smart disaggregated memory can be a viable alternative for databases deployed in cloud environments.

1 INTRODUCTION

Historically, databases have invested significant efforts to reduce I/O overheads. Initially, memory was very limited in capacity and disks were slow. Over time the bottleneck shifted as faster storage became available (SSDs, Non-Volatile Memory (NVM)), memories became larger, and multicore emerged. Yet, the I/O overhead remains a major factor in the overall performance. To minimize it, databases have relied on keeping more and more data in memory, a trend that cannot continue for two main reasons: databases induce considerable data movement, which is known to be highly inefficient in modern computing; and the amount of data to be processed keeps growing while DRAM capacity does not.

Optimized query plans typically push down selection and projection operators to filter out the base tables as early as possible. However, filtering base tables to get the data actually needed by the query is an expensive step. Base tables are fetched from storage as blocks that are placed in a buffer cache in memory. From there, a query thread reads the data and filters it to form the input to the rest of the query plan. Often, most of the data is dropped because it does not match the query's selection predicate. As more data is involved, the overhead becomes larger. Data movement has been identified as one of the biggest inefficiencies in computing [24, 25], making the way databases operate intrinsically problematic from a systems perspective, even if main memory could grow indefinitely.

However, DRAM capacity is also a major limitation, because the size of data processed by analytical databases keeps growing [47]. The current approach to tackle such a limitation is to use NVM, introduced as an alternative that is both cheaper and has higher capacity than DRAM (and persists data), but has larger latency. In databases, it is increasingly used to improve and expand the memory hierarchy [10–12, 57, 69, 70]. Such designs do not address the overhead of moving large data sets to the CPU, only to have most of it filtered or projected out. Specialized hardware between memory and the CPU has even been proposed to filter data as early as possible, minimizing bus congestion and cache pollution [4, 30].

An alternative approach for addressing memory pressure is to exploit the distributed nature of database engines, particularly in the cloud, to take advantage of non-local memory. In such distributed settings, the coupling of storage, compute, and memory capacity is problematic both cost-wise and performance-wise: the inability to independently provision each of those elements leads to inefficiencies due to over-provisioning. For instance, allocating large amounts of memory for tasks that are not compute-heavy leaves CPU unused, as other applications might not be able to run on the remaining memory. Conversely, allocating many virtual CPUs to a task may result in the memory being underutilized for lack of compute capacity for other tasks. Due to these challenges, essentially all cloud architectures follow a clear trend towards disaggregation. Currently, the most visible form of disaggregation is the separation of compute and storage. The next step is the disaggregation of memory and compute, which is being pursued in various forms: disaggregated DRAM [34, 48, 49],

*Now at Google

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

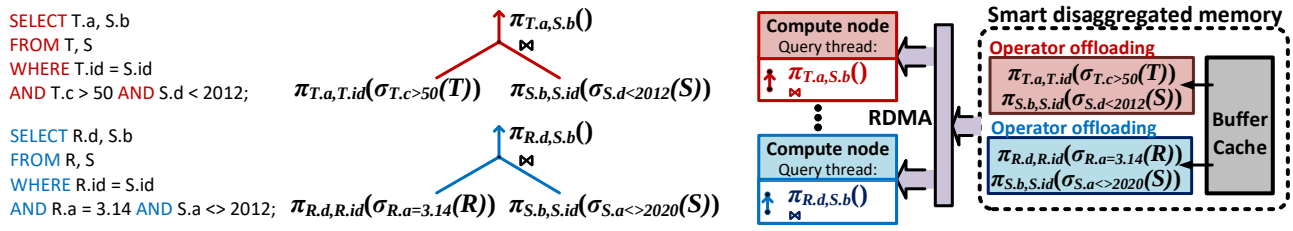

Figure 1: Farview query execution: Offloading of query operators to the smart disaggregated memory and splitting the query plan between compute and memory nodes

disaggregated persistent memory [52, 68], far memory [3, 7] and smart remote memory [45, 67].

In this paper we demonstrate that databases are uniquely positioned to exploit disaggregated memory to address both the issues of inefficient data movement and DRAM capacity. Our approach is based on physically detaching query processing from memory buffer management. The buffer pool is placed on network attached disaggregated memory, with query processing nodes provisioned on demand to run a query by reading the data from the network attached buffer pool. This design presents multiple advantages. Consider, for example, queries with low selectivity (e.g., TPC-H Q6) or an aggregation after a `GROUP BY` statement. In the first case, the query reads a large amount of data from the buffer pool just to discard most of it. In the second case, a query of the form `SELECT T.a, COUNT(*) FROM T GROUP BY T.a` will usually return only a handful of tuples, but it still requires reading the entire table. The smart disaggregated memory we propose offers the opportunity to (1) reduce data movement by pushing down operators to the disaggregated memory, so that the processing nodes receive only the relevant data; and (2) reduce memory requirements for computing nodes by centralizing the buffer cache in disaggregated memory and removing unnecessary copying of the data to the compute nodes. Figure 1 shows an example where projection and selection of two concurrent queries are offloaded to smart disaggregated memory, while the join and the final projection happen at the compute nodes.

To prove that these ideas can work in practice, we have developed *Farview* (FV), a novel platform for data processing over disaggregated memory. Farview supports near-data processing to compensate for the added latency of accessing memory through the network by moving data reduction operators (selection, projection, aggregation, etc.) to the disaggregated memory. Farview is based on a smart NIC built on top of Coyote, an open source FPGA shell [46] that enables the FPGA to support dynamic operator push down on concurrent streams reading from memory. The smart NIC supports RoCE v2 at 100 Gbps using an open source RDMA stack [67], optimizing the interaction between network and memory as well as minimizing the network processing overheads on the computing node CPU, thereby freeing processing capacity. From a database perspective, Farview acts as a disaggregated memory buffer pool with operator push down capability that is byte addressable by the threads running queries at the computing nodes.

For reasons of space, in this paper we focus on the design, architecture, and experimental evaluation of Farview when running queries, leaving other aspects such as cache replacement policies and query processing elasticity to future work. Farview currently supports a wide range of query operators: selection, projection, aggregation, distinct, group by, regular expression matching, and encryption. All these operators achieve near line-rate speed, adding insignificant latency to baseline network overheads. Farview also supports concurrent access, with multiple clients all accessing the same disaggregated memory. Our experiments show that Farview induces almost no overhead over operating on local memory and provides significant performance gains when data can be reduced in the disaggregated memory.

2 BACKGROUND AND RELATED WORK

In this section we motivate Farview and discuss related work. Farview is based on extensive experience in data center, computer, and processor design [19, 31]. For reasons of space, we focus here only on two salient aspects: memory disaggregation and near-data processing.

2.1 Coping with memory pressure

Data growth has turned DRAM into a major bottleneck [29, 47]. To cope with this bottleneck, advances in memory technologies and networking are leveraged to increase effective DRAM capacity. Within a local node, studies have explored compressing cold pages into local DRAM [47] or using local NVM directly as memory or with DRAM acting as a transparent caching layer [28, 59, 62]. These designs have also been used in databases in different ways, to expand virtual memory [57], directly as memory [10], or as a cache [69]. While in many cases there are performance advantages, these efforts require significant redesign in the database engine and do not address the underlying problem of inefficient data movement.

In a distributed setting, the notion that memory can be shared across a cluster of machines has been around for decades [8]. More recently, the advent of fast networks like InfiniBand FDR/EDR has renewed interest in exploiting memory (DRAM or NVM) accessed over the network. *Remote memory*, a distributed memory infrastructure where a group of comparably equipped compute nodes make their memory available to their peers, has been exposed to applications as a remote paging device [7, 37], as a file system [1], and as distributed shared memory [27, 54, 66]. Although this organization leverages existing resources and can improve resource utilization of otherwise unused memory, it entangles compute and memory for provisioning and expands the failure domain and attack surfaces of each machine [2, 47].

In contrast, *disaggregated memory* systems use network attached memory that is distinct from the memory in the compute nodes [49, 68]. This approach allows the disaggregated memory to scale independently of the system's computing or storage capacity [48], and removes the need to over-provision one resource to scale another. From the database perspective, this is a promising architecture. An evaluation of existing database engines (MonetDB and PostgresSQL) using LegOS [65], an operating system for disaggregated memory, indicates that the network overhead is the main bottleneck [75, 76]. The authors conclude that disaggregated memory has potential, but significant performance loss occurs due to the use of sub-optimal algorithms and lack of suitable data structures.

In Farview, we demonstrate that disaggregated memory is especially suitable to database engines when used as a buffer pool (also suggested in [75]). This makes the integration of disaggregated memory a more natural way to address the memory capacity limitation as neither the interface to memory needs to be changed nor the memory hierarchy expanded. What remains to be addressed are the data movement inefficiencies and network overheads.

2.2 Efficient data movement

Data movement inefficiencies can be addressed by using near-data processing. Expanding on decades-old work that memory and storage can be active components [43, 58, 60, 61], several approaches to memory disaggregation explore increasing the intelligence of network-attached memory. *Far memory* [3] proposes simple hardware extensions to reduce the number of network traversals to access non-local memory, and support for efficient notifications to facilitate consistency of data cached in the local memory of the nodes. In the context of databases, the advantages of processing data in the disaggregated memory have also been suggested [75, 76], but without proposing a possible implementation. The argument in favor of such designs is simple: push down selection and projection operators (as well as potentially other operators such as grouping, aggregation or even joins where one of the tables is small) to the memory or storage so that the base table is filtered out in-situ and irrelevant data does not need to be moved or sent. Although to our knowledge not yet used with disaggregated memory, the idea mirrors a growing trend to push SQL operators near the data, until now mostly to storage [41, 71]. Even more ambitious are accelerators embedded in the data path between memory and CPU caches [22, 30], which can filter data as it is read from memory to reduce data movement and cache pollution. Finally, in the cloud, systems like Amazon's AQUA [13] use SSDs attached to FPGAs to implement a caching layer for RedShift that supports SQL filtering operations and operator push-down to minimize the amount of data movement from storage to the processing nodes. These designs are based on introducing a *bump-in-the-wire* processor to be able to process data closer to where it initially resides, instead of moving it first and then processing it. In Farview we adopt a similar design, with operators placed directly in the path of the network and memory. Farview requires neither changes to the storage layer interface nor specialized processors, whilst adding a layer of dynamicity provided by the reconfigurable platforms. Moreover, we focus on disaggregating the buffer pool in DRAM, rather than introducing additional caching layers between storage and compute.

The network overheads can be addressed using advances in networking (in addition to compensating for it by using near-data processing). Most of the work on different forms of disaggregated memory utilizes low-latency RDMA instead of TCP/IP, often extending one-sided operations on RDMA to offload group-based operations for storage replication (e.g., HyperLoop [45]), concurrency and transactions for data structures (e.g., AsymNVM [52]), and memory access operations for key-value stores (e.g., StRoM [67]). RDMA employs the network protocol (InfiniBand [38], RoCE [39]) and the Network Interface Card (NIC) to move data directly between the memory of different machines. At the speed at which networks operate today, RDMA can be used to efficiently transfer large amounts of data across machines at the rates of DRAM memory channels [33]. It is thus especially suitable for disaggregated memory and databases [5]. It has been shown to speed up distributed operators such as data shuffling [50], joins [14, 15], transactional workloads [16], and indexing [77]. In Farview, we use RDMA to efficiently transfer data through the network so that the query processing thread directly gets the data from the remote buffer pool. As suggested by current architectural trends in the cloud, Farview is implemented on top of an FPGA-based smart NIC. It supports SQL operators acting on the RDMA data streams as they move along the data path connecting the disaggregated memory to the network. The design efficiently combines near-data processing with faster network transfers, while removing the need for a conventional CPU to support the disaggregated memory (a design that resembles that of AQUA, which also uses FPGAs instead of conventional CPUs, and that is aligned with how FPGAs are deployed in Microsoft's Azure [20, 32, 63]).

3 FARVIEW: SYSTEM OVERVIEW

Farview is a smart disaggregated memory attached to the smart NIC with operator offloading capabilities that behaves as a database buffer pool. Traditionally, query processing threads access base tables by reading them from the buffer pool and copying the data to their private working space. With Farview, nothing changes for the query thread, except that the read operation is on a remote disaggregated memory rather than local memory, potentially with a subset of the operators already applied.

3.1 Smart buffer pool with operator offloading

Farview exposes a data API to the buffer pool (Section 4.2) that can offload operators to the disaggregated memory. Farview executes an *operator pipeline* with one or more operators (e.g., a selection and then an aggregation) to process the data as it is read from disaggregated memory, effectively functioning as a bump-in-the-wire stream processor (Section 5). As done in conventional query processing, operator pipelines are constructed from individual blocks that implement a given operator and provide standard interfaces to combine them into pipelines. The modular nature of these pipelines and the reconfigurability of the FPGAs allows the list of operators to be swapped and easily extended in the future (e.g., join operators).

Farview currently supports a range of operators, including: (1) *projection* operators to reduce the columns returned (and potentially reduce memory accesses) (2) *selection* operators that filter data according to a collection of predicates; (3) *grouping* operators that

combine tuples (e.g., distinct, group by and aggregation); and (4) *system support operators* that process data in-situ before sending the data (e.g., encryption/decryption) and perform system optimization tasks like packing the data to reduce the overall network usage.

3.2 FPGA-based architecture

Prototyping smart disaggregated memory requires several components, including DRAM, memory controllers, a network stack, a mechanism to support concurrent access to the memory, and stream processing capacity for operator push-down. Modern FPGAs are a natural match for such functionality, as high performance and flexibility can all be combined in a single device rather than having to connect separate components such as processor, NIC, and memory, all inducing significant data movement overheads. FPGAs can also support substantial amounts of local memory, directly attached to the FPGA. This on-board memory is usually organized in multiple channels (even High Bandwidth Memory (HBM)) [6]. Farview's design (Section 4) leverages these characteristics to implement disaggregated memory as a lean component.

To deploy operators that can process data on the disaggregated memory, the FPGA is divided into multiple isolated *virtual dynamic regions* that operate concurrently. These dynamic regions can be obtained by different clients and can process different query requests. Each dynamic region serves an access request to the disaggregated memory and can implement a separate operator pipeline that can execute a set of different queries. These regions are dynamically reconfigurable: the logic deployed in them can be swapped at runtime without having to reconfigure the whole FPGA. This swap takes on the order of milliseconds, depending on the size of the regions [46].

A combination of operators is precompiled as an operator pipeline into a hardware design that is dynamically loaded into the FPGA at runtime, upon a request from a client (i.e., a thread processing a query at a computing node). The operators and their pipelines can be modified or extended, and new ones can easily be added by combining the existing ones or changing their parameters.

4 FARVIEW: IMPLEMENTATION

Farview is implemented on top of Coyote, our open source FPGA shell [46]. The shell provides a layer of abstraction hiding services like RDMA network stack and memory virtualization from concurrent system users behind high level interfaces.

4.1 Architecture

As shown in Figure 2, Farview is organized around three main modules: the network stack, the memory stack, and the operator stack. The network stack (Section 4.3) manages all external connections and RDMA requests, providing fair share mechanisms across all concurrent accesses. The memory stack (Section 4.4) implements the buffer pool, and can be used as regular memory, with blocks/pages being loaded from storage as needed. The memory stack houses the memory management unit (MMU), which handles all address translations to the on-board memory attached to the FPGA and provides the necessary arbitration and isolation between concurrent accesses. The operator stack (Subsection 4.5) contains the dynamic logic necessary to push down operators to the disaggregated memory. It lies between the memory and the network stacks and can

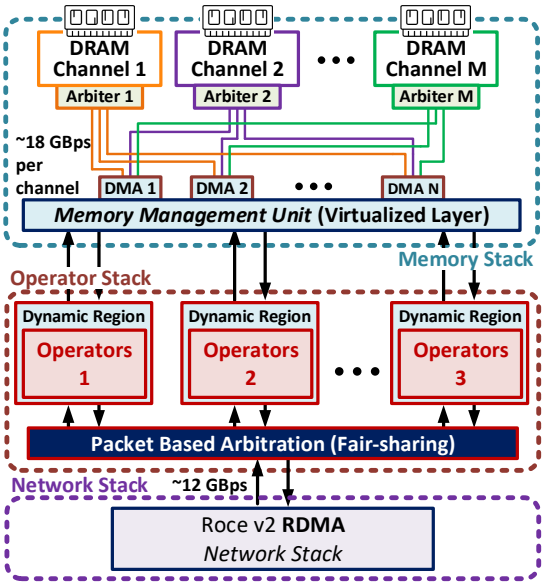

Figure 2: High level view of Farview's architecture

be seen as a specialized stream processor acting on the data as it moves from the memory stack to the network stack. It also controls how data is retrieved from the memory stack. The operator stack is reconfigurable, and its operator logic can be changed at any point without affecting the operation of the rest of the system.

Clients access the disaggregated memory by opening a connection with Farview to one of the dynamic regions, each able to contain one of the possible operator pipelines. Whenever a client makes a request to Farview, the network stack routes the request to the correct virtual dynamic region in the operator stack belonging to the client that initiated the request. The read request is forwarded to the memory stack, which translates the virtual address to a physical address in the on-board FPGA memory, and then issues the actual data request. The clients have the local catalog information that is used to determine the addresses of the tables to be accessed. The returned data is streamed back to the dynamic region, where the loaded operators are applied. Finally, the resulting data is forwarded to the network stack, and further sent directly to the memory of the client.

The interfaces of the data and the requests are all based on a simple *AXI stream* handshaking protocol [26], which provides uncomplicated synchronization, pipelining, and backpressure mechanisms, all allowing Farview nodes to support processing at high throughput. The standard interfaces help with portability across different boards. The dynamic region where the operators are loaded always exposes the same set of interfaces to the operators, thereby simplifying the task of creating the operators. This protocol also permits deep pipelining of the overall design, which allows processing to occur simultaneously in different areas of the system.

To attain high frequencies and reduce the impact of the physical distance between the stacks, data is buffered in queues as it traverses from one stack to the other. The queues, as well as any temporary state created by Farview operators, are implemented using

fast on-chip FPGA memory. The buffering allows clear decoupling of processing stages, which helps with structuring the overall system and allows Farview nodes to achieve the operating frequencies necessary to sustain processing at line speeds. The frequencies of the components in Farview range between 250 MHz (network stack, operator stack) and 300 MHz (memory stack).

Compared to existing FPGA frameworks (which support arbitrary functionality), in Farview the dynamic regions must be connected to the network and memory stacks, which have fixed locations within the FPGA, thus reducing the degrees of freedom in placement and sizing. Farview's management infrastructure must cope with network speeds of 100 Gbps (and even higher internal speeds), which require wide buses (at least 512 bit) [64, 67], further restricting region placement and sizing. We choose predefined dynamic regions to accommodate these placement and sizing restrictions. In practice this implies that the size of each virtual dynamic region is fixed and cannot be changed. However, each region is more than large enough for the purposes of offloading the operators we intend to support.

4.2 Farview programmatic interface

Farview exposes a simple high level data API covering both the critical path operations and connection management operations. The former utilize the high performance 100G fabric, whilst the latter are handled via regular TCP/IP connections. The critical path operations include both standard low level one-sided RDMA read and write commands to read (write) data from (to) memory and an extra Farview command, which invokes the operator(s) in the Farview node directly over the read data stream. We use the Farview command as the basis for more complex SQL expressions.

A client running on a remote computing node begins by establishing a connection to Farview. In response, it gets a created object representing the connection (*QPair*), which holds all the necessary information for the connection and is used as an argument to subsequent Farview methods. The following function is used for this purpose:

```
bool openConnection(QPair *qp, FView *node);
```

The client can then at any point request that a different operator pipeline be loaded. Operator pipelines are precompiled and kept in the operator pipeline library on the target Farview node. The operation is executed with the following command, where the *opid* is a unique id of the requested pipeline:

```
bool loadPipeline(QPair *qp, int32_t opid);
```

Farview memory is virtualized, and internal management is handled by the MMU in the Farview node. Since we are focusing on read-only scenarios, Farview does not currently provide concurrency control. Clients allocate memory for tables using the following allocation functions:

```
bool allocTableMem(QPair* qp, FTable *ft);
void freeTableMem(QPair *qp, FTable *ft);
```

Regular RDMA requests for simple reading/writing of the remote table can be sent with the following two functions:

```
void tableRead(QPair* qp, FTable *ft);
void tableWrite(QPair* qp, FTable *ft);
```

Farview's request corresponds to a specialized InfiniBand verb [38] that invokes the remote processing capabilities with an arbitrary parameter set, specific to each operator pipeline. This verb is invoked in the following generic function, which is used as the basis for building additional higher level functions supporting specific operator combinations and queries:

```
void farView(QPair* qp, FTable *ft, uint64_t
    ↪  *params);
```

As an example higher level function, we present a selection operator with real number predicates:

```
void select(QPair* qp, FTable *ft, uint64_t
    ↪ *projection_flags, uint64_t *
    ↪ selection_flags, float predicate);
```

This function can be used for the following type of queries:

```
SELECT S.a FROM S WHERE S.c > 3.14;
```

In this case, the `projection_flags` variable signals column `a` while `selection_flags` signals column `c`. The predicate is passed as a value. More complex variations are possible: for instance, if the hardware operator supports it, the predicate operand could also be a variable.

The interface presented here is intended to be used by the query compiler in Farview, rather than directly by the client. The development of the query compiler is left as future work.

4.3 Network stack

Farview's network stack implements a reliable RDMA connection protocol, building on an existing open source stack [67] that implements regular one-sided RDMA read and write verbs. We extend the original stack with support for out-of-order execution at the granularity of single network packets. The out-of-order execution, along with credit-based flow control and packet based processing, allows Farview to provide fair sharing, an important feature in a system shared by multiple separate clients concurrently. Crucially, it prevents any malevolent behaviour by any of the users that could lead to a complete system stall.

Similar to other remote memory systems based on RDMA (e.g., [52]), we add a Farview verb (based on InfiniBand SEND) to control the operators. Requests sent using this verb are directly written into receive queues in the FPGA fabric instead of into the memory. Because of this, the operators can react to these requests directly, incurring no additional memory latency overheads. This verb can contain an arbitrary number of additional operator-specific parameters to indicate to the disaggregated memory how to access and process the data. The network stack manages connections and keeps the necessary state, while remaining highly customizable for further extensions.

In RDMA, the information describing a single node-to-node connection or RDMA flow is associated with a *queue pair*; every network packet sent is associated with a queue pair. Each connection flow contains a set of unique identifiers, which are used to differentiate the flows and to provide network and hardware isolation between flows. The isolation extends to the MMU, where it provides the necessary protection of the memory regions associated with each queue pair. Queue pairs also keep track of the memory buffers in

Farview nodes where the tables reside, and the client nodes where results are written. As part of establishing the connection flow, the client sends the virtual address of its local buffer to the Farview node. Upon completion of query requests, Farview will load results into this client buffer using one-sided RDMA operations. The buffer in the Farview node can also be shared between different queue pairs, enabling multiple requests to share intermediate results. The current prototype focuses on read-only operators and has no transactional memory support.

Upon connection establishment, each network connection flow and its corresponding queue pair are associated with one of the available virtual dynamic regions and in turn with one data stream from the Farview memory stack. This data stream is used to read the queried data into the dynamic regions, process it, and send it over the network. The flow is ready to process data once the operator pipeline corresponding to the flow is present in the dynamic region.

4.4 Memory stack

The memory stack implements the buffer pool memory using the on-board DRAM memory attached to the FPGA. It handles dynamic memory allocations, address translations, and concurrent accesses.

The central part of this stack is the MMU, which is responsible for all memory address translations to a shared dynamically allocated memory. It propagates and routes all memory requests and subsequent data. It supports the issuing of multiple outstanding requests and has fully decoupled read and write channels. It provides parallel interfaces, isolation and protection for the requests stemming from different dynamic regions with a set of arbitrators, crossbars, and dedicated credit-based queues. Farview's MMU supports naturally aligned 2 MB pages, which greatly reduces the coverage problem of smaller pages. The MMU contains a translation lookaside buffer (TLB) implemented on Block RAM (BRAM), the fast on-chip FPGA memory. Farview's TLB holds all virtual-to-physical address mappings for the dynamic regions.

The on-board DRAM memory is organized into multiple channels. The "softcore" memory controllers for these channels are instantiated in the fabric of the FPGA. Each memory channel can provide a certain amount of memory bandwidth. Our prototype uses the Xilinx Alveo u250 [72], which has up to four separate memory channels. For the tests in this paper we utilized up to two channels, each with its own softcore controller that runs at 300 MHz. The width of the interface to the memory channel controller is 64 bytes. This implies a maximum theoretical bandwidth of 18 GBps per channel (Figure 2). The bandwidth matches the bandwidth usually found on more conventional systems with general purpose CPUs [17].

The multiple channel organization of on-board FPGA memory offers additional parallelization potential. Farview's MMU provides an interleaved abstraction for DRAM accesses that aggregates the bandwidth from multiple memory channels. It does this by allocating memory in a striped pattern across all available memory channels [46], thus maximizing the available bandwidth to each dynamic region. The higher bandwidth available to each dynamic region also enables a vectorized processing model (Section 5.3).

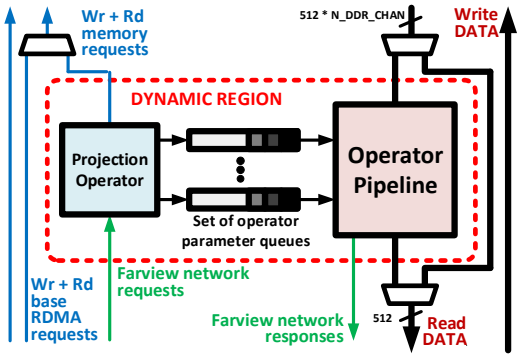

Figure 3: Single dynamic region in the operator stack

4.5 Operator stack

The operator stack is where the operator pipelines attached to connection flows to/from memory are deployed. The stack is implemented as a collection of predefined dynamic regions. The operators deployed in the dynamic regions use the interfaces exposed by Farview's network and memory stacks.

Operator pipeline logic can be deployed and swapped on-the-fly without affecting the integrity and operation of the system or other operator pipelines belonging to other clients. These regions and their access to memory are isolated from each other (see Section 4.4). Such functionality is typically not available in commercial systems, but very often studied in the literature, e.g., [44, 51, 74].

Figure 3 illustrates how a single dynamic region processes a query request. The base RDMA read and write requests forward the virtual address and transfer length parameters directly to the memory stack and to the MMU (the blue path in Figure 3), bypassing the dynamic region. If the request is a simple RDMA read/write request, it contains no additional parameters. If the request is a Farview command, it carries a number of operator-specific parameters (green path), along with information about the virtual memory locations it is accessing. The number of parameters can vary depending on the specific operators that are present in the operator pipeline. The write path allows RDMA updates to the memory. The operators' bump-in-the-wire data processing occurs along the read data path. The width of the data path's input to the dynamic region scales with the number of available memory channels. As a result, with the aforementioned striping technique, each dynamic region gets the full bandwidth potential of the disaggregated memory and its multiple memory channels. The output is forwarded to the network stack using a 64-byte datapath width, the same as the provided network interface.

Query responses are sent via the response channel (green path), which initiates one-sided RDMA transfers to the client. The operator pipeline dynamically generates these responses for each result packet only when the packet is ready to be sent to the client node. This approach enables the operators to dynamically control the size of the result data transfers, which is important for operators (e.g., filtering) where the size of the result data is unknown when the request arrives, prior to processing. The direct data streams between the memory

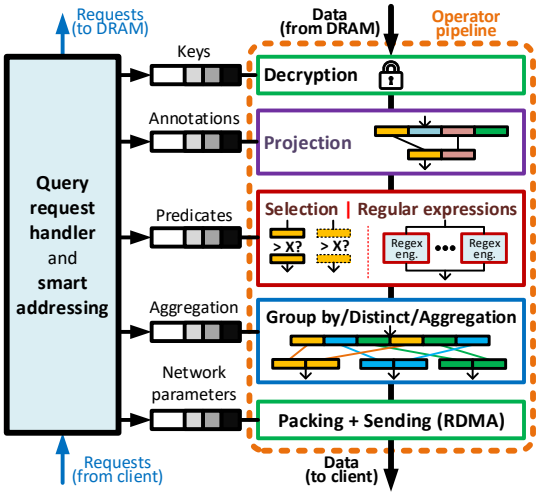

Figure 4: Operator pipeline example

controller, the operator, and the network are scaled so as to saturate the bandwidth in each module and to provide optimal performance.

5 FARVIEW: OPERATORS AND PIPELINES

In this section we discuss operator pipelines and four classes of operators: projection, selection, grouping and system support.

5.1 Operator pipelines

As described above, a query is transformed into an operator pipeline, which is deployed on a dynamic region allocated to the corresponding client. An operator pipeline contains one or more operators that provide partial query processing on datapath operations to disaggregated memory. This processing is effectively a bump-in-the-wire that operates on data without introducing significant overheads.

Figure 4 illustrates a generic operator pipeline that includes a broad set of operator classes, including projection, selection (e.g., predicate selection, regular expression matching), grouping (e.g., distinct, group by, and aggregation), and system support (e.g., encryption/decryption).[1] These example operators are described in more detail in the remainder of this section. Which operators are actually present in the pipeline depends on the requested set of queries to be executed. In one scenario, the pipeline can support projection, followed by selection and group by. In another, it can support regular expression matching on encrypted strings, which requires decryption early in the pipeline. The reconfigurable nature of the regions provides flexibility, as it allows arbitrary operator types and combinations to be natively supported by the system.

When a query request arrives, it is first forwarded to the query request handler, which requests the data from the memory stack via smart addressing. At the same time, any necessary parameters for additional processing are forwarded to the remaining operators in the pipeline. Data arriving from the memory is processed in a streaming fashion by these operators. Once the processing is done, the resulting data is sent back to the client over the network. Each pipeline has the potential to be fed with input data each FPGA clock cycle. In query terms, this translates to each pipeline being fed with up to a single tuple in each cycle. In the same manner, a pipeline has the potential to produce results on the output of every cycle. Using this design, operator processing overhead can be efficiently hidden behind the memory and network operations.

Operators are written by the Farview developer as part of the smart disaggregated memory system design, using common hardware description languages like VHDL or Verilog or in the C++-like syntax supported by high-level synthesis tools such as Vivado HLS. The operator pipeline is precompiled, so that it can be deployed to a dynamic region at runtime. Operator implementations use Farview's network, memory and operator stack interfaces, rather than the interfaces of the underlying FPGA board, which makes the operators portable across Farview deployments on different platforms.

5.2 Projection operators

Projection: A common operation in databases is projection, which returns a subset of a table's columns. Consider for example a projection of the form `SELECT S.car, S.price FROM S`, where `S.car` and `S.price` are non-consecutive attributes and have a number of fixed-length attributes between them. The projection operator reads the table from the disaggregated memory, parses the incoming data stream based on query parameters describing the tuples and their size, and projects the requested tuples into the pipeline for further processing using annotation. During parsing, the tuples are annotated with parameters from the requested query, and obtained from the parameter queues. These parameters are simple flags that state which of the columns are part of the selection, projection or grouping phases. Their interpretation depends on the actual combination of operators being used and their specific implementations.

Smart addressing: In scenarios where queries request only a small subset of the columns from a very wide table, performance would benefit from reading only the requested columns from memory, rather than reading full rows and applying the projection on the incoming data stream. For this purpose, we implement a smart addressing optimization that issues multiple, more specific, data requests to memory. Smart addressing is most effective when the total number of columns per tuple is large and the number of projected columns is much smaller than the total; otherwise, it is more efficient to read entire tuples and project using annotations, as described above, since the memory access is sequential. We explore the crossover point between these two modes in Section 6.3.

5.3 Selection operators

Selection operations that filter data directly map to the SQL `WHERE` clause (e.g., in queries of the form `SELECT * FROM T WHERE T.a > 50`). These operators have the ability to greatly reduce the amount of data to be processed by later stages and ultimately the overall amount of data sent through the network, thus reducing the overall network bandwidth usage. For example in TPC-H Q6, only 2% of the data is finally selected. Pushing the filtering to the disaggregated memory reduces the I/O overhead by orders of magnitude.

[1] We assume that all data is stored in row format, but there is nothing intrinsic preventing support for column data.

Farview's selection operators consist of predicate selection, regular expression matching and vectorized selection operators.

Predicate selection: For selection, the value of an attribute is compared against a constant provided in the query. In FPGAs, such a comparison can be implemented in different ways. We choose to hardwire the selection predicate as an actual matching circuit instead of creating a truth table as done in [71], as Farview has the ability to dynamically exchange the operators. This approach uses fewer resources and, at the same time, supports a variety of different possible predicates. It also permits complex predicates defined over different tuple columns, which can be split into multiple pipelined cycles. The supplied annotations from the request determine which of the columns in the tuple are evaluated during the predicate matching phase.

Regular expression matching: String matching is becoming an increasingly important operator in SQL (e.g., using either `LIKE` predicates as in TPC-H Q16 or regular expression matching). It is even more important in unstructured data types, such as in the case of JSON fields in PostgreSQL. In Farview we have integrated an open source regular expression library for FPGAs [40] and use it to filter strings. In these operators, data is retrieved from the remote node only when it matches the given regular expression. The operator implements regular expression matching using multiple parallel engines, instantiated in the operator stack. The parallelization allows the module to fully sustain processing at line rate. Unlike software solutions, the performance of the operator is dominated by the length of the string and does not depend on the complexity of the regular expression used [40].

Vectorization: Farview implements a limited form of vectorization as an optimization to improve the performance of stateless operators like selection. To alleviate the inefficiencies of the tuple-at-a-time query processing model [36] and its `next()` function calls that pass tuples from one operator to the next, the database community has adopted query compilation [56] and column stores. Column stores either process data in full batches like MonetDB [18], or in smaller vectors like VectorWise [78]. The latter allows the use of tight loops and/or SIMD instructions to process column data, allowing DBMSs to take advantage of the latest CPU advances for data processing. In Farview, we use a vectorized model similar to that of VectorWise, but with a vector size that is chosen based on the degree of memory striping (described above), rather than trying to fit the size of the processor's L1 cache. With vectorization, data is read in parallel from multiple memory channels, and individual tuples are emitted to a set of selection operators executing in parallel. The number of parallel operators is chosen based on the number of memory channels and the tuple width. This approach achieves both higher read bandwidth from the memory stack (due to memory striping) and higher processing throughput (due to the parallel operators). At the moment the operator pipeline with vectorization is enabled for the simpler queries and workloads that can be easily parallelized without data dependencies (e.g., selection).

5.4 Grouping operators

Distinct: The distinct operator eliminates repeated column entries before they are sent over the network. It directly maps to the SQL `DISTINCT` clause, in queries such as: `SELECT DISTINCT T.a, T.b FROM T`. It operates by hashing the values and preserving the entries in the hash tables present in the fast FPGA on-chip memory. As the complete hashing is calculated in the FPGA, the distinct operation can be done on multiple columns without noticeable performance overhead, but using more FPGA resources.

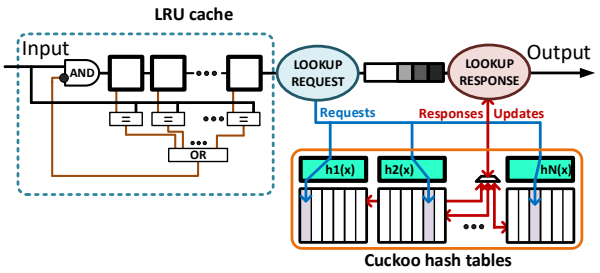

Figure 5: Architecture of the DISTINCT operator

To sustain the line rate without negatively affecting the overall pipeline processing, the distinct operator needs to be fully pipelined in order to overcome the latency of the lookups and updates of the hash table. This pipelining creates potential data hazards, in the case where two successive tuples with the same value will be inserted into the hash table and ultimately sent over the network as distinct elements. Because of the latency of the hash table, the second (following) tuple cannot see the update produced by the first one. To approach this problem we apply the strategy explained in [71] by implementing an LRU cache to hide the hash table latency. The main difference is the far higher line rate that we have to sustain in our system (over 40 times greater), yielding additional design constraints.

To guarantee full pipelining and constant lookup times, the hash table that we implement does not handle collisions. Instead, collisions are written into a buffer, which is sent to the client to be deduplicated in software. To greatly reduce the collision likelihood, we implement cuckoo hashing, with several hash tables that can be looked up in parallel. Upon eviction from one of the tables, the evicted entry is inserted into the next hash table with a different function. This occurs in the background and does not affect the full pipelining of the operator.

To successfully hide the latency of the hash table, we implement a cache to hold the most recent keys. The cache needs to be a true Least Recently Used (LRU) cache in order to guarantee protection from possible data hazards. The standard implementations of LRU caches come with a lot of overhead, as pointers and extra history-keeping data structures need to be present. For this reason, we implement the cache with a shift register, which adds a negligible latency to the data streams (the amount depends on the number of cuckoo hash tables), but is able to efficiently provide a quick lookup. The nature of the shift register provides a true LRU replacement policy and this solution thus fully satisfies the strict requirements imposed by Farview. The design of the distinct operator is shown in Figure 5.

Group by: In many applications, data is read and grouped to perform some form of aggregation (e.g., TPC-H Q1). Operations like these directly map to the SQL `GROUP BY` clause (e.g., queries such as `SELECT T.b, COUNT(*) FROM T GROUP BY T.b`).

Farview provides a group by operator with a structure similar to the distinct operator, with analogous challenges and design choices. The same cuckoo hash tables are used to preserve the groups. The implemented cache in this case is write-through, as it is no longer sufficient to just discard the data prior to sending it. The operator reads the complete table and all of its tuples without sending anything over the network, to perform the full aggregation. At the same time, it inserts the distinct entries into a separate queue. Once the aggregation has completed, the queue is used to lookup and flush the entries from the hash table along with any of the requested aggregation results to the network.

Aggregation: Aggregation operators can easily be supported in FPGAs as standalone, where simple computations are performed directly on the passing data streams, or on top of the group by operator, where each entry in the hash table contains an additional aggregation result. Farview supports a range of standard aggregation operators like `count`, `min`, `max`, `sum` and `average`.

5.5 System support operators

Encryption/decryption: A key concern for both remote memory and smart disaggregated memory is the need for data encryption [47]. Farview implements encryption so that the data is treated similarly to Microsoft's Cypherbase [9], where a database stores only encrypted data, but can still answer queries over such data by using an FPGA as a trusted module. We have implemented encryption as an operator using 128-bit AES in counter mode. Since the AES module is fully parallelized and pipelined, it can operate at full network bandwidth. This means that no throughput penalty is paid when this operator is applied on the stream, incurring only a negligible overhead in latency. This allows the data to be stored securely. Because no real processing penalty is incurred, encryption/decryption can be placed at both the input and the output of the operator pipeline. Similarly one could provide additional system support operators such as compression, decompression, etc.

Packing: At the end of the processing pipeline, the annotated columns are first packed based on their annotation flags in a bid to reduce the overall data sent over the network. Multiple columns across the tuples are packed into 64 byte words prior to being written into the output queue. This packing uses an overflow buffer to efficiently sustain the line rate. In case of the vectorized processing model, the tuples are first combined from each of the parallel pipelines with a simple round-robin arbiter.

Sending: The sender unit is the final step before the results are emitted to the network stack. It monitors the queue where the packed results are written, and, based on the queue's status, issues specific RDMA packet commands needed to produce correct packet header information in the network stack. The sender module's dynamic approach to handling RDMA commands allows us to create RDMA commands even when the final data size is not known a priori, as is the case with most of the operators.

6 EVALUATION

In this section we evaluate Farview's performance, and compare it with alternatives using a local buffer cache or a remote buffer cache. We first describe our experimental setup, including the hardware implementation details of our platforms. We then measure baseline RDMA performance using microbenchmarks, individual query performance using various operators, and query performance with multiple clients.

Table 1: Resource consumption of Farview

Configuration	CLB LUTs	Regs	BRAM tiles	DSPs
6 regions	24%	23%	29%	0%
Operators (per dynamic region)	**CLB LUTs**	**Regs**	**BRAM tiles**	**DSPs**
Projection/Selection/Aggregation	<1%	<1%	0%	0%
Regular expression	2.3%	<1%	0%	0%
Distinct/Group by	2.1%	1.3%	8%	0%
En(de)cryption	3.6%	<1%	0%	0%
Packing/Sending	<1%	<1%	0%	0%

6.1 Experimental setup

We compare Farview's smart disaggregated buffer pool (**FV**) with two different baselines: a buffer cache implemented in local (client) memory, where the processing is done on the local CPU (**LCPU**), and a remote buffer cache implemented on the memory of a different machine and reachable through a commercial NIC via two-sided RDMA operations (**RCPU**). This latter configuration resembles what is being done today for storage, where part of the processing is moved to a CPU located in the storage server. It also matches the definition of remote memory proposed in the literature. For RDMA microbenchmark experiments, we compare remote reads from Farview (**FV**) to remote reads to a different machine using one-sided RDMA operations over a commercial NIC (**RNIC**) that accesses the remote memory over PCIe. Finally, in the selection tests we also use a version of Farview with vectorization (**FV-V**).

We have implemented Farview on a Xilinx Alveo u250 board [72]. The board is part of the XACC cluster [73], which contains 10 different Alveo boards connected by a switch. The board has up to 4 DRAM channels (16GB each) connected directly to the FPGA. Each channel has a softcore (running as reconfigurable logic on the FPGA) DRAM controller with a maximum theoretical bandwidth of 18GB/s. In our tests we used two of the four available channels to reduce compilation times. The tests can be done with four channels as well, which might yield some additional performance in specific cases. The board has two (QSFP) 100Gbps network ports. We use six dynamic regions in our experiments; Farview has been tested with up to ten regions, the empirical limit for our device. Similarly to DRAM channels, we choose six regions to limit compilation times and routing complexity.

The CPU baselines contain Intel Xeon Gold CPUs: **LCPU** uses a Xeon 6248 (clocked at 3.0–3.7 GHz), and **RCPU** uses a Xeon 6154 (clocked at 2.5–3.9 GHz). For the remote buffer cache and one-sided RDMA microbenchmark baselines (**RCPU** and **RNIC**, respectively), we used a commercial Mellanox 100G card (ConnectX-5 VPI) [53]. For the CPU-based baselines we used all available compiler and code optimizations.

As shown in Table 1, Farview requires only modest FPGA resources (less than 30% of the total on-chip resources) to implement the operator stack, the network stack, and the memory stack. The majority of the utilized on-chip memory is attributed to the memory management unit and the state keeping structures of the operator

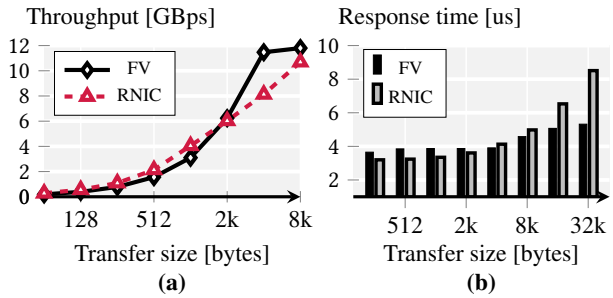

Figure 6: RDMA throughput and response time

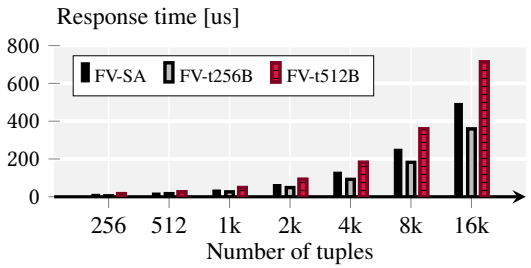

Figure 7: Standard projection vs. smart addressing

and network stack. Most of the implemented operators are not compute heavy and do not consume many resources, making it easier to combine them.

6.2 RDMA throughput and response time

For each experiment, we measure the running time until the final results are written to the memory of the client machine for both Farview and the baselines. This makes the performance numbers comparable, as the initial and end states are the same. We evaluate performance for a range of the operators and supported queries. Unless otherwise mentioned, our base tables consist of 8 attributes, where each attribute is 8 bytes long. The results for each experiment are averaged over multiple runs. As the CPU configurations may experience interference (e.g., due to context switches, interrupts), the **LCPU** and **RCPU** results are averaged over 10000 runs. Because the FPGA circuits provide more deterministic behavior, the Farview (**FV**) experiments are averaged over 1000 runs.

To characterize the system, we measure the network throughput and response time of RDMA reads from Farview (**FV**). For reference and to establish a baseline we also provide the results for reads of remote memory accessed with a commercial RDMA NIC (**RNIC**).

Figure 6(a) shows the median throughput for RDMA read operations. In Farview, a single dynamic region is present. To obtain valid read measurements, we measure the network Round Trip Time (RTT) and average it over 1000 runs. The transfer size represents the total data sent over the network for a single request. We vary this parameter until we saturate the network, using a 1kB packet size. The results show the that when we use RDMA, the network bandwidth is under-utilized for small requests. Below 4 kB, where the saturation takes place, **RNIC** achieves better throughput. This happens because **RNIC** uses a specialized circuitry running at a higher clock rate, which provides better performance for small packets. Reading from local on-board FPGA memory peaks at 12 GBps, less than the 18GBps memory channel bandwidth. This indicates that the network is the main bottleneck. In the **RNIC** case, where memory is accessed over PCIe, throughput peaks at ~11 GBps because it is bound by the PCIe bus bandwidth. This result shows the limitation of PCIe-attached memory as proposed in remote memory approaches, regardless of whether it is DRAM or NVM.

In Figure 6(b), we present the median response time for an RDMA read operation. The setup is the same as for the throughput measurements. The results show the response times of accessing remote memory over PCIe in comparison to accessing remote on-board FPGA DRAM memory. The difference during reads is ~1 us, consistent with PCIe latencies [55]. The reduction in response time provided by Farview is substantial (at least 20%). **RNIC** offers lower response times for smaller transfer sizes, but for higher transfer sizes the multi-packet processing and page handling in the FPGA network stack performs better. The network round trip latency dominates the overall time. Above 8 kB, the amount of data causes a substantial increase in response times.

6.3 Projection

To investigate how to maximize the efficiency of accessing DRAM, we compare the standard projection operator, where the whole table is fetched from memory and projection is done in the first pipeline stage, versus the smart addressing operator, which issues individual memory requests only for the target projected columns. In this experiment, we project three contiguous 8-byte columns from a larger row. Figure 7 illustrates execution times for the smart addressing operator on a 512-byte tuple (**FV-SA**) and the standard projection operator with tuple sizes of 256 bytes and 512 bytes (**FV-t256B** and **FV-t512B**, respectively). For smaller tuples (**FV-t256B**), it is more beneficial to read the whole table sequentially from the DRAM memory and handle the projection in the operator pipeline. Once the tuples become larger (512 bytes), it is better to use smart addressing to read only the columns that are required by the query (**FV-SA**). In the following experiments, which use 64B tuples, smart addressing is not used.

6.4 Selection

As mentioned, selection in Farview can have more than one predicate, even on multiple columns. The overhead of the selection on the overall processing is negligible and is fully hidden behind memory operations. For the evaluation, we run the following query:

```
SELECT * FROM S WHERE S.a < X AND S.b < Y;
```

We compare Farview with the aforementioned baseline systems. The 64-byte tuples are equal to the width of the pipeline, which leads to the highest pipeline throughput (i.e., a tuple is forwarded each cycle). We vary the selectivity of the query and present our results in Figure 8.

As we observe, in all cases (**FV, FV-V**) Farview outperforms both **LCPU** and **RCPU**. **LCPU** incurs high overheads, as it must read the input data from DRAM (not from the processor cache) and then write the results back to DRAM. The **RCPU** baseline additionally has to transfer the data through the network, leading to consistently

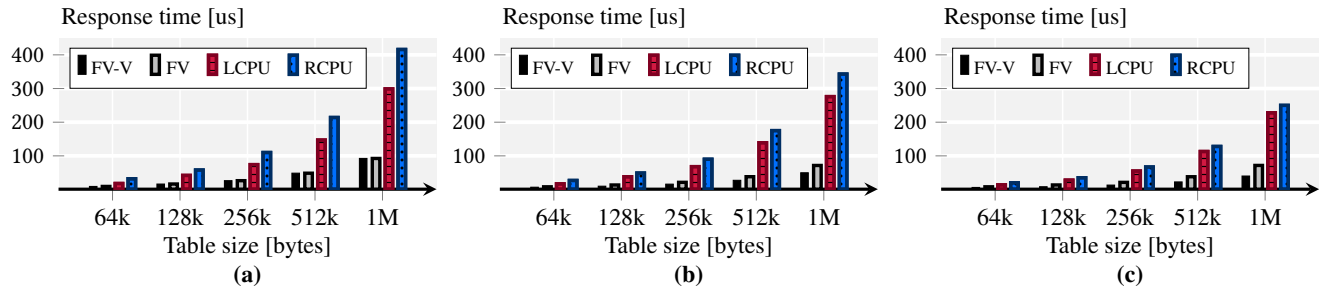

Figure 8: Response times for selection queries with (a) 100% selectivity, (b) 50% selectivity, and (c) 25% selectivity

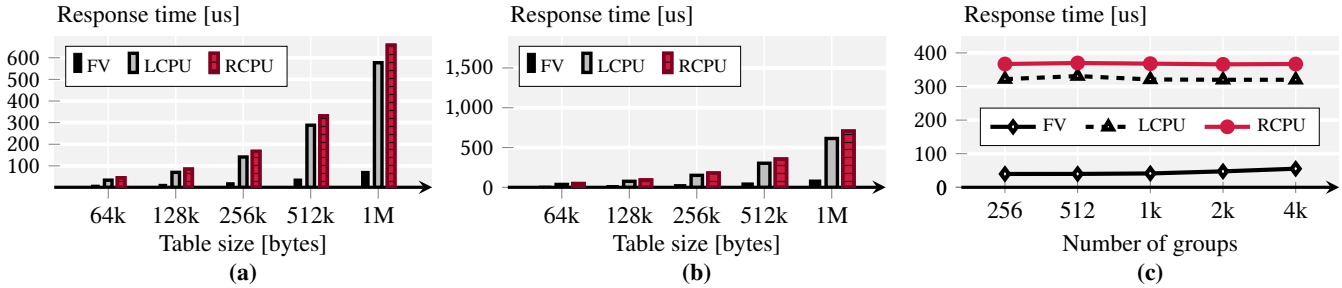

Figure 9: Response time comparisons for (a) a distinct query, (b) a group by query with aggregation on an increasing number of distinct elements, and (c) a group by query by with aggregation on a stable number of elements

slower response times than **LCPU**. The advantage of the bump-in-the-wire processing present in Farview is clear, especially as the amount of data becomes larger. We now go into specific details for each selectivity level.

On Figure 8(a), the query does not discard any tuples and it fetches the whole table. The query is equivalent to executing:

```
SELECT * FROM S;
```

Since no data is excluded from the selection predicate, **FV** and **RCPU** send the whole table through the network. Farview has similar performance for the vectorized (**FV-V**) and non-vectorized (**FV**) models of processing, because the available network bandwidth is the bottleneck (i.e., parallelization does not provide additional benefit). The memory bandwidth of the remote node is thus underutilized in this specific scenario.

Figure 8(b)'s 50% selectivity query alleviates the pressure on the network and permits more utilization of the DRAM bandwidth, leading to higher overall performance for Farview. In this case, the vectorized model is slightly more performant than the standard execution model, as the parallelization of the processing allows the reads from DRAM to occur at higher speeds. Still, even at this selectivity, the network is the bottleneck and the DRAM bandwidth is not fully utilized for a single client. The execution times of the baselines improve relative to 100% selectivity, especially as the input size grows, but they are still slower than both of Farview's execution models.

With 25% selectivity, only a small portion of the data is sent over the network, so the network is no longer the bottleneck. For **LCPU**, the data movement between the DRAM and the CPU is reduced because less data is written back; as a result, its performance is better when compared to 50% selectivity, but still worse than Farview. For Farview's non-vectorized model (**FV**), the bottleneck shifts to the bandwidth of a single query pipeline and performance is similar to the 50% selectivity case. The pipeline parallelization of the vectorized model (**FV-V**) can fully utilize the available memory bandwidth, and thus **FV-V** is roughly twice as fast as **FV**.

6.5 Grouping

In this section we evaluate the performance of the `DISTINCT` and `GROUP BY` grouping operators. We use the same baselines as in the selection experiments. For these operators, our baselines use a hashing implementation based on a very fast hash map library[2]. Figure 9(a) presents the results for the following query:

```
SELECT DISTINCT(S.a) FROM S;
```

The number of distinct elements is randomized and the results are averaged out over a number of runs. For the cuckoo hash table implemented in Farview, we assume that no hash collisions occur. In the case of collisions, they would be written in an overflow buffer and sent to the CPU for post-processing. Farview outperforms both baselines, and the baseline runtimes increase dramatically as the input size gets larger. We attribute part of the difference to reading from/writing to DRAM, as in the case of selection. Two additional factors contribute to the slowdown of the baselines: 1) the memory resizing of the hash table as more elements are added and 2) the ability of FPGAs to hash much faster than CPUs, even when the hash function is complex [42]. Additionally, in Farview the query is fully pipelined and there is not much overhead compared to the base selection. Finally, the number of distinct tuples has an impact on performance, similar to the impact of selectivity in the selection

[2]https://github.com/greg7mdp/parallel-hashmap

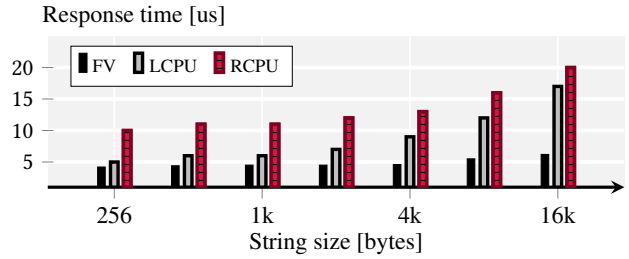

Figure 10: Regular expression matching

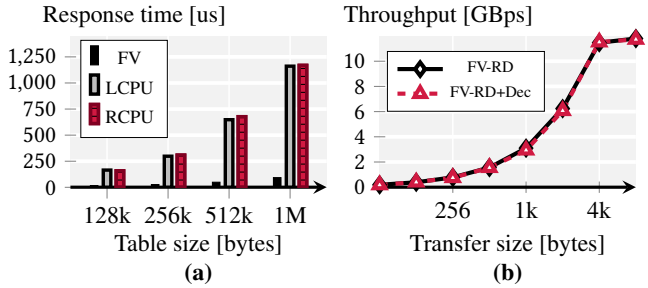

Figure 11: Encryption response time and throughput

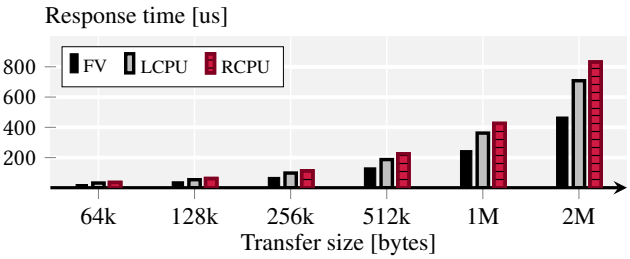

Figure 12: Multiple clients

experiments. The lower the number of distinct elements is, the lower the amount of data moved through the network, leading to better performance.

Next, we execute a simple query that has a `GROUP BY` and an aggregation (`SUM`):

```
SELECT S.a, SUM(S.b) FROM S GROUP BY S.a;
```

In Figure 9(b) and Figure 9(c), we present the performance of the query for increasing data sizes and number of tuples, respectively. The `GROUP BY` operator performs a similar operation as the `DISTINCT` operator. The main difference is that for `GROUP BY`, we calculate aggregated statistics for each of the distinct tuples in the hash table. After the aggregation is complete, all of the (unique) entries in the hash table along with their aggregations are sent over the network. This process adds a small amount of latency to the operator execution. The response time is thus bigger if the number of aggregates is higher. Even with this added latency, Farview outperforms the **LCPU** and **RCPU** baselines for both experiments, for the same reasons that were mentioned in the distinct experiment.

6.6 Regular expression matching

We compare the performance of Farview and the baselines for regular expression matching for different string sizes, where the regular expression matches 50% of the generated strings. The baselines use the highly optimized Google *RE2* regular expression library [35]. **FV** outperforms both **LCPU** and **RCPU**. **FV's** regular expression operator is able to sustain the full line rate regardless of the predicate complexity, due to its use of deep pipelining and parallel regular expression engines, which take advantage of the spatial architecture of the FPGA. This implementation outperforms *RE2*, as the implementation on the CPU has far less parallelization potential and the overhead of the data movement from/to DRAM is quite high.

6.7 Encryption/decryption

As an example of a commonly used system support operator, we explore encryption/decryption. The encryption algorithm used in Farview is a 128-bit AES in parallelized counter mode. The operator can be placed at the beginning or the end of the operator pipelines (handling decryption and encryption) with only a small extra overhead. This allows Farview to decrypt data residing in disaggregated memory for processing and sending it to the client; to encrypt data after processing to secure the transmission to the client; or to decrypt the data, process it, and encrypt it again for transmission. The response time graph in Figure 11(a) compares the time taken to decrypt data being read in Farview and in the two baselines. The ability of the FPGA to sustain line rate processing yields a big advantage, as the overhead of the encryption is fully hidden. **FV** significantly outperforms the **LCPU** and **RCPU** baselines, which use the same encryption/decryption scheme through the *Cryptopp* [23] library. The difference in execution time is caused by both the *cold* caches as well as the decryption overhead in the CPU compared with the highly parallelizable AES implementation available in the FPGA. The throughput graph in Figure 11(b) compares an RDMA read operation in Farview (**FV-RD**) and the same operation together with the decryption (**FV-RD+Dec**) on the read data stream. As the throughput graph shows, there is no noticeable performance penalty, indicating that encryption/decryption can be easily combined with all the previous operators without changing the overall performance.

6.8 Multiple clients

We finally evaluate the behaviour of the system with multiple clients reading from memory at the same time. We use six clients running the *distinct* query in both Farview and the two CPU baselines (**LCPU**, **RCPU**). The number of distinct elements is small to prevent the network from becoming the main bottleneck and to maximize DRAM performance in all of the clients. For the CPU baselines, we use MPI with 6 processes. The measurements shown in Figure 12 represent the time taken until all six client queries have completed. Farview achieves better performance than both CPU baselines due to the spatial parallelization between multiple dynamic regions, each containing a separate client. The decoupling of the DRAM and the fair sharing (Section 4.4) provide optimal distribution of the DRAM bandwidth between all dynamic regions and their clients. Both CPU baselines compete for access both to the DRAM and the shared caches, causing interference that affects the overall performance.

7 CONCLUSIONS

Farview implements network-attached disaggregated memory with the capability to offload query processing operators directly to the memory. In this paper, we have discussed the design of Farview and how it helps to address DRAM capacity challenges (by allowing us to move the buffer pool to a central location) and data movement inefficiencies (by enabling near-data processing to filter the data before it is sent through the network). Through the use of RDMA, Farview provides performance that is comparable to that attainable using local memory, a performance advantage that is augmented by the ability to process data in-situ. The next steps for the Farview project are to develop a query optimizer that takes the new capabilities of the system into consideration, to design suitable cache management strategies to move data back and forth to persistent storage, and to expand the range of operators supported. We also want to explore, as part of a query optimizer, options such as performing joins against small tables in the memory by reading the small table into the FPGA and matching the tuples read from memory against it. Additionally, we plan to port Farview to Enzian [21], a heterogeneous research platform that offers both more network and memory bandwidth.

ACKNOWLEDGEMENTS

We would like to thank Xilinx for the generous donation of the equipment used to perform the experiments in the paper and for access to the XACC ETHZ Cluster. The work of Dario Korolija has been funded in part by a donation from HPE.

REFERENCES

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions: a Simple Abstraction for Remote Memory. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. 775–787.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing*. 121–127.

[3] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 120–126.

[4] Kathirgamar Aingaran, Sumti Jairath, and David Lutz. 2016. Software in Silicon in the Oracle SPARC M7 Processor. In *2016 IEEE Hot Chips 28 Symposium (HCS)*. 1–31.

[5] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. 2019. DPI: The Data Processing Interface for Modern Networks. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Online Proceedings*.

[6] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software Co-design from a Database Perspective. In *CIDR'20*.

[7] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 14:1–14:16.

[8] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Peter J. Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer* 29, 2 (1996), 18–28.

[9] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *ICDE'15*. 435–446.

[10] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1753–1758.

[11] Joy Arulraj and Andrew Pavlo. 2019. Non-volatile Memory Database Management Systems. *Synthesis Lectures on Data Management* 11 (2019).

[12] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. 2019. Multi-tier Buffer Management and Storage System Design for Non-volatile Memory. *arXiv preprint arXiv:1901.10938* (2019).

[13] AWSCloud. 2020. *AQUA (Advanced Query Accelerator) for Amazon Redshift*. https://pages.awscloud.com/AQUA_Preview.html

[14] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1463–1475.

[15] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment* 10, 5 (2017), 517–528.

[16] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proc. VLDB Endow.* 12, 13 (2019), 2325–2338.

[17] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* (2016).

[18] Peter A Boncz, Martin L Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.

[19] K. M. Bresniker, S. Singhal, and R. S. Williams. 2015. Adapting to Thrive in a New Economy of Memory Abundance. *Computer* 48, 12 (2015), 44–53.

[20] Adrian Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, and et. al. 2016. A Cloud-Scale Acceleration Architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[21] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: an open, general, CPU/FPGA platform for OS research. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[22] Oracle Corporation. [n. d.]. Accelerating Spark SQL Using SPARC with DAX.

[23] CryptoPP. Accessed 2021. Crypto++® Library 8.6. https://www.cryptopp.com/.

[24] William Dally. 2011. Power, programmability, and granularity: The challenges of exascale computing. In *2011 IEEE International Test Conference*. IEEE Computer Society, 12–12.

[25] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[26] ARM developer. 2021. *About the AXI4-Stream protocol*. https://pages.awscloud.com/AQUA_Preview.html

[27] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*.

[28] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *European Conference on Computer Systems*.

[29] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *European Conference on Computer Systems (EuroSys) 2018*.

[30] Yuanwei Fang, Chen Zou, and Andrew Chien. 2019. Accelerating Raw Data Analysis with the ACCORDA Software and Hardware Architecture. In *Proc. VLDB Endow.*

[31] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[32] Daniel Firestone, Andrew Putnam, Hari Angepat, Derek Chiou, Adrian Caulfield, Chun Chung, Matt Humphrey, and et. al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*.

[33] Carsten Binnig Andrew Crotty Alex Galakatos and Tim Kraska Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment* 9, 7 (2016).

[34] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *OSDI*. 249–264.

[35] Google. Accessed 2021. re2 Library. https://github.com/google/re2.

[36] Goetz Graefe. 1990. Encapsulation of parallelism in the volcano query processing system. *ACM SIGMOD Record* 19, 2 (1990), 102–111.

[37] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[38] InfiniBand. Accessed 2021. InfiniBand™ Architecture Volume 1 and Volume 2. https://www.infinibandta.org/ibta-specification/.

[39] InfiniBand. Accessed 2021. InfiniBand™Architecture Specification Release 1.2.1 Annex A16: RoCE. https://cw.infinibandta.org/document/dl/7148.

[40] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.* (Aug. 2017), 1202–1213.

[41] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A High-Performance Database System Leveraging In-Storage Computing. *Proc. VLDB Endow.* 9, 12 (2016), 924–935.

[42] K. Kara and G. Alonso. 2016. Fast and robust hashing for database operators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–4.

[43] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.* 27, 3 (1998), 42–52.

[44] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 107–127.

[45] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. HyperLoop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 297—-312.

[46] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 991–1010.

[47] H. Andrés Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[48] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009)*. 267–278.

[49] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. 189–200.

[50] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2019. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. *ACM Trans. Database Syst.* 44, 4 (2019), 17:1–17:45.

[51] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *ASPLOS '20*. 827–844.

[52] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *ASPLOS'20*. 757–773.

[53] Mellanox. Accessed 2021. ConnectX®-5 VPI Card. https://www.mellanox.com/files/doc-2020/pb-connectx-5-vpi-card.pdf.

[54] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference ATC 2015*.

[55] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking *(SIGCOMM '18)*. 327–341.

[56] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[57] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.

[58] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, ISCA 1998, Barcelona, Spain, June 27 - July 1, 1998*. 192–203.

[59] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules. In *MEMSYS 2019*. 288–303.

[60] David A. Patterson, Thomas E. Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos E. Kozyrakis, Randi Thomas, and Katherine A. Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (1997), 34–44.

[61] David A. Patterson, Krste Asanovic, Aaron B. Brown, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Christoforos E. Kozyrakis, David R. Martin, Stylianos Perissakis, Randi Thomas, Noah Treuhaft, and Katherine A. Yelick. 1997. Intelligent RAM (IRAM): The Industrial Setting, Applications and Architectures. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. 2–7.

[62] Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the Intel optane byte-addressable NVM. In *MEMSYS '19: Proceedings of the International Symposium on Memory Systems*. 304–315.

[63] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, and et. al. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*.

[64] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *FPL 2019*. 286–292.

[65] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *OSDI*. 69–87.

[66] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *ACM Symposium on Cloud Computing (SOCC) 2017*.

[67] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *EuroSys'20*. 29:1–29:16.

[68] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *USENIX ATC 2020*.

[69] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 1541–1555.

[70] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*.

[71] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *Proc. VLDB Endow.* 7, 11 (2014), 963–974.

[72] Xilinx. Accessed 2021. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html#specifications.

[73] Xilinx. Accessed 2021. Xilinx Adaptive Compute Cluster (XACC) Program. https://www.xilinx.com/support/university/XUP-XACC.html.

[74] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *ASPLOS'20*. 845–858.

[75] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Online Proceedings*. www.cidrdb.org.

[76] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow.* 13, 9 (2020), 1568–1581.

[77] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*. 741–758.

[78] Marcin Zukowski and Peter A Boncz. 2012. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27.