# Leveraging high-performance data transfer to offload data management tasks to SmartNICs

Scott Levy, Whit Schonbein
*Sandia National Laboratories*
Albuquerque, New Mexico, USA
{sllevy, wwschon}@sandia.gov

Craig Ulmer
*Sandia National Laboratories*
Livermore, California, USA
cdulmer@sandia.gov

***Abstract*—Network interface controllers (NICs) with general-purpose compute capabilities ('SmartNICs') present an opportunity for reducing host application overheads by offloading non-critical tasks to the NIC. In addition to moving computation, offloading requires that associated *data* is also transferred to the NIC. To meet this need, we introduce a high-performance, general-purpose data movement service that facilitates the offloading of tasks to SmartNICs: The SmartNIC Data Movement Service (SDMS). SDMS provides near-line-rate transfer bandwidths between the host and NIC. Moreover, SDMS's In-transit Data Placement (IDP) feature can reduce (or even eliminate) the cost of serializing data on the NIC by performing the necessary data formatting during the transfer. To illustrate these capabilities, we provide an in-depth case study using SDMS to offload data management operations related to Apache Arrow, a popular data format standard. For single-column tables, SDMS can achieve more than 87% of baseline throughput for data buffers that are 128 KiB or larger (and more than 95% of baseline throughput for buffers that are 1 MiB or larger) while also nearly eliminating the host and SmartNIC overhead associated with Arrow operations.**

***Index Terms*—Network interfaces, SmartNICs, data analytics**

## I. Introduction

Network interface controllers (NICs) with general-purpose compute capabilities – variously referred to as SmartNICs, Data Processing Units (DPUs) or Infrastructure Processing Units (IPUs) – present an opportunity for reducing host application overheads by offloading tasks to the NIC. While SmartNICs typically have less powerful compute resources and less plentiful memory resources than the associated host, offloaded computation can be overlapped with the execution of the host application yielding an overall benefit. Researchers have thus proposed utilizing SmartNICs for offloading tasks including distributed databases [1], [2], security [3], system services [4], and collective communication [5].

Effective offloading requires that host data be migrated to the SmartNIC. For example, Bayatpour et al. [5] used SmartNIC memory to stage host data to be exchanged in MPI collective operations, and Ulmer et al. [2] moved data from host memory to the SmartNIC to facilitate their insertion in a distributed database. There is an existing need for a low-overhead, high-bandwidth method for transferring memory contents from host to SmartNIC [2]. To address this need, we introduce a high-performance, general-purpose data movement service that facilitates the offloading of tasks to SmartNICs: The **S**martNIC **D**ata **M**ovement **S**ervice (SDMS). SDMS provides near-line-rate transfer bandwidths between host memory and SmartNIC memory. Moreover, SDMS's In-transit Data Placement (IDP) feature can reduce (or even eliminate) the cost of data placement operations (e.g., serialization) on the SmartNIC by performing the necessary data formatting during the transfer. In this paper, we use benchmarks to assess the cost borne by the host to initiate data transfers with SDMS, characterize approaches for maximizing transfer bandwidth, and demonstrate that SDMS's transfer bandwidths compare favorably to results from a standard bandwidth benchmark.

We also provide an in-depth case study to characterize the performance of using SDMS to facilitate the offload of data management operations related to Apache Arrow [6], a framework widely used in data analytics. We benchmark the multi-step process of converting raw output data to a serialized Apache Arrow object to identify candidates for offloading, and evaluate the degree to which IDP can help maximize SmartNIC throughput. For single-column tables, SDMS using IDP can achieve more than 87% of baseline throughput for data buffers that are 128 KiB or larger (and more than 95% of baseline throughput for buffers that are 1 MiB or larger) while significantly reducing the host overhead and increasing the SmartNIC's throughput for Arrow data transformation operations. The primary contributions of this paper are:

- SmartNIC Data Movement Service, a flexible, high-performance, general-purpose service for transferring data from host to SmartNIC that facilitates offloading work to the SmartNIC to reduce host overheads (§II);
- Comprehensive performance analysis of SDMS, demon-

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan https://www.energy.gov/downloads/doe-public-access-plan.

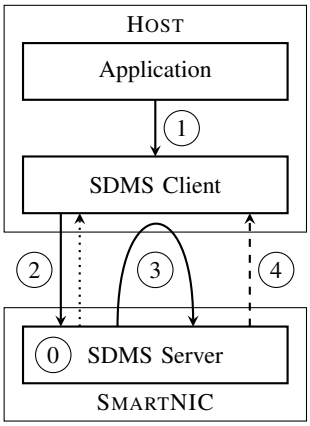


Fig. 1: High-level operation of SDMS

strating that it achieves transfer bandwidths that compare favorably to results from a standard bandwidth benchmark (§IV);
- Detailed analysis of the multi-step process of converting data to serialized Apache Arrow [6] objects to identify work to offload to the SmartNIC (§§V-D,V-E); and
- A case study evaluating the performance benefit of offloading Apache Arrow operations to a SmartNIC with SDMS, including the benefits of using the IDP feature of SDMS to manipulate the data in-transit (§§V-F,V-G).

## II. SDMS

The SmartNIC Data Movement Service (SDMS) provides host applications with high-performance data transfer to facilitate offloading tasks to the SmartNIC while minimizing host involvement.

### A. Design Goals

We identified two design objectives for SDMS:
- *Minimize Host Involvement*: A principle goal of offloading is to free up resources on the host processor(s). To meet this goal, SDMS is designed to use RDMA `get` operations issued by the SmartNIC to minimize host involvement in the data transfer. Transfers are acknowledged lazily to eliminate the need for the host to be interrupted to process acknowledgements.
- *Maximize SmartNIC Throughput*: Because the processors on a SmartNIC are frequently less powerful and have less memory (both cache and bulk memory) than the associated host processor, offloaded work may execute more slowly on the SmartNIC than it would on the host. As a result, there is a risk that the offloaded work could become a bottleneck. As detailed below, SDMS addresses this issue by providing high-performance data transfer, and utilizing in-transit data placement to reduce SmartNIC load.

### B. Operation of SDMS

SDMS builds on the services provided by `hodcarrier` [7]. The `hodcarrier` library provides a simple, open-source wrapper around the InfiniBand Verbs API that hides the low-level details of interface (e.g., managing queue pairs, polling completion queues) to simplify the development of higher-level services, including SDMS. SDMS interfaces with the application and manages the process of transferring the data associated with the offloaded computation. Figure 1 depicts the basic sequence of operations for transferring data from host memory to the SmartNIC. A detailed description of the transfer process is provided below:

⓪ During the initialization of SDMS, the application can optionally provide the Server with the size(s) of the data buffers that will be processed to enable the Server to improve performance by pre-allocating and caching destination buffers before the application runs.

① When the application has data that is ready to transfer, it calls the SDMS Client API and provides: (i) the memory addresses of the buffers to transfer; (ii) the sizes of each of the buffers to transfer; and (iii) the remote keys for the RDMA transfer of the buffers.[1]

② The SDMS Client sends the request to the SDMS Server on the SmartNIC (solid, downward arrow). The SDMS Server running on the SmartNIC acknowledges the request to the Server (dashed, upward arrow).

③ The SDMS Server acquires a destination buffer for the transfer and initiates an RDMA Read operation (`IBV_WR_RDMA_READ` opcode).

④ After the buffer has been transferred, the SDMS Server notifies the SDMS Client that the transfer is complete and the source buffer can be reused. Completion notification is done lazily to eliminate the need for the Server to check for message arrival; the notification is piggybacked on the next message from the Server to the Client.

The SDMS library was designed to offload data transfer to the greatest extent possible from the host to the SmartNIC. The only required host participation is preparation of the buffers for RDMA transfer (e.g., ensuring that memory has been registered) and to request the transfer from the SDMS Server. All other processing is done on the SmartNIC.

### C. In-transit Data Placement (IDP)

SDMS implements an optimization that performs simple data placement operations in the process of transferring data from host memory to SmartNIC memory. For example, if we have three individual memory buffers in host memory that we ultimately wish to consolidate into a single contiguous memory buffer, we can allocate a contiguous buffer in SmartNIC memory and configure the RDMA Get operations such that the three buffers are coalesced into the single memory buffer in SmartNIC memory as part of the transfer. In the absence of this feature, either the host or the SmartNIC would need to explicitly serialize the data (i.e., copy the buffers into a single memory buffer either before (host) or after (SmartNIC) the

[1]RDMA operates on the level of memory regions so we use that terminology here. However, it is straightforward to acquire contiguous memory regions from common, high-level data structures (e.g., C++ STL containers).

transfer). By incorporating simple data movement operations into the transfer to SmartNIC memory, we can reduce (or even eliminate) the need for explicit memory copies. As we demonstrate later in this paper (*see* Section V-G) SDMS's IDP feature can be used to help serialize data structures and to manipulate the contents of existing data structures.

## III. Experimental setup

The data in this paper was collected on Glinda, a High Performance Data Analytics (HPDA) cluster. Each compute node has a single 32-core, 2.8GHz AMD EPYC 7543P processor and 512 GiB of DDR4 memory. Each compute node also includes an NVIDIA Ampere A100 GPU, and an NVIDIA BlueField-2 VPI Data Processing Unit (DPU). The interconnect is 100 Gb/s InfiniBand.

## IV. Benchmarking SDMS

In this section, we evaluate our implementation of SDMS in the context of the design objectives described in Section II-B.

### A. Minimizing Host Offload Overhead

An important benefit of offloading data management tasks to the SmartNIC is to free the host processor so that it can do other work (e.g., continuing to run the application that is generating the data) in parallel with data processing. SDMS is designed to minimize host involvement; once the host has called the SDMS to send its transfer request to the SDMS Server on the SmartNIC, it is free to resume its other work. As a result, we expect that the cost of data transfer on the host should be relatively constant and independent of message size. The data in Figure 2 confirms this expectation. This figure shows the time that the host is involved in the transfer data to the SmartNIC with SDMS. The data shown in green and blue represent the minimum and maximum measurements, respectively. The orange data represent the average time measurements. While there is some variation across buffer sizes, the averages are all between 250 and 350 $\mu$s. Moreover, there is not an observable correlation between the time required to request a transfer and the buffer size: the Pearson correlation coefficient, $R$, for the average time is 0.379 (indicating a weak, positive correlation). With respect to the absolute time required to initiate a transfer from the host, the current implementation is built on Linux sockets over the management interface. As a result, while there is room to optimize this transfer, its current performance (100s of microseconds to initiate a transfer) means that the magnitude of the benefit of this optimization would be modest.

*Summary: SDMS facilitates data transfer from host memory to SmartNIC with very modest, size-independent host processing requirements, fulfilling our first design objective.*

### B. Maximizing SmartNIC Throughput

To characterize the impact of the cost of data transport on SmartNIC throughput for SDMS, we use an existing InfiniBand bandwidth benchmark, `qperf` [8], to establish an upper bound on the network transfer bandwidth. Using `qperf`

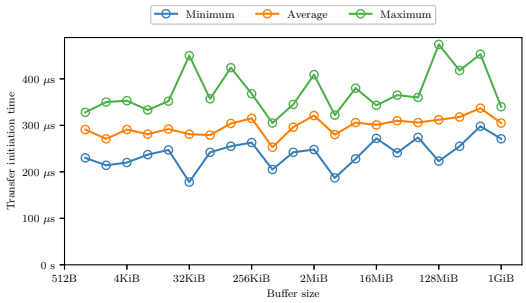



Fig. 2: Host time to initiate an SDMS transfer

enables us to evaluate the extent to which SDMS is able to use the full network bandwidth to transfer data from host memory to SmartNIC memory. Figure 3 compares the empirical bandwidth of SDMS with two different configurations of `qperf`. The SDMS data in this figure represent the average over 10 trials. The `qperf` data represent the average that it reports. The `qperf` results are collected by launching a server process on the host and then connecting to that server from the SmartNIC and executing the `rc_rdma_read_bw` test (i.e., ultimately executing RDMA Get operations to transfer data from host memory to the SmartNIC).

Bandwidth benchmarks, including `qperf`, initiate large numbers of transfer operations at once to saturate the network and maximize measured bandwidth. The number of requests posted by `qperf` is controlled in the source code by the value of `NCQE`. By default, the value of `NCQE` is 1024 and is not modifiable at runtime. While initiating a large number of transfers in this fashion establishes a valuable upper bound, users commonly want to transfer one buffer (or a small number of buffers) at a time and are thus generally not able to reach this upper bound (especially for small message buffers). As a result, to get a more realistic upper bound on transfer bandwidth, we modified `qperf` to allow the value of `NCQE` to be set at runtime.

The blue line in Figure 3 shows the bandwidth measured by `qperf` as a function of the size of the message buffer for the default value of `NCQE` (1024). This version of `qperf` is able to achieve full network bandwidth ($\approx$ 100 Gbps) for message buffers that are between 8 KiB and 4 MiB in size. For messages larger than 4 MiB, the achievable bandwidth drops by approximately 15%. We expect that, in most cases, our applications will request the transfer of small numbers of buffers at a time (e.g., once per timestep). As a result, the measurements with `NCQE` set to 1 represent a fairer baseline to compare the performance of SDMS against. Therefore, the orange line shows the bandwidth measured by `qperf` with the value of `NCQE` set to 1. These data show that, for small message buffers, the measured bandwidth is significantly lower for sending single message buffers (`NCQE`=1) than the default approach (`NCQE`=1024). The green line represents bandwidth measured using SDMS. To ensure a fair comparison with `qperf`, we implemented this benchmark to ensure that

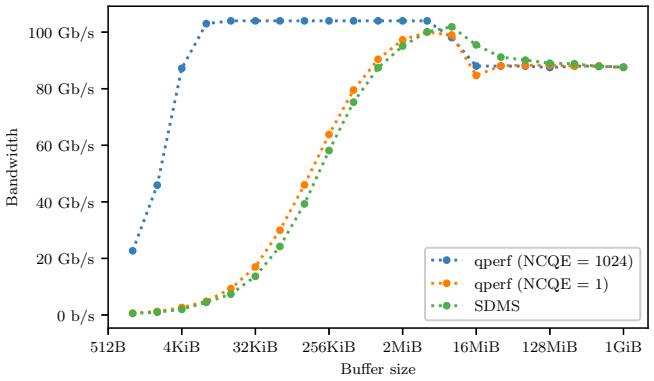
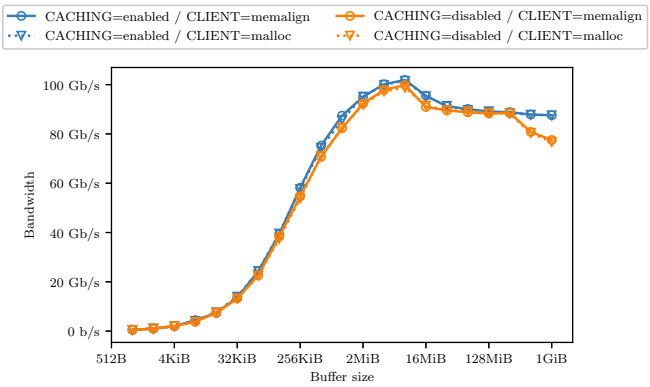


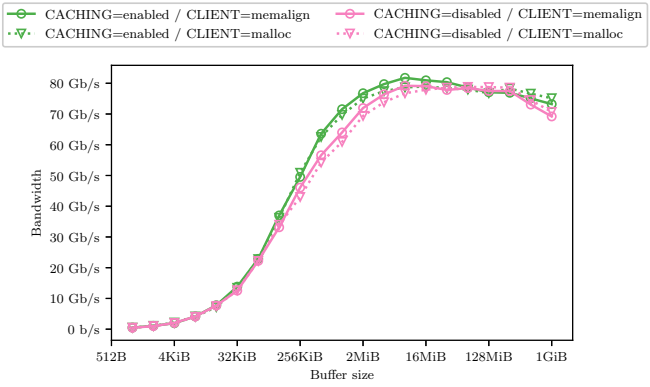

Fig. 3: Comparison of bandwidth measured with SDMS and bandwidth measured with the `qperf` benchmark.

buffers are registered and reused such that memory registration costs are not included in our measurements. The data in this figure show that, for messages that are 64 KiB or larger, the bandwidth measured with SDMS is never less than 85% of the bandwidth measured with `qperf` (`NCQE` = 1), and in some cases is higher (e.g., for 16 MiB buffers).

The default buffer management strategy for SDMS uses two principal optimizations to ensure high performance. The first is to allocate memory with `memalign` to align buffers on the host (source) and SmartNIC (destination) buffers to page boundaries. The second is to pre-allocate and cache registered buffers in SmartNIC memory to maximize reuse and reduce the overhead of memory registration. Figure 4 characterizes the impact that each of these optimizations have on the transfer bandwidth achieved by SDMS. These two figures evaluate eight combinations of the three configuration options. The top figure (Figure 4a) shows the transfer bandwidth when the SmartNIC buffers are allocated on page boundaries with `memalign` while buffer caching is enabled/disabled on the SmartNIC, and the use of `malloc` or `memalign` is varied on the client (i.e., the host). The bottom figure (Figure 4b) shows the transfer bandwidth when the destination buffers are allocated with `malloc` and are not guaranteed to be page-aligned. These data show that the critical optimization is to use SmartNIC buffers that are page-aligned. The peak transfer bandwidth achieved for trials with page-aligned destination buffers is more than 24% higher than the peak achieved with non-page-aligned destination buffers. These data show that buffer caching may yield an additional improvement, especially for large buffers. For 1 MiB SmartNIC buffers, caching increased the transfer bandwidth by nearly 6% for page-aligned buffers (Figure 4a) and nearly 11% for non-page-aligned buffers (Figure 4b).




(a) Server (SmartNIC) buffer allocation with `memalign`




(b) Server (SmartNIC) buffer allocation with `malloc`

Fig. 4: Bandwidth measurements to characterize the impact of different buffer management strategies

*Summary: The raw transfer bandwidth from host memory to SmartNIC memory provided by SDMS (near-line-rate) is sufficient to prevent data transfer itself from being a bottleneck, consistent with our second design objective for SDMS. Page-aligned destination buffers are required to achieve this level of performance.*

## V. Case Study: Offloading Apache Arrow Transformation and Serialization

In this section, we describe and analyze an example of how SDMS can be applied to offload data management operations, with an emphasis on avoiding overwhelming the relatively modest compute resources available on the SmartNIC. Specifically, we describe a workflow that uses Apache Arrow [6] to connect the output of scientific applications to subsequent phases of analysis and visualization.

### A. Connecting Workflow Applications with Apache Arrow

Workflows on HPC systems frequently need to migrate data between one parallel simulation or analysis job's memory space to another's as efficiently as possible [9]. One of the challenges of facilitating these transfers is agreeing upon a common data representation that all parties support. While HPC users have historically relied on visualization libraries

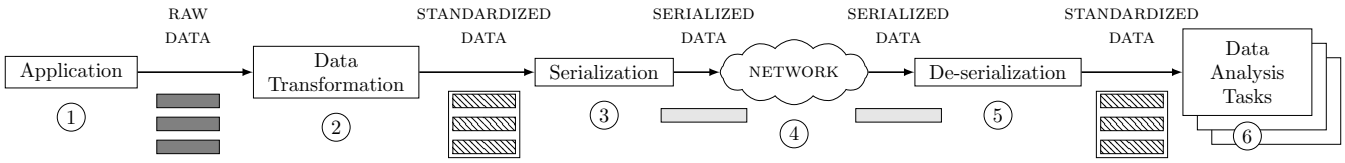



Fig. 5: An overview of the steps involved to transform simulation output data stored in application-native data structures to Arrow objects that are provided as input to data analysis and visualization tasks.

to export simulation results to analysis tools [10], there is significant interest in integrating the data science community's analysis tools into HPC workflows, including tools that use machine learning and AI techniques to extract insight. Fortunately, many of these tools have already been standardized to provide data interoperability via Apache Arrow.

Apache Arrow is an open-source library for efficiently representing, processing, transmitting, and storing tabular data [6]. At its core, Arrow implements a space-efficient, in-memory columnar store in C++ and features a robust API for manipulating tabular data that has been adapted to several languages, including C++, Python, Go, R, and Rust. Users represent their data in one or more two-dimensional tables, each with its own user-defined schema. Individual columns are implemented as chunked arrays to enable users to split and combine table components efficiently, as well as to support both batch and streaming computations. Arrow includes a compute framework to apply database-like queries to a table that is both thread- and SIMD-aware. Most importantly for our work, Arrow defines an on-wire data format for inter-process communication (IPC) that makes it easier for applications to serialize and share data efficiently. As such, Arrow has the potential to serve as a conduit for HPC applications to exchange data with dozens of Arrow-enabled environments [11], including Pandas [12], Apache Spark [13], Dask [14], Ray [15], and GeoMesa [16].

Previous work has demonstrated that Arrow is well-suited for representing particle simulation data and that tables serialized into Arrow's IPC format can be revived and inspected by a SmartNIC with minimal overhead [1]. In addition to providing a mechanism for users to query the live content of a simulation's in-transit data [17], Arrow can be used to refine data at the SmartNIC. In a particle-sifting example, a distributed group of SmartNICs reorganized simulation results to simplify work for downstream consumers [2]. In this example, each simulation rank on the host converts particle data to an Arrow format and streams updates to the SmartNIC. Once a SmartNIC acquired sufficient data, an Arrow filter operation grouped particles by ID to enable data to be distributed to the corresponding SmartNICs.

### B. Case Study Overview

Figure 5 depicts an example workflow that connects simulation output to analysis tasks. Production workflows typically have additional steps (e.g., mesh generation, multi-physics coupling) but for the purpose of our case study, we have focused on the data analysis and visualization portion of the workflow. Data in workflows like this run the simulation and analysis tasks on separate job allocations. Data is commonly exchanged between these two stages through a parallel file system, however, in-memory storage (e.g., Faodel [18], DataSpaces [19]) could also be used.

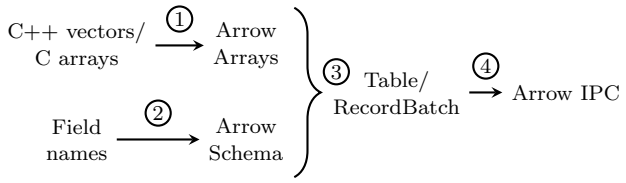



Fig. 6: Converting C++ vectors or C arrays to Arrow IPC.

In this workflow, data transformation to the Arrow data format is a multi-step process. (1) The simulation generates data in an application-defined format; for instance, in the example described in Section V-A, the application output comprises C++ STL vectors containing particle locations, velocities, etc. (2) Raw data is transformed into an Arrow Table or RecordBatch object. (3) The Table or RecordBatch is serialized into Arrow IPC format. (4) The serialized data is transferred over the network to the analysis tasks and (5) de-serialized to recover the Arrow object and potentially integrated with data from other processes. Note that Arrow enables de-serialization to be done in-place with very low overheads. Following the de-serialization step, the requested analysis can be performed.

Figure 5 shows that the burden of transforming application output to Arrow objects and serializing it is borne by the application job running the principal computation. This presents an opportunity to reclaim host resources by offloading these operations to the SmartNIC. Moreover, SDMS can reduce the amount of work that must be done on the SmartNIC by directly placing host data into appropriate data structures in SmartNIC memory, *see* Section II-C. In the remainder of this section, we characterize the potential benefit of using SDMS to offload Arrow transformation and serialization.[2]

### C. Arrow Data Serialization

Arrow provides two mechanisms for transforming data into serialized Arrow RecordBatches: (i) one based on the Arrow C++ API; and (ii) an alternative using the lightweight Arrow

[2]We could also consider offloading the de-serialization operation in the analysis job, but given that the overhead of this operation for Arrow IPC buffers is very low, the benefit in this instance is also low.

| Step | C++ | C |
|---|---|---|
| ① | For each vector:<br>`NumericBuilder.AppendValues`<br>`NumericBuilder.Finish` | Create a parent `ArrowArray` struct.<br>For each array:<br>Create child `ArrowArray` structure |
| ② | For each column name:<br>`arrow::Field` constructor<br>Once:<br>`arrow::schema` constructor | Create a parent `ArrowSchema` struct.<br>For each array:<br>Create child `ArrowSchema` structure |
| ③ | `arrow::RecordBatch::Make` | `arrow::ImportRecordBatch` |
| ④ | `arrow::ipc::MakeStreamWriter`<br>`writeRecordBatch`<br>`GetExtentBytesWritten`<br>`arrow::AllocateResizableBuffer`<br>`writeRecordBatch` | `arrow::ipc::MakeStreamWriter`<br>`writeRecordBatch`<br>`GetExtentBytesWritten`<br>`arrow::AllocateResizableBuffer`<br>`writeRecordBatch` |

TABLE I: Arrow API calls for each step in Figure 6.

C application-binary interface (ABI). The latter has the benefit of allowing an application to avoid linking the Arrow libraries by instead including a header defining C structures that can subsequently be processed directly by the Arrow C++ libraries. However, both mechanisms follow the same general sequence of steps, shown in Figure 6: ① convert application-native data structures (e.g., C++ STL vectors or C arrays) into Arrow objects (C++ Arrow Arrays), ② convert a list of field names (i.e., text strings describing each column) to an Arrow Schema object, ③ combine the Arrow Array and Schema objects to create a Arrow RecordBatch object, and ④ serialize the RecordBatch into a buffer stored in the Arrow IPC format. At this point, the serialized data can be moved off-node.

The C++ and C approaches differ in their respective sequence of calls, as shown in Table I. For the C++ API, vectors provided by the application are transformed into IPC format through a sequence of calls to C++ methods. In contrast, with the C ABI, tree-like collections of C structures (defined in an Arrow header) are created by the application to encapsulate the data and the text labels for those arrays. Steps ① and ② represent the creation of those collections, which are then imported into a RecordBatch using the C++ Arrow API in step ③. The code for the final step is identical for both approaches; `writeRecordBatch` appears twice because a dummy write is necessary to determine the required IPC buffer size.

### D. Identifying Targets for Offloading

To assess the amount of host CPU time that could be recovered by using SDMS to offload Arrow conversion steps to the SmartNIC, we developed a benchmark to evaluate the C++ and C serialization approaches. Given a total data size (in bytes) and a number of columns, the benchmark creates a collection of C++ vectors (for C++) or arrays (for C) of equal length (one per RecordBatch column) and whose data sizes sum to the total requested amount. Randomly generated 64-byte strings are used for the column names in our experiments. The benchmark then executes the sequence of operations summarized in Table I, timing the duration of each step.

All experiments are executed on the host nodes of the system described in Section III, using Arrow v11.0.0. Total data sizes range from 128B to 1GiB by powers of 2. The number of columns range from 1 to 512, and results are averaged over ten runs on ten different nodes.

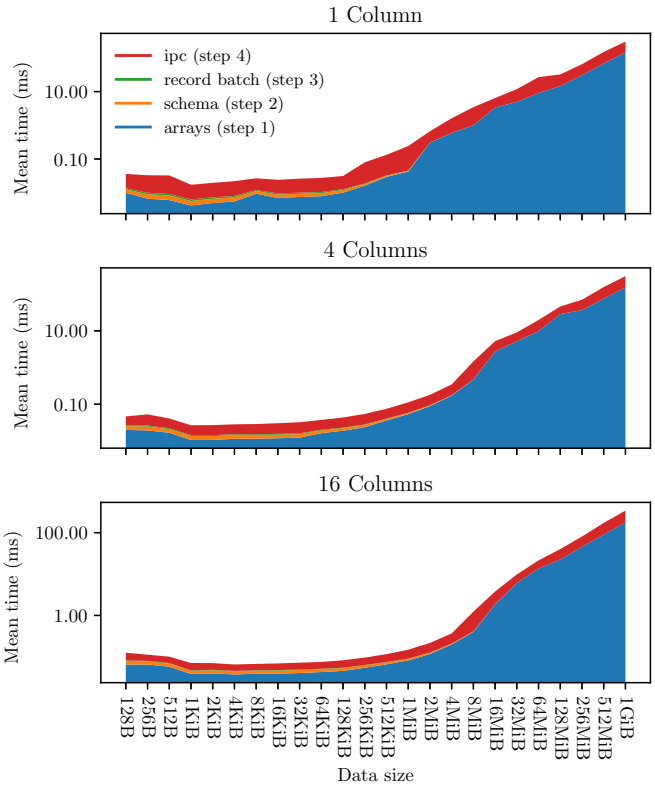



Fig. 7: Per-step contribution to the total time for Arrow IPC conversion using the C++ API on the host.

*1) C++ API Results:* Figure 7 shows the contribution of each step to the total serialization time using the C++ API, for tables with 1, 4 and 16 columns; results for tables with other numbers of columns are similar insofar as the proportions of each step are comparable. As can be seen from the figure, serialization time is dominated by converting data to Arrow Arrays (step ① in fig. 6) and serializing the Arrow RecordBatch object to an IPC buffer (step ④). Taken together, these two steps account for approximately 86% to 99.9% of the conversion time for data buffers between 128B and 1GiB in size[3], regardless of number of columns. Therefore, the benchmark results indicate that offloading before steps ④ or ① (which will include offloading ④) will provide the most benefit. However, offloading only step ④ is superfluous because transferring its input data to the SmartNIC would itself require serialization on the host.

*Summary: For the C++ API, the best candidate for offloading work is before execution of step ① (Array creation).*

*2) C ABI Results:* Figure 8 shows the contribution of each step to the total serialization time using the C ABI. Because under this method exporting arrays and schema is

[3]Note that the $y$-axes are $\log_2$ and the results for the individual steps are stacked. Therefore, the results for steps ① and ④ are comparable in magnitude despite the visual appearance in Figure 7.

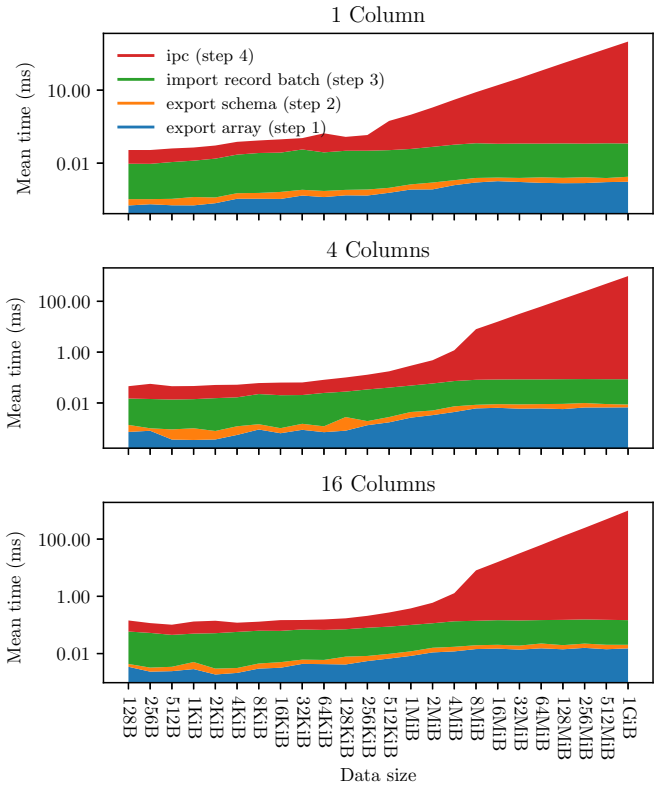



Fig. 8: Per-step contribution to the total time for Arrow IPC conversion using the C ABI on the host.

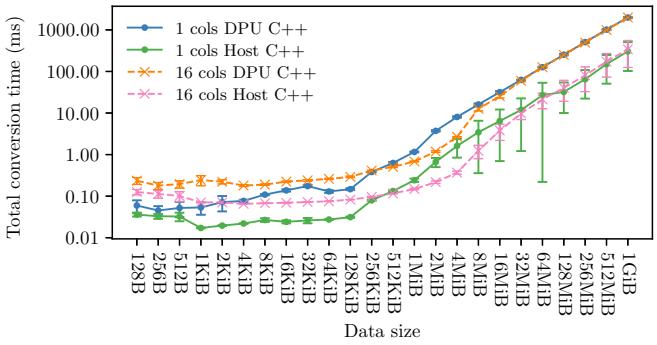



Fig. 9: Comparison of total conversion time using C++ method on BlueField-2 DPU and host. Error bars show standard deviation.

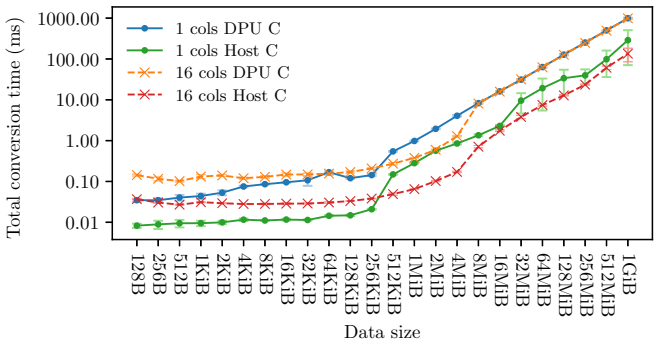



Fig. 10: Comparison of total conversion time using C method on BlueField-2 DPU and host. Error bars show standard deviation.

nothing more than setting up several C structures, the cost for these steps (① and ②) never exceeds 2% of the total conversion time, regardless of the number of columns or data size. For small total data sizes, the time required to import a RecordBatch (step ③) is a significant fraction of the total (e.g., approximately 30% for the single column case), but rapidly drops for data sizes larger than 512KiB until it is less than 2% for 2MiB and larger. As a result, for all numbers of columns, conversion time is dominated by the need to copy data during IPC (step ④), ranging from approximately 55% to 99.9% of the total conversion time as the data size increases from 128B to 1GiB. However, as observed for the C++ API case, the value of offloading ④ by itself is limited because the act of preparing the data for movement after ③ would itself require serialization by the host. Similarly, offloading after ② would require serializing the objects that form the input of ③. Consequently, offloading all four steps (i.e., prior to step ①) promises the best results.

*Summary: For the C interface, the best candidate point for offloading work is prior to step ①.*

### *E. Relative Overheads of SmartNIC Offloading*

Because of the relative weakness of computational resources on a BlueField SmartNIC in comparison to the host (*cf.* [20], [21]) we also evaluated the cost of running the Arrow conversion and serialization task on the SmartNIC relative to the host. This is important because by Little's law, slower SmartNIC processing speeds can limit throughput, thereby increasing the amount of SmartNIC resources (e.g., memory) required to service the host application and increasing the potential for the SmartNIC to become a bottleneck.

Executing the benchmark on the SmartNICs revealed that the relative contributions to the total serialization time are consistent with those observed for the host (Figures 7 and 8): For C++, total times are dominated by Array and IPC serialization, and for C, total times are dominated by RecordBatch import and IPC. However, the Arrow conversion process is significantly slower on the SmartNIC than on the host. Figures 9 and 10 compare the average total Arrow conversion times for the host and the SmartNIC using the C++ and C methods, for 1 and 16 columns with data sizes ranging from 128B to 1GiB, in powers of 2. Using the C++ API, for a single column, the SmartNIC is 1.4× (256B) to 7.9× (128MiB) slower than the host, with a median and mean across all data sizes of 4.8× and 4.8× slower, respectively. For 16 columns, SmartNIC times range from 1.6× (256B) to 10.2× (8MiB) slower, with a median and mean of 4.6× and 4.3×.

Using the C interface, for a single column, running on

the SmartNIC is between 3.3× (32MiB) and 11.7× (64KiB) than running on the host, with an overall mean and median of 5.6× and 4.9×, respectively. The data from the 16 column experiments show the SmartNIC trailing the host by factors ranging from 3.8× (256B) to 11.4× (8MiB), with a mean and median slowdown of 6.4× and 5.5×, respectively.

*Summary: Blindly offloading the entire Arrow conversion to the SmartNICs causes the SmartNICs to become a throughput bottleneck because it performs this conversion significantly slower than the hosts. Moreover, some cases may exhibit especially severe slowdowns, see e.g., the single column, 128B C ABI offload results in Figure 10*

### F. Using IDP to Increase SmartNIC Throughput of Offloaded Serialization Operations

IDP allows SDMS to place buffer contents from the host directly into data structures in SmartNIC memory[4] (e.g., Arrow IPC buffers, Arrow C ABI data structures). SDMS thus has the potential reduce the work done by the SmartNIC by avoiding having to do that work altogether, mitigating the relative overheads observed above.

The sequence of conversion steps (①-④) described in Figure 6 suggests possible IDP strategies for both the C++ and C approaches. For C++, the *direct-to-arrays* strategy places the contents of C++ STL vectors in host memory directly into Arrow Arrays in SmartNIC memory, subsuming step ① into data movement. In a *direct-to-record-batch* scenario, SDMS initialization includes sufficient information to construct the Arrow Schema (step ②) and allocate space for the RecordBatch columns, so that SDMS can transfer host data directly into the RecordBatch, subsuming steps ① and ③. Finally, in the *direct-to-ipc* scenario, the same SDMS configuration information is used to prepare a destination buffer in SmartNIC memory such that data can be transferred directly to a pre-existing buffer in the Arrow IPC format. The C interface does not include a *direct-to-arrays* option, but rather offers a *direct-to-structs* approach, where SDMS uses IDP to place application data directly into C Array and Schema structures.

Results from our Arrow benchmark enable us to estimate the amount of SmartNIC processing time that can be saved by using these IDP strategies. For example, Figure 9 shows that for one column tables, the minimum SmartNIC processing time saved ranges from 100 $\mu$s (256B) to nearly a second (989.9 ms for 1GiB) using the C++ *direct-to-arrays* strategy; the range is similar for 16 columns. By bypassing the conversion process entirely, the *direct-to-ipc* strategy offers the largest reduction in SmartNIC processing time, ranging from 179 $\mu$s (256B) to 2 seconds (1GiB) for 16 columns. Similarly, Figure 10 shows that for the C ABI, *direct-to-structs* offers very small recoveries to SmartNIC processing time, e.g., from 3 $\mu$s (2KiB) to 22 $\mu$s (128MiB) for 16 columns, while *direct-to-ipc* obtains the largest reductions (e.g., 105 $\mu$s (512) to 993.5 ms (1GiB) for 16 columns).

*Summary: By using IDP to perform serialization in-transit, SDMS can significantly reduce the amount of time the SmartNIC must spend processing host data, reducing the potential of the SmartNIC to become a bottleneck.*

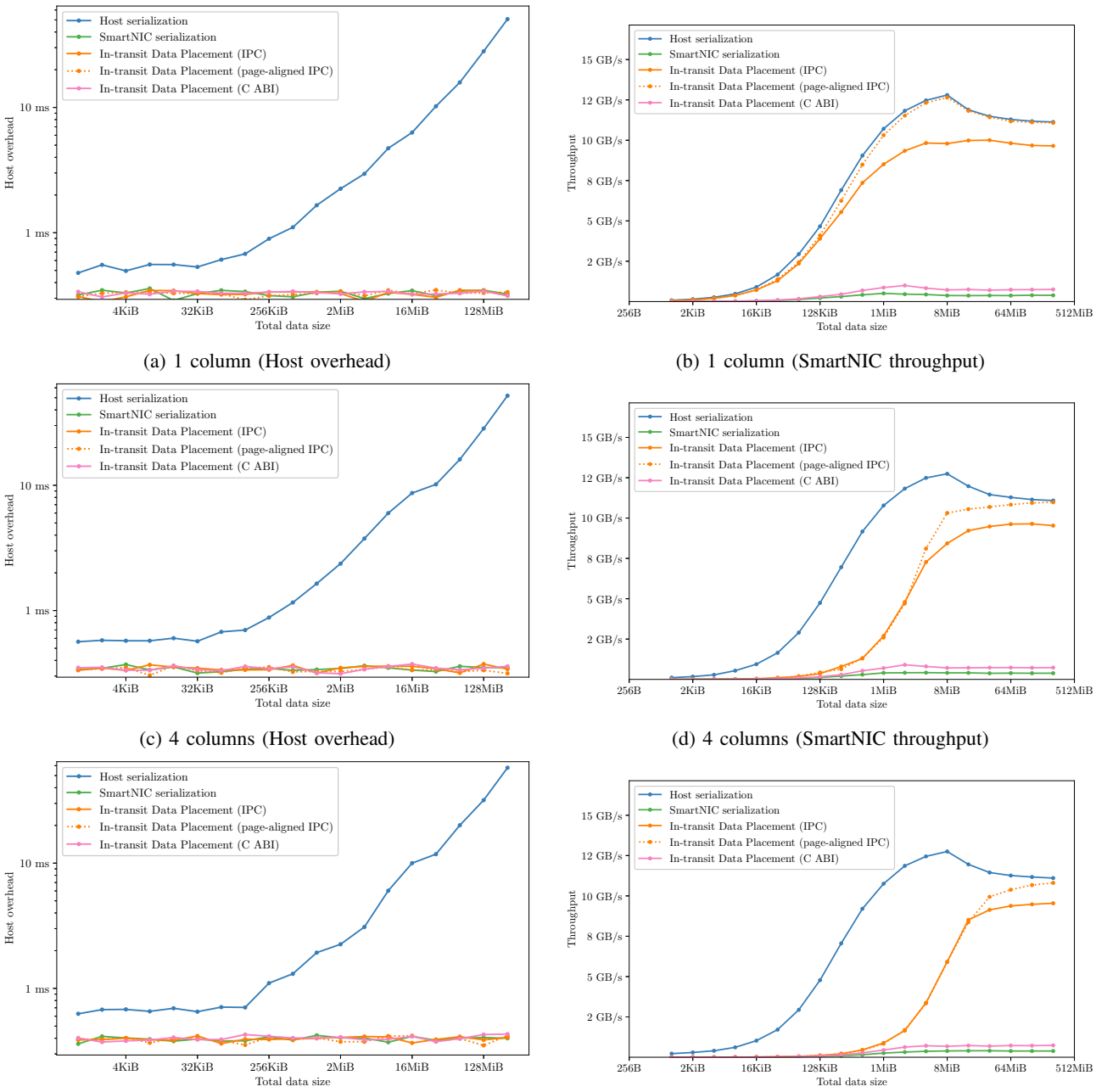
### G. Performance of Offloading Arrow Operations with SDMS

There are several possible approaches to transforming simulation output to Arrow objects and serializing it for transfer to our data analysis tasks. For our Arrow-based case study, we consider the following five approaches to understand the potential benefit of offloading these tasks to the SmartNIC. The experimental setup for these data is described in Section III.

- *Host Serialization*. Application data is transformed to an Arrow object and serialized to an Arrow IPC buffer on the host. This is the baseline approach that represents the case where no computation is offloaded. Because the Arrow objects are serialized on the host, the off-node transfer is limited to a single, contiguous data buffer.
- *SmartNIC Serialization*. Application data is transferred natively from host memory to SmartNIC memory (e.g., from a `std::vector` in host memory to a `std::vector` in SmartNIC memory). After the transfer, the SmartNIC performs the transformation to an Arrow object and the serialization to an Arrow IPC buffer.
- *In-transit Data Placement (IPC)*. Application data is transferred directly from application-native data structures to known locations in existing Arrow IPC buffers in SmartNIC memory (based on the *direct-to-ipc* approach for the C++ API described in Section V-F). The result is that serialization is completed as part of the transfer to the SmartNIC and the data never has to be transformed to an Arrow object.
- *In-transit Data Placement (page-aligned IPC)*. As demonstrated in Section IV, transfer bandwidth is higher for RDMA Get operations when the destination buffer is page-aligned. To characterize the benefit of page-alignment, we padded each column to ensure that the data is page-aligned within the Arrow IPC buffer. Note that at the time of writing Arrow does not include support for page-alignment, and a permanent solution will require modification of the Arrow source.
- *In-transit Data Placement (C ABI)*. As discussed in Section V-C, it is also possible to use Arrow's C ABI [22] to transfer simulation output data by directly modifying pre-existing C data structures (this approach is based on the *direct-to-structs* approach for the C ABI described in Section V-F). This approach to data transfer eliminates the need to transform the data to Arrow data structures. However, the serialization operation needs to be performed after the data arrives in SmartNIC memory.

Figure 11 shows the results of our experiments that measure host and SmartNIC overheads related to Arrow transformation

[4]The SDMS client will allocate the destination data structure before the data transfer. Ideally, either the application will provide information during initialization about the data so that data structures can be allocated during initialization or a suitable data structure could be reused.




(a) 1 column (Host overhead)

(b) 1 column (SmartNIC throughput)

(c) 4 columns (Host overhead)

(d) 4 columns (SmartNIC throughput)

(e) 16 columns (Host overhead)

(f) 16 columns (SmartNIC throughput)

Fig. 11: Median overhead and throughput measurements to characterize the impact of different Arrow serialization strategies

and serialization tasks for each of our five approaches to offloading. Every experiment is repeated ten times. Each row of subfigures corresponds to an Arrow table with a different number of columns. The left column of subfigures shows the host overhead for each experiment. The $x$-axes for these figures show the total size of the data being processed. The $y$-axes of these figures are the median time for the host's involvement in the offload process. These results correspond to our design objective to minimize the host's involvement in offloading the data to the SmartNIC, *see* Section II-B. The data in these three subfigures show that the overhead of transforming the data entirely on the host (i.e., the Host serialization case) is significantly higher than any of the offload approaches (nearly two orders of magnitude larger in some cases). Moreover, the overhead does not increase with data size.

*Summary: Overhead measurements are consistent with our benchmark results Sections IV-A and V-C and demonstrate the overhead of offloading Arrow transformation operations to the SmartNIC is: (i) independent of the total size of the data; and (ii) substantially lower than the overhead of performing the transformation directly on the host.*

In Figure 11, the right column of subfigures characterizes the SmartNIC throughput for for each experiment. The $x$-axes of these figures are the total size of the data. The $y$-axes are the median SmartNIC throughput, calculated as the quotient of the number of data bytes transferred and the amount of time that elapsed from beginning of the transfer from host memory (i.e., when the RDMA Get request is issued) to the time that the data resides in a Arrow IPC buffer in SmartNIC memory. These data correspond to our second design objective to maximize SmartNIC throughput, *see* Section II-B. In Section IV-B, we showed that data transfer by itself would not be a bottleneck. In this section, we examine how the amount of work that is performed on the SmartNIC affects its throughput.

Figure 11b shows the results for a single-column Arrow table. This case establishes a valuable baseline because there is a single contiguous memory buffer to transfer in all cases. The solid blue line shows that host serialization yields the highest throughput. This case minimizes the work required in the SmartNIC because all of the transformation and serialization operations are performed on the host before the transfer is initiated. The solid orange line shows the results for the case where we use IDP to transfer the contents of the Arrow objects on the host to the appropriate locations in an pre-existing Arrow IPC buffer in SmartNIC memory. As discussed above, Arrow is page-alignment agnostic. As a result, the baseline IDP for IPC yields much lower throughput than host serialization. However, the dashed orange line demonstrates that if we artificially pad the Arrow IPC buffers so that transfers land on page boundaries then we can nearly match the performance of host serialization. Specifically, these data show that page-aligned IDP for IPC is able to achieve more than 87% of baseline throughput for data buffers that are 128 KiB or larger (and more than 95% of baseline throughput for buffers that are 1 MiB or larger). At the other end of the spectrum, the pink line shows SmartNIC serialization incurs the greatest overhead because all of Arrow transformation and serialization operations take place on the SmartNIC (*cf.* Section V-E, benchmark results comparing host and SmartNIC performance). We can improve the performance of this approach by using one of our three IDP-based approaches to transfer data directly from host memory to Arrow data structures in SmartNIC memory.

Figures 11d and 11f show the throughput for 4- and 16-column tables, respectively. The most striking difference between these figures and the 1-column results (Figure 11b) is that the results from the two IDP for IPC experiments (the solid and dashed orange lines) are effectively shifted rightward relative to the 1-column results. There are two principal reasons for this: (i) for a fixed data size, tables with more columns require smaller transfers and, as shown in Section IV, the achievable throughput for smaller buffers is generally smaller than for larger buffers; and (ii) each column requires a separate transfer operation; as the number of columns increases, the additional per-transfer overhead further decreases the overall throughput. However, for sufficiently large buffers, using IDP for page-aligned IPC buffers is still able to nearly match the throughput for our host serialization experiments. Although SDMS's IDP feature is not able to match the results of the Host serialization case, it nearly eliminates the work that the host processor has to do to in the Host serialization cast to effectuate the transformation and dramatically improves the SmartNIC throughput relative to naive computation offload (i.e., SmartNIC serialization).

*Summary: Throughput results confirm the intuitions provided by our benchmark results in Section V-F. By using IDP to serialize the data during the transfer we reduce the probability that the offloaded work becomes a bottleneck.*

## VI. Related Work

*Offloading work to SmartNICs*. Several existing research efforts have examined offloading computation to SmartNICs. Liu et al. [23] characterize the performance of BlueField-2 DPUs using benchmarks to identify the kinds of computation that would be candidates for offload. Several frameworks for offloading work to SmartNICs have also been proposed, *see* [24], [25]. Sarkauskas et al. [26] examine how large, non-blocking collectives can be offloaded to BlueField-2 DPUs. Similarly, Bayatpour et al. [5] present a method for offloading non-blocking all-to-all collectives. Karamati et al. [20] show that it is possible to accelerate a molecular dynamics mini-application by offloading parts of the main computation to a BlueField-2 DPU. Gootzen et al. [27] use virtualization to offload of a cloud file-system client to BlueField-2 DPUs.

*Offloading Arrow operations to SmartNICs*. Several research efforts have also examined the specific case of offloading work related to Apache Arrow operations to SmartNICs. Liu et al. [1] examine how BlueField-2 DPUs can be used to compress particle data stored in Apache Arrow objects. In support of offloading Arrow operations to BlueField-2 DPUs, Ulmer et al. [2] used Faodel primitives to transfer the associated data to SmartNIC memory. Because their results showed that the transfer bandwidth was a small fraction of the network bandwidth, they specifically identified transfer bandwidth as an opportunity for improvement. The Fletcher framework enables the acceleration of Apache Arrow operations using FPGAs [28]. Ahmad et al. [29] show that Apache Arrow Flight provides very fast transport of Arrow RecordBatch objects.

*SmartNIC Data Transfer*. NVIDIA's DOCA SDK [30] includes a library for optimizing transfers between host and SmartNIC buffers. This library simplifies development but is more focused on raw performance than offloading application functions and is limited to NVIDIA BlueField devices. Similarly, DPDK [31] enables developers to implement packet

processing pipelines in SmartNICs, but is focused on IP packet flows instead of HPC data streams

In contrast to these existing approaches, we present a high-speed, general-purpose software service that can be used to efficiently offload the transformation of application data to serialized Apache Arrow objects to SmartNICs.

## VII. Conclusion

In this work we introduced SDMS, a high-performance, general-purpose data movement service supporting the offloading of tasks to 'smart' network interfaces. Benchmarking establishes that SDMS provides near-line-rate transfer bandwidths between the host and SmartNIC, and has low communication initiation overheads regardless of buffer size (Section IV). Through an in-depth case study of the use of SDMS to offload Apache Arrow data manipulation to the NIC, we demonstrated that offloading these tasks may significantly reduce host overhead (Section V). Moreover, we demonstrate that the impact to throughput incurred by offloading work to the NIC can be mitigated by using SDMS's IDP feature (Section V-G): For single-column tables, IDP achieves more than 87% of baseline throughput for data buffers that are 128 KiB or larger, and more than 95% of baseline throughput for buffers that are 1 MiB or larger, while also nearly eliminating the host and SmartNIC overhead associated with Arrow operations.

## References


[1] J. Liu, C. Maltzahn, M. L. Curry, and C. D. Ulmer, "Processing Particle Data Flows with SmartNICs," in *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*. IEEE, 2022, pp. 1–8. [Online]. Available: https://doi.org/10.1109/HPEC55821.2022.9926325

[2] C. Ulmer, J. Liu, C. Maltzahn, and M. Curry, "Extending composable data services into SmartNICs," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2023.

[3] J. Yan, L. Tang, J. Li, X. Yang, W. Quan, H. Chen, and Z. Sun, "UniSec: a unified security framework with SmartNIC acceleration in public cloud," in *Proceedings of the ACM Turing Celebration Conference-China*, 2019, pp. 1–6.

[4] R. E. Grant, W. Schonbein, and S. Levy, "RaDD Runtimes: Radical and Different Distributed Runtimes with SmartNICs," in *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, 2020, pp. 17–24.

[5] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda, "BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 18–37.

[6] The Apache Software Foundation, "Apache Arrow: A cross-language development platform for in-memory analytics," https://arrow.apache.org/, undated.

[7] Sandia National Laboratories, "hod-carrier," https://github.com/sandialabs/hod-carrier, undated.

[8] Linux RDMA, "qperf," https://github.com/linux-rdma/qperf, undated.

[9] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, pp. 121–144, 2020.

[10] H. Childs, S. D. Ahern, J. Ahrens, A. C. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier *et al.*, "A terminology for in situ visualization and analysis systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020.

[11] The Apache Software Foundation, "Projects powered by Apache Arrow," https://arrow.apache.org/powered_by/, 2023.

[12] NumFOCUS Inc., "pandas," https://pandas.pydata.org/, 2023.

[13] The Apache Software Foundation, "Apache Spark," https://spark.apache.org/, 2023.

[14] M. Rocklin *et al.*, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, vol. 130. SciPy Austin, TX, 2015, p. 136.

[15] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," 2018.

[16] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest, "Geomesa: a distributed architecture for spatio-temporal fusion," in *Geospatial informatics, fusion, and motion video analytics V*, vol. 9473. SPIE, 2015, pp. 128–140.

[17] J. Liu, C. Maltzahn, and C. Ulmer, "Opportunistic query execution on smartnics for analyzing in-transit data," in *IEEE High Performance Extreme Computing Conference, HPEC 2023*. IEEE, 2023.

[18] C. Ulmer, S. Mukherjee, G. Templet, S. Levy, J. Lofstead, P. Widener, T. Kordenbrock, and M. Lawson, "Faodel: Data management for next-generation application workflows," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, 2018, pp. 1–6.

[19] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 25–36.

[20] S. Karamati, C. Hughes, K. S. Hemmert, R. E. Grant, W. W. Schonbein, S. Levy, T. M. Conte, J. Young, and R. W. Vuduc, "'Smarter' NICs for faster molecular dynamics: a case study," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 583–594.

[21] C. Ulmer, J. Friesen, and J. Kenny, "Glinda: An HPDA cluster with Ampere A100 GPUs and BlueField-2 VPI SmartNICs," Sandia National Lab.(SNL-CA), Livermore, CA (United States), Tech. Rep., 2023.

[22] The Apache Software Foundation, "Apache Arrow: The Arrow C data interface," https://arrow.apache.org/docs/format/CDataInterface.html#c-data-interface, undated.

[23] J. Liu, C. Maltzahn, C. Ulmer, and M. L. Curry. (2021) Performance Characteristics of the BlueField-2 SmartNIC. _eprint: 2105.06619. [Online]. Available: https://arxiv.org/abs/2105.06619

[24] 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*, 2019, pp. 318–333.

[25] G. Vallee, "Mimosa: Multi-tenant intelligent modular offloading service architecture," https://github.com/gvallee/dpu_offload_service, 2023.

[26] N. Sarkauskas, M. Bayatpour, T. Tran, B. Ramesh, H. Subramoni, and D. K. Panda, "Large-message nonblocking MPI_Iallgather and MPI_Ibcast offload via BlueField-2 DPU," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2021, pp. 388–393.

[27] P.-J. Gootzen, J. Pfefferle, R. Stoica, and A. Trivedi, "DPFS: DPU-powered file system virtualization," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, 2023, pp. 1–7.

[28] J. Peltenburg, J. Van Straten, L. Wijtemans, L. Van Leeuwen, Z. Al-Ars, and P. Hofstee, "Fletcher: A framework to efficiently integrate FPGA accelerators with Apache Arrow," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 270–277.

[29] T. Ahmad, "Benchmarking Apache Arrow Flight-a wire-speed protocol for data transfer, querying and microservices," in *Benchmarking in the Data Center: Expanding to the Cloud*, 2022, pp. 1–10.

[30] NVIDIA, "NVIDIA DOCA software framwwork," https://developer.nvidia.com/networking/doca, 2023.

[31] Linux Foundation, "DPDK," https://www.dpdk.org, 2023.