Offloading Data Management Services to SmartNICs: Project Summary

Craig Ulmer, Jianshen Liu, Carlos Maltzahn, Aldrin Montana, Matthew L. Curry, Scott Levy, Whit Schonbein, and John Shawger

ABSTRACT

Modern workflows for high-performance computing (HPC) platforms rely on data management and storage services (DMSSes) to migrate data between simulations, analysis tools, and storage systems. While DMSSes help researchers assemble complex pipelines from disjoint tools, they currently consume resources that ultimately increase the workflow's overall node count. In FY21-23 the DOE ASCR project "Offloading Data Management Services to SmartNICs" explored a new architectural option for addressing this problem: hosting services in programmable network interface cards (SmartNICs). This report summarizes our work in characterizing the NVIDIA BlueField-2 SmartNIC and defining a general environment for hosting services in compute-node SmartNICs that leverages Apache Arrow for data processing and Sandia's Faodel for communication. We discuss five different aspects of SmartNIC use. Performance experiments with Sandia's Glinda cluster indicate that while SmartNIC processors are an order of magnitude slower than servers, they offer an economical and power efficient alternative for hosting services.

Offloading Data Management Services to SmartNICs: Project Summary

Craig Ulmer (PI)
Dept. 08753, Scalable Modeling and Analysis
Sandia National Laboratories
cdulmer@sandia.gov

Jianshen Liu, Carlos Maltzahn, and Aldrin Montana
Center for Research in Open Source Software (CROSS)
UC Santa Cruz
{jliu120, carlosm, akmontan}@ucsc.edu

Matthew L. Curry, Scott Levy, and Whit Schonbein
Dept. 01423, Scalable System Software
Sandia National Laboratories
{mlcurry, sllevy, wwschon}@sandia.gov

John Shawger
University of Wisconsin-Madison
shawgerj@cs.wisc.edu

ACKNOWLEDGEMENT

CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

The *Offloading Data Management Services to SmartNICs* project was a three-year effort (FY21-23) funded by the U.S. Department of Energy's Office of Science to investigate whether emerging programmable network interface cards (or SmartNICs) could improve the way scientific computing workflows execute on high-performance computing (HPC) platforms. The problem we identified in our proposal was that current architectures are hindered by the fact that workflows must allocate additional processing and memory resources from the platform's compute nodes to host data management and storage services (DMSSes) that migrate data between the workflow's applications and insulate users from I/O overheads. Our hypothesis was that the overall efficiency of the system could be improved by adding SmartNICs to a portion of the platform's compute nodes and adapting the services to be offloaded to the SmartNICs. We theorized that this approach would free host resources for other tasks while still preserving data locality benefits. This report documents our work in confirming this hypothesis over the last three years. Topics are organized into three primary themes:

- **Characterizing Current-Generation Hardware:** We conducted a wide range of performance experiments on the BlueField-2 SmartNIC to determine the strengths and weaknesses of current generation hardware. While the embedded processors found on these cards are an order of magnitude slower than host processors, the BlueField-2 SmartNIC features sufficient resources to manipulate in-transit data and perform useful tasks such as hardware-accelerated data compression. An examination of procurement and operation costs of Sandia's Glinda cluster confirmed that the BlueField-2 is power efficient (30W idle, 43W active) and cost effective compared to adding other compute nodes.

- **Creating an Environment for Offloading Services:** We constructed a flexible environment for hosting services on SmartNICs through a combination of Apache Arrow, Sandia's Faodel software, and multiple libraries we developed to simplify interactions with the BlueField-2 (e.g., compression, host-to-card injection, and a dynamic query interface).

- **Offloading Services:** Finally, we constructed multiple examples of data management services to demonstrate that the SmartNIC could offload data management services in a heterogeneous platform. These examples include a service for reorganizing particle datasets; a query service for inspecting in-transit data that automatically determines if a query should be executed at the SmartNIC or the client; and job-local storage services that use NVMe-oF to place storage at the local SmartNIC without disturbing the host.

With the caveat that commercial SmartNICs could benefit from improvements in cost, power utilization, and quality-of-service controls, we assert that our original hypothesis was correct. There are future opportunities in this space to design low-interference, distributed storage systems, create customized disaggregated compute platforms, and leverage new GPU and hardware accelerators that are available in emerging commercial products.

ACRONYMS & DEFINITIONS

Acronym	Definition
ASIC	application-specific integrated circuit
CPU	central processing unit
CSD	computational storage device
DHT	distributed hash table
DMSS	data management and storage service
DPDK	data plane development kit
DPU	data processing unit
FPGA	field-programmable gate array
GGA	generic global accelerators
GPU	graphics processing unit
HCA	host channel adapter
HPC	high-performance computing
HPDA	high-performance data analytics
HPL	high performance linpack
IP	intellectual property
IPC	inter-process communication
NIC	network interface card
NVMe	non-volatile memory express
NVMe-oF	non-volatile memory express over fabrics
PE	processing element
PFS	parallel file system
QoS	quality of service
RADOS	reliable autonomic distributed object store
RDMA	remote direct memory access
RPC	remote procedure call
SDMS	SmartNIC data movement service
SHA	secure hash algorithm
SIMD	single instruction, multiple data
SIMT	single instruction, multiple threads
SmartNIC	a user-programmable network interface card
SoC	system-on-chip
TFT	tag-folding table

1.   INTRODUCTION

The responsibilities of the I/O subsystem in high-performance computing (HPC) platforms have grown significantly over the last decade. In addition to delivering high-bandwidth, high-volume storage for results and checkpoints, the platform is expected to provide a variety of data management and storage services (DMSSes) that have become an essential part of users' workflows. These services include lightweight key/value stores for aggregating state, in-memory object stores for data handoffs between workflow applications, I/O libraries that manage shared state for complex structured data, and programmable storage frameworks that generate live annotations of simulations.

The importance of these DMSSes has driven the scalable I/O community to re-evaluate how services are architected and deployed in modern HPC platforms. Rather than build fixed, system-level services (e.g., burst buffers), many researchers embrace flexible, user-level services that are co-scheduled with simulations. Current research advocates a "composable service" model [1] where a small number of communication components are used to create services that are highly customized to a workflow's requirements. This approach provides users with freedom to specify when and where their DMSSes run on a platform.

A valid criticism of this work is that current architectures lack an optimal location for hosting these services. Researchers have explored hosting services in the simulation's compute nodes [2], supplemental compute nodes [3, 4, 5, 6, 7], the storage system, and "bump-in-the-wire" network hardware. These approaches either steal resources from the simulation, increase network congestion, or are impractical due to security or cost.

In the Offloading Data Management Services to SmartNICs project we have explored an alternative approach: embedding DMSSes in Smart Network Interface Cards (SmartNICs) located in a platform's compute nodes. Emerging SmartNICs such as the NVIDIA BlueField-2 card supplement a traditional NIC with processing and memory resources that are user programmable. Transitioning a composable service library to function on these SmartNICs places services in close proximity to simulations without consuming host resources. Our work seeks to resolve fundamental challenges that arise from this architectural change and evaluate how well DMSSes perform in an environment that mixes compute nodes and SmartNICs.

1.1.   Background

The scalable I/O community has a long history of developing data management and storage services (DMSSes) to improve the way in which large datasets are migrated between different resources in HPC platforms. A significant portion of this work originated in caching services that were built in I/O forwarding nodes to mitigate the high cost of writing simulation results and
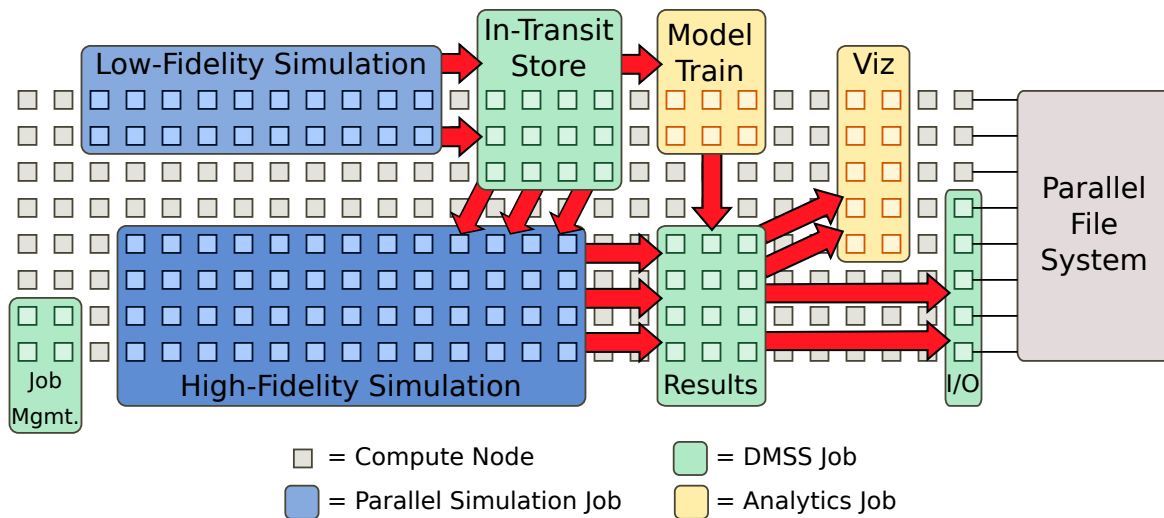
Figure 1-1. Workflow Example: Modern workflows use multiple DMSSes to connect concurrent, parallel applications that are distributed across the compute nodes of an HPC platform. These services allow data to flow between jobs without disrupting internal MPI communication or relying on the file system for handoffs.

checkpoints to disk. These nodes use volatile (and now non-volatile) memory to absorb a simulation's periodic I/O bursts and convert the data flows into a trickle the storage system can sustain [8]. Additional in-transit efforts have examined the benefits of offloading the task of packaging datasets into well-known file containers [9]. In addition to offloading work from compute nodes, there can be efficiency benefits to moving serial I/O software from the data-parallel processors found in compute nodes to general-purpose processors found elsewhere in the platform [10].

Other communities have driven the need for better DMSSes on HPC platforms. Visualization efforts have developed services that allow analysis jobs to be co-scheduled with simulations to permit users to extract information and render visualizations as the simulation progresses [7]. The workflow community needs DMSSes that enable parallel datasets to be routed from one simulation tool to another. High-performance data analytics (HPDA) users are also requesting more access to HPC platforms to solve large, data-intensive problems; these users often work outside of the MPI space and simply need sophisticated DMSSes that can move large volumes of data between different machine-learning and deep-learning tasks. It is not uncommon to find all of these communities working together to solve larger problems. As illustrated in Figure 1-1, an advanced particle simulation workflow at Sandia needed to move data through several, concurrent tools on the same platform. DMSSes routed data between multiple simulations, a visualization framework, a TensorFlow predictor, and a caching service that helped stage data out to disk.

Realizing that each of these communities is implementing similar functionality for different reasons, there have been recent efforts to build DMSSes that can be reused in different contexts. Rather than design one-size-fits-all services, the trend in this research has been to create a collection of communication components that can easily be combined to build new services that are customized to an application's requirements. This "composable service" idea is present in multiple DMSS frameworks such as Mochi [1], BESPOKV [11], and Faodel [12]. In general, these effort include (1) an RDMA portability layer for efficient communication that does not

interfere with MPI, (2) a communication engine for sequencing asynchronous operations, (3) naming services for locating resources, and (4) higher-level APIs such as key/value stores for simplifying data references.

1.1.1. Service Placement

A key question in DMSS research is: *Where should services be placed on an HPC platform to maximize benefits?* In current architectures, there are four common places for hosting services:

1. **Simulation Compute Nodes:** DMSSes occupy resources on a compute node that is also participating in a simulation. In situ approaches generally provide the most performant coupling between simulations and services due to locality. However, in situ services consume resources which would otherwise be available to simulations and can even change performance characteristics of the running simulation (e.g., jitter, scalability). This category typically includes consistency management, peer caches, and on-node burst buffers.

2. **Supplemental Compute Nodes:** DMSSes occupy extra compute nodes outside of the simulation. In vitro services can provide significant capability without perturbing simulations, but they reduce the efficiency and throughput of the HPC system by consuming powerful resources that could otherwise be used for other or larger simulations. This strategy also places a buffer between the simulation and storage, potentially requiring extra network bandwidth. This category often includes key/value stores, databases, and publish/subscribe distributed memory services.

3. **In Storage:** DMSSes execute entirely within the storage system or other servers within the service section of the HPC system. This approach provides bandwidth efficiency, as services have unfettered access to data on the storage system. Compute nodes are no longer withheld from simulation work, yielding increased throughput. However, services provided by the storage system are the most stringently controlled. Security is paramount, as services have more direct access to underlying infrastructure in a privileged portion of the platform; and performance management is necessary to keep the storage system responsive to other users. These considerations typically restrict in-storage services to a core set of system services, and have entirely precluded running any user code.

4. **In Network:** DMSSes execute in "bump-in-the-wire" resources that exist in the network fabric. Active network researchers have proposed supplementing the network fabric with processing elements that can process data as messages move through the communication fabric. When implemented with streaming processors such as FPGAs, this approach is beneficial because there is very little overhead between the network and processor. Unfortunately, the lack of commercial support for this option makes it expensive to realize.

1.1.2. SmartNICs

Recently, network vendors have introduced new Smart Network Interface Cards (SmartNICs) to solve customization problems that arise in commercial data centers and clouds. As illustrated in

Figure 1-2(a), SmartNICs supplement traditional NIC hardware with processor, memory, and storage resources that can be programmed to implement custom functionality at the network's edge. While network security researchers have used SmartNICs to monitor and secure enterprise networks for several years, the primary consumer of SmartNICs today is multi-tenant cloud providers. Both Amazon Web Services and Microsoft Azure [13] employ SmartNICs to securely route messages between a customer's virtual machines. Offloading these operations to the NIC removes extra traversals through the host's facilities and guarantees traffic is always encrypted when it leaves the node. The sheer volume of SmartNIC hardware required by the cloud industry is helping to drive SmartNIC prices to affordable levels.

(a)
(b)
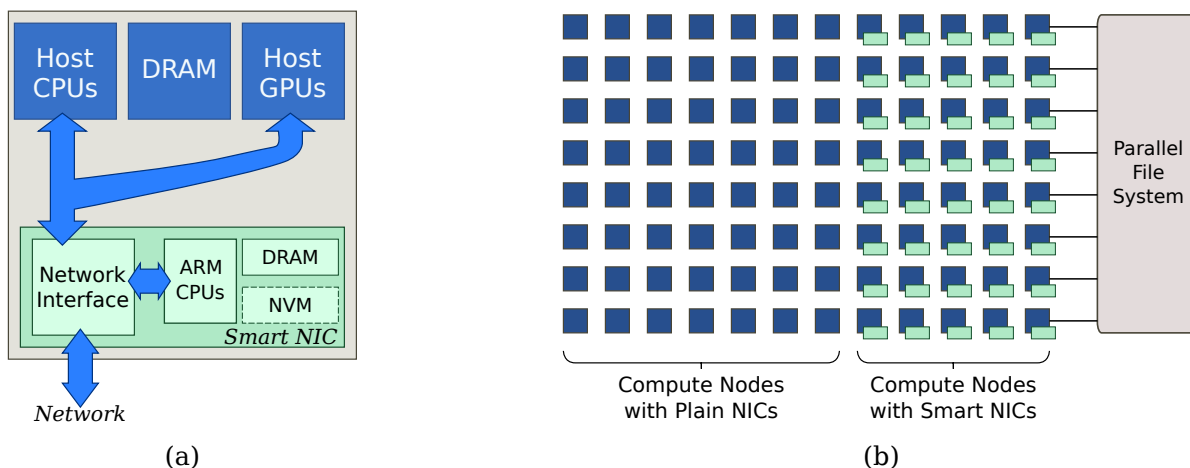
Figure 1-2. SmartNICs: (a) SmartNICs provide on-card resources for processing data close to the network. (b) New platforms will include SmartNICs in a portion of the compute nodes, enabling DMSSes to be embedded at these nodes without host overhead.

NVIDIA's BlueField, BlueField-2, and BlueField-3 product lines demonstrate that vendors can create viable commercial products that embed user-programmable processing resources in high-speed NICs. The current-generation BlueField-2 features eight 64-bit Arm processor cores, 16GB of DRAM, 60GB of eMMC storage, and one to two 100Gb/s network ports that can communicate with InfiniBand or Ethernet. The BlueField-2 SmartNIC's processors run an OS that is independent of the host. While security researchers typically configure the SmartNIC as an in-line, packet-processing engine for manipulating the host's network traffic, the card can be configured to allow the Arms to share the network with the host. As such, the SmartNIC simply appears as just another compute node on the communication fabric. Current BlueField-2 SmartNICs are roughly 80% more expensive than a traditional InfiniBand adapter card. NVIDIA's BlueField-3 features improvements in architecture and is expected to be available in volume by 2024.

1.1.3. *Placing DMSSes in SmartNICs*

The availability of reasonably-priced SmartNICs presents an opportunity for system architects to add a large number of user-programmable processors to the system architecture without needing to increase either the compute node or network switch port counts. As illustrated in Figure 1-2(b),

we envision future platforms will equip a portion of the system's compute nodes with SmartNICs. These SmartNICs effectively become extra communication endpoints in a job allocation that can implement embedded tasks. These resources provide a new option for hosting DMSSes:

5. **At the Network's Edge:** The service is hosted on smaller, user-controlled resources that are at the boundary of each compute node. This fifth paradigm has the potential to blend benefits of on-node, in-compute, and in-storage paradigms while avoiding many of their downsides.

Prior to this project, no DMSS was designed for this type of deployment as the only practical location for executing services was a host's processors. This project has focused on creating a heterogeneous environment where workflows can place services in host processors and SmartNICs.

1.1.4.   Potential Benefits of SmartNICs for Services

From a research perspective, there are multiple ways offloading services to SmartNIC could add value:

- **Increased simulation scalability and efficiency:** Moving portions of libraries and frameworks from the compute node can reduce jitter; moving large buffers with asynchronous tasks to the NIC will increase a code's ability to leverage asynchronous I/O mechanisms. Avoiding in-compute strategies may significantly reduce the energy consumption of such services.

- **Accomplish more with in-transit data:** Much data is discarded due to a lack of storage system ability to store it all, and the simulation's requirement to use its memory and processors to make forward progress. Allowing non-reduced data to be processed by services in the NIC can give simulations a better chance to find interesting phenomena or generate better reductions in parallel with the simulation.

- **Software-defined, hierarchically-managed storage:** Placing data management services in node-local hardware provides developers with freedom to implement application-specific mechanisms for dictating how data is cached and migrated between distributed resources. Programmable storage services will leverage this capability to implement streaming analytics, a capability that is challenging to provide in current HPC software environments.

- **Utilization and resilience opportunities:** Staging data in the SmartNIC provides an opportunity for services to make better scheduling decisions for the network fabric. Similarly, services may offer greater resilience in scenarios where host software crashes but the NIC survives.

1.1.5. *Bridging the Gap*

One of the hardships of high-performance computing research is the heartbreaking gap between *theoretical possibilities* and *realization in commercially-viable hardware*. It is common for new technologies to go through multiple product iterations before the hardware and software is finally mature enough to meet the envisioned goals. As researchers our responsibility is to be both critical and encouraging of new technologies to help elevate their technical readiness. Prior to the start of this project in 2020, multiple research communities had already begun to investigate how SmartNICs could be leveraged for different purposes, including network traffic analysis [14] and services such as distributed key/value stores [15]. However, these efforts were primarily path-finding efforts at small scale. There was little information about how SmartNICs performed in an HPC environment and no empirical data for SmartNICs operating at medium to large scales. We proposed exploring this space to illuminate the characteristics of current SmartNICs and better shape the direction of hardware and software used in upcoming platforms.

1.2. The Offloading Data Management Services to SmartNICs Project

As a means of better evaluating what role SmartNICs could play in hosting data services in future architectures, the DOE's Office of Science funded a three year research project in FY21 named "Offloading Data Management Services to SmartNICs" in the Advanced Scientific Computing Research (ASCR) program. This research project was a collaboration between Sandia National Laboratories and UC Santa Cruz. As illustrated in Figure 1-3 (a) this environment includes (1) communication software to enable interactions between applications and services in local or remote nodes and (2) a data processing layer to standardize data representation and allow users to express data-parallel computations. Once this environment exists, collections of SmartNICs will be able to implement distributed workflows to increase performance.

(a) (b)
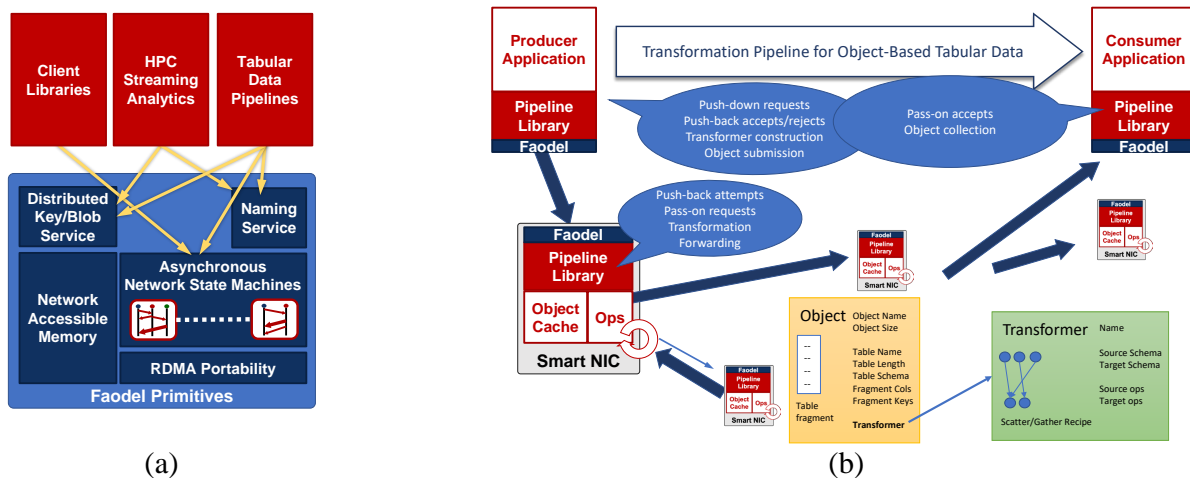
Figure 1-3. Building DMSSes for Smart NICs: (a) Three example services for Smart NICs will leverage different components from Faodel. (b) The tabular data pipelines example will route data through multiple Smart NICs.

1.2.1. Statement of Proposed Research

The proposal for this project stated the following hypothesis:

> "
> Our hypothesis is that adding SmartNICs to a portion of the compute nodes in the platform will improve the efficiency of the overall system because they will provide a way to offload data management and storage services from hosts while preserving locality.
> "

The proposal identified three different research thrusts for the project:

Making SmartNICs Accessible in HPC Platforms: We will address fundamental communication challenges that arise when transitioning to a platform that allows DMSSes to execute on both hosts and SmartNICs. As illustrated in Figure 1-3(a), we will adapt our existing Faodel communication infrastructure to SmartNICs to permit applications to interact with local and remote SmartNICs. This work involves characterizing SmartNIC hardware and adapting current communication components to exploit the hardware capabilities of these devices.

Offloading Operations to SmartNICs: We will investigate three different application scenarios in which we expect there to be a benefit from delegating work to an embedded processor at the edge of the network. These scenarios include (1) offloading asynchronous operations commonly required in DMSS client libraries, (2) performing streaming analytics on HPC data, and (3) constructing a transformation pipeline for object-based tabular data.

Hosting Services in a Mixed Environment: We will explore the tradeoffs of hosting services in a platform where work can be offloaded to SmartNICs and/or supplemental compute nodes. This work will develop a new set of service benchmarks and leverage workflows drawn from production computing to quantitatively answer whether a system with SmartNICs offers an advantage.

1.2.2.   Mapping Research Plan Questions to Answers

The Offloading Data Management Services to SmartNICs proposal provided additional details about each thrust and outlined a research plan for the project based on our knowledge of the hardware in 2020. Table 1-1 summarizes the fundamental research questions the research plan posed and lists a simple answer to each question. Detailed information about the answers can be found in the corresponding chapters in this report.

Table 1-1. Questions from the Research Plan

Question	Simple Answer	Chapter(s)
Making SmartNICs Accessible in HPC Platforms		
How does the BlueField-2's computational performance compare to hosts?	~4-10x slower	3, 6
Does the BlueField-2 offer any compute advantages?	compression	5
Can the SmartNIC's processors saturate a 100Gb/s network?	TCP:no, IB:yes	3
Can existing DMSSes be extended to leverage SmartNICs?	yes (Faodel)	4
Is there a communication advantage for interactions with the local SmartNIC?	slightly (6.9% more bandwidth)	3
Are there other opportunities for exploiting locality?	serialization	8, **D**
Offloading Operations to SmartNICs		
Can SmartNICs be used to insulate hosts from asynchronous I/O?	partially (ephemeral storage)	9, **D**
Can data-parallel processing libraries improve SmartNIC performance?	yes (Arrow, Kokkos)	4
Can data transformation pipelines be mapped to multiple SmartNICs?	yes (particle sifting)	6
What are the benefits of leveraging collections of SmartNICs?	distributed workloads	6
Hosting Services in a Mixed Environment		
How can we compare the cost of compute tasks in a heterogeneous environment?	MBWU, query overheads	3, 7
What impact does offloading have on workflows?	decreased I/O phase	8, 6
What state information can SmartNIC's mine to improve performance?	data content, query complexity	7
Can machine learning techniques help predict when SmartNICs should offload work?	yes (push down/push back)	7
What are the system costs for offloading work to the SmartNIC	power and network penalties	10, 3, 8

D: Deviations from the research plan are discussed in the following section

1.2.3.   Deviations from the Research Plan

We believe the work executed in this three year project is largely faithful to the original research plan listed in the proposal. However, as we learned more about the capabilities of current generation SmartNICs, we realized that a few of the proposed research paths would not be as profitable as other alternatives. As such we adjusted our plans to make better use of our time in this effort. The following are topics where we deviated from the original plan.

Asynchronous I/O Pipelines: One of our original goals was to use the SmartNIC as an engine for offloading asynchronous I/O operations. We envisioned an environment where the SmartNIC would interact with the RADOS object store of a Ceph storage system and make decisions about whether data processing operations should be executed in the SmartNIC or the storage nodes (via SkyhookDM). Unfortunately early experiments with Ceph ran into performance problems where neither a SmartNIC nor a host could achieve more than 20Gb/s of throughput over a 100Gb/s connection. Additional tests found that the BlueField-2 had trouble generating TCP/IP traffic with any target. Given that Ceph's communication is TCP/IP and Ceph's DPDK-based Crimson updates would not be available until the third year of this project, we considered other options for

exploring asynchronous I/O pipelines with SmartNICs and found two related research paths. First, we implemented a push-down/push-back query capability for SmartNICs that demonstrates how SmartNICs can make decisions about whether to execute complex requests locally or defer to other nodes (see Chapter 7). Second, we investigated mechanisms by which the SmartNIC could leverage NVMe-oF to attach remote storage and mitigate ephemeral storage costs in compute nodes (see Chapter 9).

Locality: Multiple research topics in our original proposal were based on an optimistic assumption that current-generation SmartNICs would be tightly coupled with the host node's processors and would offer communication advantages over interactions with other nodes in the system. However, early point-to-point network experiments with the BlueField-2 indicated that there were only slight bandwidth advantages when the host communicated with its local card, and that those situations may be difficult to exploit outside of benchmarks. While there still could be locality benefits when factoring network congestion, the lack of a clear bandwidth advantage precluded the need to extensively refactor the communication stack. Instead, we focused on implementing a host-to-card transfer library that optimizes transfer times, simplifies host library dependencies, and allows us to implement serialization tasks while data is transferred to the local SmartNIC (see Chapter 8).

1.3. Project Contributions

The following are the core contributions made in the Offloading Data Management Services to SmartNICs project.

- **Early BlueField-2 SmartNIC Characterization**: We presented unbiased performance evaluations of the BlueField-2 early in its product cycle and highlighted both the strengths and weaknesses of the hardware.

- **Establishment of 100+ Node SmartNIC System**: Our team worked closely with Sandia's institutional computing team to ensure that the Glinda cluster's SmartNICs were successfully deployed. Glinda is one of the first SmartNIC clusters to have more than 100 nodes. We helped document the stand-up process to help other infrastructure teams work through common problems and make better-informed decisions in their procurements.

- **Adapted Faodel to the BlueField-2**: We adapted the Faodel communication library to the BlueField-2. To the best of our knowledge, this is the first time a composable data service library has been adapted to work in a heterogeneous architecture.

- **Offloading Serialization**: We devised a technique for handling serialization at the same time as data is transferred to the SmartNIC for transmission. This work recognizes that a significant amount of overhead in ejecting data from a compute node involves gathering data from different regions and reorganizing it into a network-transportable object. Our approach defers serialization to the SmartNIC and reduces the number of host copies required.

- **Implemented a Distributed Particle Sifting Service**: We constructed a particle sifting service for HPC workflows that uses a large number of SmartNICs to reorganize particle data streams into a form that is easier for analytics to consume.

- **Demonstrated Apache Arrow's Relevance to HPC**: This project explored the use of Apache Arrow to represent and process scientific data. While Arrow is not currently appropriate for unstructured data, it is competitive with existing scientific computing standards for hosting tabular data and enables researchers to take advantage of a wide array of data science tools.

- **Query Services for In-Transit Data**: We constructed a query service for in-transit data that uses statistics and machine learning techniques to predict whether it will be faster for a SmartNIC to execute a query locally or simply return the raw data.

- **Ephemeral Storage Offload**: We demonstrated that NVMe-oF can be leveraged on SmartNICs to enable in-node storage to be managed without interrupt and compute penalties to the host. This work is a starting point for zero-overhead ephemeral file systems in compute nodes.

1.4.   Project Artifacts

The Offloading Data Management Services to SmartNICs project produced several research artifacts over the course of three years.

Peer-Reviewed Publications

- "Processing Particle Data Flows with SmartNICs" [16]: We presented work from Chapter 5 at the IEEE High Performance Extreme Computing Conference in 2022 and received an Outstanding Student Paper award.

- "Extending Composable Data Services into SmartNICs" [17]: We presented work from Chapter 4 at the Composable Systems Workshop at IPDPS and received a Best Paper award.

- "Opportunistic Query Execution on SmartNICs for Analyzing In-Transit Data" [18]: We presented query work from Chapter 7 at the IEEE High Performance Extreme Computing Conference in 2023.

Ph.D. Dissertations

- *Extending Composable Data Services to the Realm of Embedded Systems* [19]: Jianshen Liu successfully completed his Ph.D. defense at UC Santa Cruz in June 2023.

Technical Reports

- "Performance Characteristics of the BlueField-2 SmartNIC" [20]: This arXiv report measured performance details for the BlueField-2 SmartNICs and is the basis for Chapter 3 [1].

- "Glinda: An HPDA Cluster with Ampere A100 GPUs and BlueField-2 VPI SmartNICs"[21]: This Sandia technical report describes the construction of Glinda and is summarized in Chapter 2.

- "Offloading Node-Local Filesystems in High Performance Computing Environments" [22]: John Shawger documented the work presented in Chapter 9 in Sandia's 2023 CSRI Summer Proceedings.

Poster Presentations

- Aldrin Montana presented "Decomposing Queries for Composable Data Services" at the Monterey Data Conference in 2023.

Community Engagement

- Craig Ulmer and Matthew Curry served on the Supercomputing SmartNICs Panel Sessions in 2021 and 2022.
- Craig Ulmer and Matthew Curry served on the SmartNIC Summit Organizing Committee in 2022.
- Carlos Maltzahn organized the Computational I/O Stack Workshop at UC Santa Cruz in 2023.
- Craig Ulmer gave a guest lecture on SmartNICs to an undergraduate class at Slippery Rock University in 2023 as part of the DOE ASCR Computational Research Leadership (CRLC) Seminar Series.

Open Source Software

- `Bitar`[2]: We released a new library to simplify accessing (de-)compression accelerators.
- `Faodel`[3]: Sandia's Faodel library was extended with support for Apache Arrow data.
- `hod-carrier`[4] : We released a low-level library built on InfiniBand Verbs for providing high-performance data transfer between host and SmartNIC. This library forms the basis of the SmartNIC Data Movement Service described and evaluated in Chapter 8.

---

[1] https://www.nextplatform.com/2021/05/24/testing-the-limits-of-the-bluefield-2-smartnic/
[2] https://github.com/ljishen/bitar
[3] https://github.com/sandialabs/faodel
[4] https://github.com/sandialabs/hod-carrier

1.5. Report Organization

This report provides a summary of the different contributions made in this project and is organized as follows.

- **SmartNIC-Equipped Computing Platforms**: Chapter 2 describes the architectural features of both the BlueField-2 SmartNIC and the computing platforms that were used in this research project.

- **Performance Characteristics of the BlueField-2 SmartNIC**: Chapter 3 then explores different benchmarks we conducted on the BlueField-2 to quantify its strengths and compare it to other computing platforms. This work indicates that developers should expect the Arm processors to be an order of magnitude slower than host processors.

- **An Environment for Hosting Data Services in SmartNICs**: Chapter 4 explores the means by which we can create a reusable environment for hosting DMSSes in SmartNICs through libraries such as Faodel and Apache Arrow.

- We then explore five different aspects of offloading services to SmartNICs:

    - **Leveraging Compression Hardware**: Chapter 5 examines the compression hardware accelerator found in the BlueField-2 and quantifies the impact this hardware has when processing in-transit data using our `Bitar` library.

    - **Reorganizing Distributed Datasets**: Chapter 6 details a case study where a collection of distributed SmartNICs work together to reorganize particle simulation data into a form that is easier for external analytics to consume.

    - **Querying In-transit Data**: Chapter 7 explores query interfaces into SmartNICs to allow users to inspect in-transit data. This work includes a prediction engine that determines whether it would be quicker for the SmartNIC to execute the query ("push-down") or simply return the raw data back to the client ("push-back").

    - **Optimizing Host-to-Card Data Transfers**: Chapter 8 addresses performance concerns of moving data between the host and its local SmartNIC and evaluates our SmartNIC Data Movement Service for simplifying high-performance data transfers between host and SmartNIC memory.

    - **Storage Services**: Chapter 9 investigates options for attaching a node's NVMe storage to the SmartNIC via NVMe-over-Fabric in preparation for future work that will offload ephemeral storage to compute nodes.

- **Costs**: Chapter 10 investigates the costs associated with procuring and operating a platform with SmartNICs. This work estimates the BlueField-2 SmartNIC's power use and discusses the ramifications of operational cost on broader SmartNIC adoption.

- **Conclusion**: Chapter 11 provides our closing thoughts on the current state of SmartNICs and lists opportunities for future work.

2.   SMARTNIC-EQUIPPED COMPUTING PLATFORMS

The availability of low-cost Arm and RISC-V IP cores has motivated several hardware vendors to include reprogrammable resources in hardware products that have traditionally only offered fixed functionality. For example, storage vendors sell computational storage devices (CSDs) that allow users to perform data transformations at the disk to assist in deduplication and error handling [23]. Similarly, network vendors are including user-programmable processing resources in their network cards and switches to offload collective communication operations [24], enhance security [25, 26], and present disaggregated storage to the host [27]. While the embedded processors in these devices may be an order of magnitude slower than host processors, vendors have demonstrated that there is great value in placing small pieces of embedded software in hardware devices that are distributed throughout a computing platform [28].

(a)   (b)
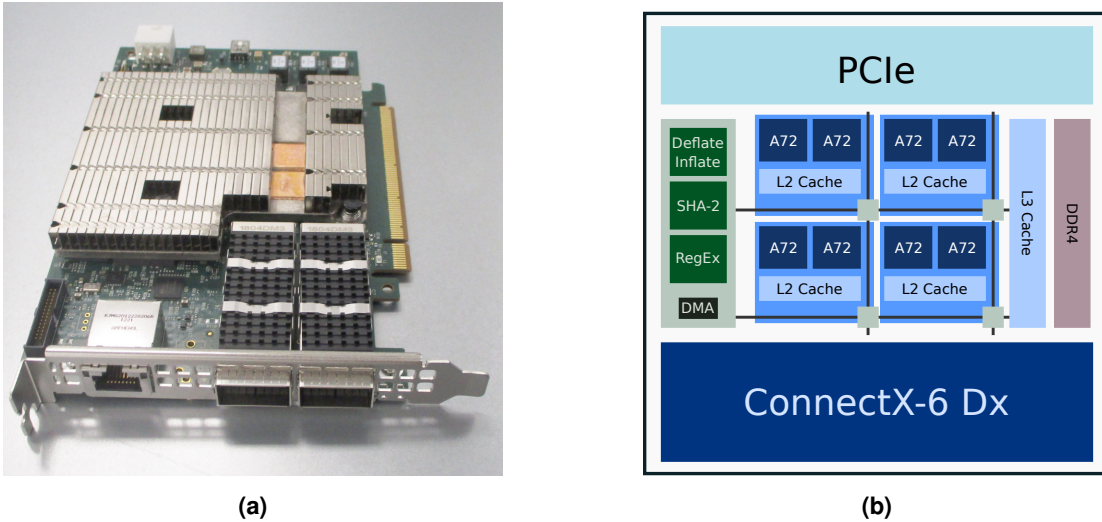
Figure 2-1. The (a) BlueField-2 DPU SmartNIC and its (b) SoC Architecture

In the Offloading Data Management Services to SmartNICs project we focus on leveraging programmable network interface cards (SmartNICs) to host services that are useful for HPC workflows. In this chapter we provide background information about the BlueField-2 SmartNIC that was selected for this project and answer fundamental questions about its operating characteristics based on our experiences with the hardware. We then describe two SmartNIC-equipped platforms where we conducted our experiments in this project. First, CloudLab is an NSF-funded cloud that provides university researchers with access to a wide range of computing technologies. Second, Glinda [21] is a high-performance data analytics (HPDA) cluster at Sandia that was deployed in the second year of this project.

2.1.   The BlueField-2 SmartNIC

The Offloading Data Management Services to SmartNICs project selected the NVIDIA
BlueField-2 DPU (data processing unit) to serve as the SmartNIC for this work because it
supplements a traditional HPC InfiniBand network interface with multiple, user-programmable
Arm processor cores. As illustrated in Figure 2-1b the BlueField-2 System-on-Chip (SoC) ASIC
contains a ConnectX-6 network interface, a PCIe switch, Arm processors, and hardware
accelerators. This section describes how the BlueField SmartNIC architecture evolved over time
and answers basic questions about how the card operates.

2.1.1.   SmartNIC Evolution at Mellanox/NVIDIA

Mellanox Technologies, Ltd. had a long history of developing high-performance network
products before they were acquired by NVIDIA in 2019. Mellanox initially achieved success in
the early 2000's by producing low-cost switches and network interface cards (NICs) for HPC that
implemented the InfiniBand network standard. In 2007 they expanded their customer base by
developing a NIC with a ConnectX ASIC that could communicate with either InfiniBand or
10Gb/s Ethernet. While the ConnectX family of ASICs has never been user programmable,
Mellanox has constructed specialty NICs such as the Innova Flex that feature FPGA resources
that security researchers can leverage to monitor and manipulate network traffic. In response to
requests by commercial cloud vendors to provide a programmable NIC that could help secure
cloud infrastructure, Mellanox developed the BlueField SmartNIC product line. The original
BlueField SmartNIC supplemented a ConnectX-4 network chip with 8-16 Arm processor cores
and 16GB of DDR DRAM. While the Arm processors had limited performance due to power and
thermal constraints, multiple researchers demonstrated that useful work can be offloaded into the
SmartNICs [29, 30].

The BlueField-2 SmartNIC was announced in 2019 and made available in late 2020. While the
BlueField-2 employs only half the processor cores of its predecessor, the cores run at three times
the clock rate (2.75GHz vs 800MHz). The BlueField-2 cards also include custom hardware to
accelerate cryptography, compression, and regular expression operations. NVIDIA offers
multiple variations of the card with different network speeds (25Gb/s-200Gb/s), network fabrics
(Ethernet or Ethernet/InfiniBand), and processor speeds (2.0GHz-2.75GHz). NVIDIA also
produces the BlueField-2X converged card, which combines a BlueField-2 DPU and an Ampere
A100 GPU into a single PCIe card. The converged card may be desirable in situations where
several GPUs are attached to a system through InfiniBand and users simply need a minimal
hardware path for accessing remote GPU resources.

2.1.2.   BlueField-2 Operating Modes

The Arm processors on the BlueField-2 SmartNIC have dedicated memory and boot an operating system that is independent of the host. As illustrated in Figure 2-2, the BlueField-2 can be configured to boot into one of two modes:

- **Embedded Function Mode** (default): In Embedded Function Mode traffic between the host and the network is routed through software that runs on the Arm processor. Packet processing software and applications such as Open vSwitch can be used to inspect, manipulate, and route packets on behalf of the host. This mode is commonly used in network security applications where the SmartNIC serves as an embedded hypervisor for protecting the host system.

- **Separated Host Mode**: The BlueField-2 SmartNIC can alternatively be configured to run with the Arms functioning as an independent host that shares the node's network connection. In this mode, the host exchanges data directly with the ConnectX network interface. The host may communicate with the Arm processors through traditional network operations, such as sockets or RDMAs.
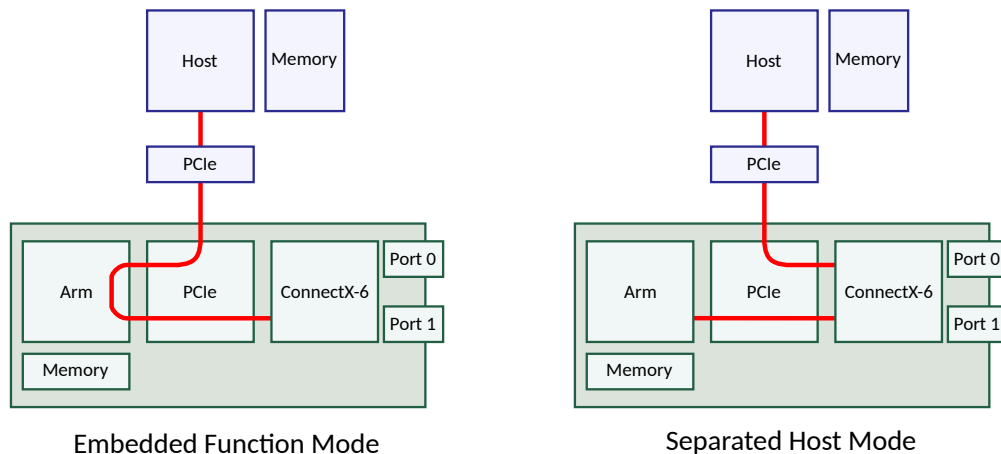
Figure 2-2. BlueField-2 Operating Modes

These two modes are commonly referred to as *on-path* and *off-path* processing of network data in the literature. While on-path processing is desirable in streaming situations (e.g., packet inspection), it can be challenging to implement higher-level functionality that involves complex APIs or data that is larger than a self-contained message. Research efforts such as INCA [31] seek to remedy these issues, but require additional support. Our experience with the BlueField-2 has been that InfiniBand does not work properly when Embedded Function Mode is enabled. As such the BlueField-2 cards used in our work are configured to operate in Separated Host Mode except where noted in this report.

2.1.3.   *Software Development and the NVIDIA DOCA SDK*

Once a BlueField-2 DPU is properly installed in a system, developers can log into it and use it in a manner that is very similar to other hosts in the platform. The Ubuntu OS on the card includes a `gcc` 9.4.0 compiler and build tools, and can be easily updated with other tools from Ubuntu's repositories. We have successfully built a full suite of additional tools and libraries from source using LLNL's Spack[1]. Host applications do not require special libraries to interact with the BlueField-2 other than libraries used for traditional network operations (e.g., TCP/IP sockets or ibverbs). The Data Plane Development Kit (DPDK[2]) does include device-specific support for the BlueField-2 that streamlines IP communication with the card and enables users to take advantage of the compression accelerator.

NVIDIA provides the DOCA SDK as a means of simplifying development costs when constructing applications that leverage the BlueField-2's capabilities. DOCA is a collection of host and card libraries that target a mix of different use cases. While a central part of DOCA focuses on creating a trusted environment for offloading security operations, the SDK includes libraries that help optimize host/card data transfers and simplify access to the card's accelerators. However, users should carefully *review the DOCA EULA*[3] before committing to DOCA. Item 4(c) of the current EULA has the following restriction:

```
You may not disclose the results of benchmarking, competitive analysis, regression
or performance data relating to the SOFTWARE without the prior written permission
from NVIDIA Mellanox.
```

---

[1]`https://spack.io`
[2]`https://www.dpdk.org/`
[3]`https://docs.nvidia.com/doca/sdk/eula/index.html`

2.1.4. General Questions and Answers

One of the challenges of being an early adopter of BlueField-2 equipment was that there was a shortage of general information about the hardware. Table 2-1 provides a short list of questions and answers about BlueField-2 hardware, based on our experiences.

Table 2-1. General Questions about the BlueField-2

Question	Simple Answer
Software	
Will software compiled on hosts with A72 Arms run on the BlueField-2?	yes
Is the BlueField-2's compression accelerator compatible with DEFLATE software?	yes
Can the compression hardware work with streaming data?	yes
Can the host access the compression hardware?	yes
Can the Arm processor PXE boot its OS?	yes
Resilience	
Is the BlueField-2's OS disrupted if the host reboots?	no
Is the host's OS disrupted if the BlueField-2 reboots?	no
Does the BlueField-2 lose network access while the host reboots?	no
Networking	
Does host-to-card InfiniBand work if the port is physically unplugged?	no
Are there (documented) mechanisms for the Arm to sense the host's traffic?	no
Does the host's network traffic get priority over the Arm's?	no
Can the BlueField-2 VPI use InfiniBand on one port and Ethernet on the other?	yes
Will the host and Arm both be visible if the SM does not support 2 LIDs/port?	no
Does InfiniBand work properly when the card is in Embedded Compute Mode?	no
Is the out-of-band network port required for the card to function?	no
PCIe	
Does the Arm list the host's PCIe cards in `lspci`?	no
Can the Arm access the host's NVMe devices through NVMe-over-Fabric?	yes
Is an external cable necessary for console access or firmware updates?	no
Power	
Does the BlueField-2 have a documented, on-card power monitor?	no
Does the BlueField-2 really need the power/cooling listed in the specifications?	yes
What is the stated maximum power of the BlueField-2?	64W
What is the idle power of the BlueField-2?	30W
Export Control Restrictions	
Are there export control restrictions with the encryption hardware?	yes
May foreign nationals at the labs use cards with encryption hardware?	yes

1. We have not tested Embedded Compute Mode extensively, but in our firmware versions the InfiniBand port was missing or nonfunctional.
2. Cards with encryption hardware enabled are subject to export controls (Export Control Classification Number 5A002.a.2).
3. NVIDIA's export regulation compliance page: https://www.nvidia.com/en-eu/networking/export-regulations/

2.2. NSF's CloudLab

One of the challenges of conducting systems research is that computing infrastructure is expensive to procure and maintain. While the national laboratories have significant computing resources available, lab policies limit how these resources are used to ensure that tax payer money is not spent improperly. Recognizing that infrastructure costs were impeding scientific discovery and collaboration in universities, the National Science Foundation established CloudLab in 2014 as a distributed platform where university researchers could conduct various cloud experiments [32]. CloudLab hardware consists of 25,000 compute cores hosted at three locations: the University of Wisconsin, Clemson University, and the University of Utah. After obtaining an account through a university affiliation, a researcher may provision resources in the cloud and control how both the systems and the network infrastructure are configured.

Given that CloudLab was an early recipient of BlueField-2 SmartNIC hardware, the UCSC members of this project requested access and conducted a variety of experiments that influenced the overall direction of this project. The initial BlueField-2 systems in CloudLab were connected to a 100Gb/s Ethernet network in the cloud.

2.2.1. CloudLab Systems

One of the benefits of CloudLab is that it hosts a wide variety of architectures from different time periods. Table 2-2 lists the architecture differences of different systems that we used for comparisons in Chapter 3.

Table 2-2. Specifications of Stress-ng Test Platforms

Date	Platform	CPU	Clock	Cores	DRAM	Memory Type	Network
2009 Q1	d710	Intel Xeon E5530	2.4 GHz	1 x 4	12GB	DDR3-1066	4 x 1Gb
2012 Q2	r320	Xeon E5-2450	2.1 GHz	1 x 8	16GB	DDR3-1600	2 x 1Gb
2012 Q4	m400	Armv8 Atlas/A57	2.4 GHz	1 x 8	64GB	DDR3-1600	2 x 10Gb
2013 Q3	c6220	Xeon E5-2650 v2	2.6 GHz	2 x 8	64GB	DDR3-1866	2 x 10Gb, 4 x 1Gb
2013 Q3	c8220	Intel E5-2660 v2	2.2 GHz	2 x 10	256GB	DDR3-1600	2 x 10Gb, 1 x 40Gb IB
2014 Q3	c220g1	Intel E5-2630 v3	2.4 GHz	2 x 8	128GB	DDR4-1866	2 x 10Gb, 1 x 1Gb
2014 Q3	c220g2	Intel E5-2660 v3	2.6 GHz	2 x 10	160GB	DDR4-2133	2 x 10Gb, 1 x 1Gb
2014 Q3	d430	Intel E5-2630 v3	2.4 GHz	2 x 8	64GB	DDR4-2133	2 x 10Gb, 2 x 1Gb
2014 Q3	dss7500	Intel E5-2620 v3	2.4 GHz	2 x 6	128GB	DDR4-2133	2 x 10Gb
2015 Q4	m510	Intel Xeon D-1548	2.0 GHz	1 x 8	64GB	DDR4-2133	2 x 10Gb
2016 Q1	xl170	Intel E5-2640 v4	2.4 GHz	1 x 10	64GB	DDR4-2400	4 x 25Gb
2017 Q3	c220g5	Intel Xeon Silver 4114	2.2 GHz	2 x 10	192GB	DDR4-2666	2 x 10Gb, 1 x 1Gb
2021 Q2	BlueField-2	Armv8 A72	2.5 GHz	1 x 8	16GB	DDR4-1600	2 x 100 Gb/s

1. The BlueField-2 card is part number MBF2H516A-CENO_Ax
2. All platforms except the BlueField-2 ran Ubuntu 20.04 (kernel 5.4.0-51-generic).

2.3.   Sandia's Glinda Cluster

In FY21 the Mission Computing Council at Sandia National Laboratories approved a request to procure a new high-performance data analytics (HPDA) platform to support data-intensive workloads. The architectural goal of this system was to create a sizable pool of single-processor, single-GPU compute nodes that data scientists could use to prototype machine learning and deep learning algorithms before scaling up to larger, many-GPU DGX systems that exist at Sandia. At the request of the researchers in this ASCR project, the specifications for this procurement required that the high-speed network interface cards for this system be implemented with BlueField-2 SmartNICs. The intent of this request was to create a sizable SmartNIC testbed within a cluster so that network researchers could better evaluate what role SmartNICs should play in data-intensive workflows.

The procurement process successfully resulted in a 126-node system named *Glinda* that was integrated into an existing HPDA platform named Kahuna. As documented in the Glinda stand-up report [21], multiple factors hindered the deployment, including COVID supply chain issues, motherboard PCIe timing issues with the BlueField-2, and power stability issues within the compute nodes. Fortunately, all issues were resolved and the system was made available to researchers in early 2022. As pictured in Figure 2-3 Glinda is comprised of seven racks situated in the SNL/CA's B902 data center.

Figure 2-3. Sandia's Glinda HPDA Platform

2.3.1. Glinda Compute Node

A Glinda compute node is an Atipa Procyon SE218-8G4 2U server. This server includes four bays for hosting GPU accelerator cards and uses the Gigabyte G242-Z11-00 motherboard. As illustrated in Figure 2-4, a compute node is composed of a single-socket EPYC processor, 512GB of memory, an NVMe disk, a BlueField-2 DPU, and an NVIDIA Ampere A100 GPU. Fans in the middle of the chassis pull cool air from the system's front into the GPU bays and then route it through the CPU, memory, and SmartNIC before venting the heat out the back. Power from the front-middle GPU bay is routed to the BlueField-2 card through a PCIe power extension cable. While the initial deployment exhibited timing problems during the boot process, targeted firmware updates from Gigabyte and NVIDIA resolved PCIe issues and resulted in a stable deployment.
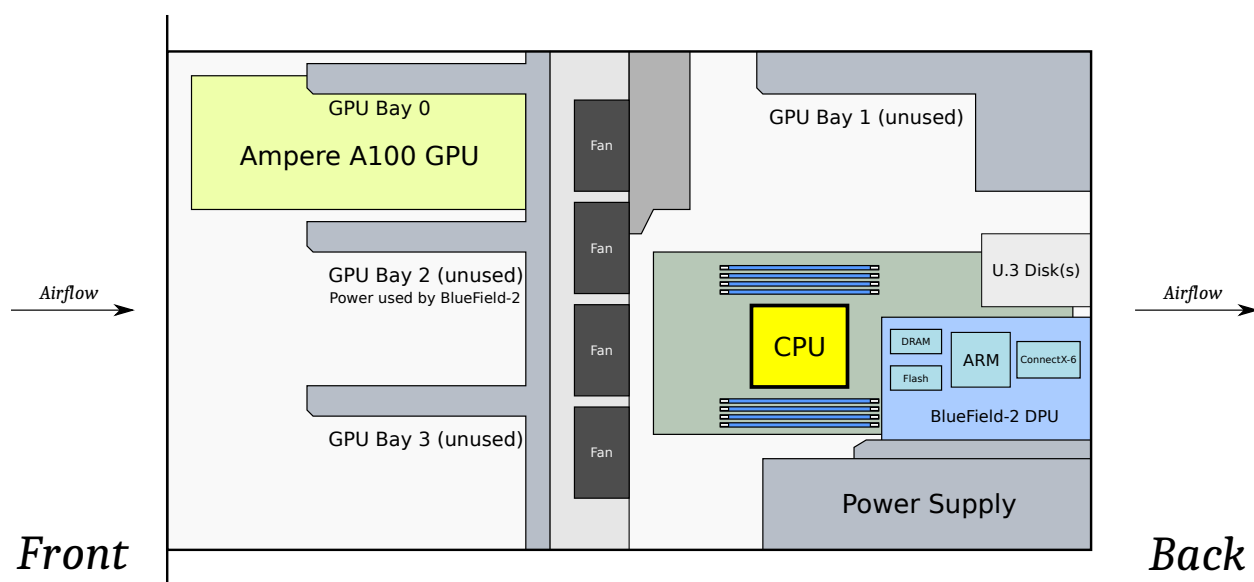
Figure 2-4. Overhead view of a Glinda compute node

Singra Testbed: A 16-node test cluster named *Singra* was also established at Sandia to aid in development. Singra nodes are similar to Glinda nodes, with the exception that the Singra motherboard features two AMD EPYC 7513 processors (2x32 Zen3 cores at 2.6GHz) instead of one 7543P.

2.3.2. Processor Comparison

From a processing perspective, the Glinda compute node is interesting because it is a heterogeneous architecture with three components: a traditional x86_64 multicore host processor, an embedded Arm multicore processor in the SmartNIC, and a single-instruction, multiple-threads (SIMT) GPU for data-parallel operations. Architectural details for these processors are summarized in Table 2-3.

Table 2-3. Glinda Processor Details

Feature	SmartNIC	Host CPU	GPU
Processor	Arm A72	EPYC 7543P	Ampere GA100
Physical Cores	8	32	108 SMs
Total Threads	8	64	-
Base Clock	2.75GHz	2.8GHz	765MHz
Max Boost Clock	-	3.7GHz	1.41GHz
L1 I-Cache	384KB	1MB	-
L1 D-Cache	256KB	1MB	192KB/SM
L2 Cache	4MB	16MB	40MB
L3 Cache	6MB	256MB	-
Memory Capacity	16GB	512GB	40GB
Memory Channels	1	8	-
Memory Type	DDR4-3200	DDR4-3200	HBM2
Memory Bandwidth	25GB/s	204GB/s	1,555GB/s
PCIe	PCIe 4.0 x16	PCIe 4.0 x128	PCIe 4.0 x16
TDP	64W	225W	250W

1. NVIDIA uses an L1 I-cache in the A100 but does not disclose its size.
2. The A100's 108 Streaming Multiprocessors (SMs) house a total of 6,912 FP32 CUDA Cores

From these numbers we see that while the clock speeds and memory interfaces are similar between the SmartNIC DPU and the host processor, the host features more advanced processor cores, more (4x) cores, more (40x) data cache, and substantially larger (8x) and faster (8x) memory. In the same manner, the GPU has orders of magnitude more compute cores and much more (7x) memory bandwidth than the host CPU. As such, we expect to see an order of magnitude difference or more between each of these technologies.

2.4.   Summary

The NVIDIA BlueField-2 SmartNIC is an HPC NIC that includes user-programmable processor cores. While the SmartNIC can be configured as an in-line packet processing engine for Ethernet networks, we focus on the separated host mode that treats the SmartNIC's Arms as another compute node connected to the network fabric. Users can execute software on these processors in a manner that is similar to any other Linux node in the platform. This project has benefited from two BlueField-2 platforms. First, nodes from NSF's CloudLab system enabled us to conduct early-access experiments. Second, Sandia's Glinda platform provided a sizable system for experiments involving over 100 SmartNICs.

3.   PERFORMANCE CHARACTERISTICS OF THE BLUEFIELD-2

The first step in evaluating what role SmartNICs can play in offloading data management and storage services is understanding the strengths and weaknesses of current-generation hardware. For this work we focus on the NVIDIA BlueField-2 SmartNIC because it features user-programmable Arm processors and supports both Ethernet and InfiniBand networking. In this chapter we explore four different aspects of the BlueField-2's performance characteristics. First, we measure the computational performance of the Arm processors with a collection of microbenchmarks. These experiments help us understand how well the Arm processors perform different classes of operations and allow us to compare the SmartNIC to different servers. Second, we examine the processing headroom that is available when the SmartNIC is busy transmitting large amounts of data to a 100Gb/s Ethernet network. These experiments demonstrate how challenging it is for embedded processors to saturate high-speed Ethernet, but also confirm that the BlueField-2 has sufficient resources for manipulating data at these speeds. Third, we measure InfiniBand performance when different combinations of host and SmartNIC CPUs communicate. These tests highlight the advantages of using InfiniBand instead of Ethernet and indicate that there is a slight advantage for communication between a host and the Arm processors of the local SmartNIC. Finally, we briefly describe some of the characteristics of the BlueField-2's computational accelerators. The compression unit is examined in greater detail in Chapter 5.

3.1.   Computational Assessments

The goal of our first performance evaluation is to characterize the *computational* strengths and weaknesses of the BlueField-2 hardware. This characterization is important because it helps us determine scenarios where it may be profitable to offload computations to SmartNICs that are in a workflow's data path. It is often difficult to determine which hardware components in an embedded device will have the most influence on performance without a great deal of experimentation. Therefore this evaluation focuses on running a wide range of microbenchmarks that help illuminate many different aspects of the hardware's capabilities.

The `stress-ng` [33] tool was selected for this evaluation because it contains a large number diverse thrashing functions (called "stressors") that are designed to stress different software and hardware components of a system. In contrast to integrated tests, each `stress-ng` stressor repeats a specific operation continuously for a fixed period of time. For example, the msync stressor tests the msync(2) system call while the CPU stressor tests the CPU's floating-point, integer, and bit manipulation performance separately, each with a different function. `stress-ng` contains a total of 250 stressors that cover a wide range of resource and function domains, including disk I/O, network I/O, DRAM, filesystem, system calls, CPU operations, and CPU cache. Inside `stress-ng` these domains are used as classes to categorize stressors. Our evaluation collected

performance measurements for several systems and analyzes differences at both the individual stressor level as well as the broader class levels.

One challenge in comparing the performance of different stressors is that results are reported in terms of "bogus operations per second" or bogo-ops-per-second. Each stressor simply counts the number of times a particular operation can be performed in a given amount of time. While this metric provides a means of comparing how well *different systems* perform the same task, it is meaningless to directly compare the bogo-ops-per-second numbers of *different stressors*.

3.1.1. *Experiment Setup*

The `stress-ng` test was run on the BlueField-2 SmartNIC as well as a variety of host systems available in CloudLab to better understand the computational capabilities of the card. As listed in Table 2-2 the 12 host systems all used Intel x86_64 processors with the exception of the m400 system, which is based on the older Armv8 A57. While it may not be fair to compare the BlueField-2 card's embedded processor to general-purpose server processors, all of the servers were at least three years old. The d710 is based on a 12-year old processor.

In addition to the CloudLab systems, we conducted the `stress-ng` experiments on a stock Raspberry Pi 4B (RPi4) system with 4GB of RAM. While the RPi4 is not a particularly fast embedded system, it is ubiquitous and serves as a reference hardware platform that others may use for comparing results. Performance numbers reported in this section are normalized to the RPi4.

On each platform, we sequentially ran all default stressors (a total of 218^1), each of which ran for 60 seconds. Each stressor launches one instance on each online CPU core. We repeated this execution on a platform five times and averaged the results. To calculate the relative performance of a stressor for a platform, we divide the mean bogo-ops-per-second value for the platform by the corresponding value obtained for the RPi4. Some stressors did not run on some platforms because of a lack of support for a required capability. For example, the rdrand stressor did not run on the BlueField-2 SmartNIC as well as on the m400 platform because the Arm CPU does not support the rdrand instruction. The final results from all platforms are plotted in Figure 3-1.

---

[1]Some stressors were not executed on any platform because we did not use root privileges when running the tests.
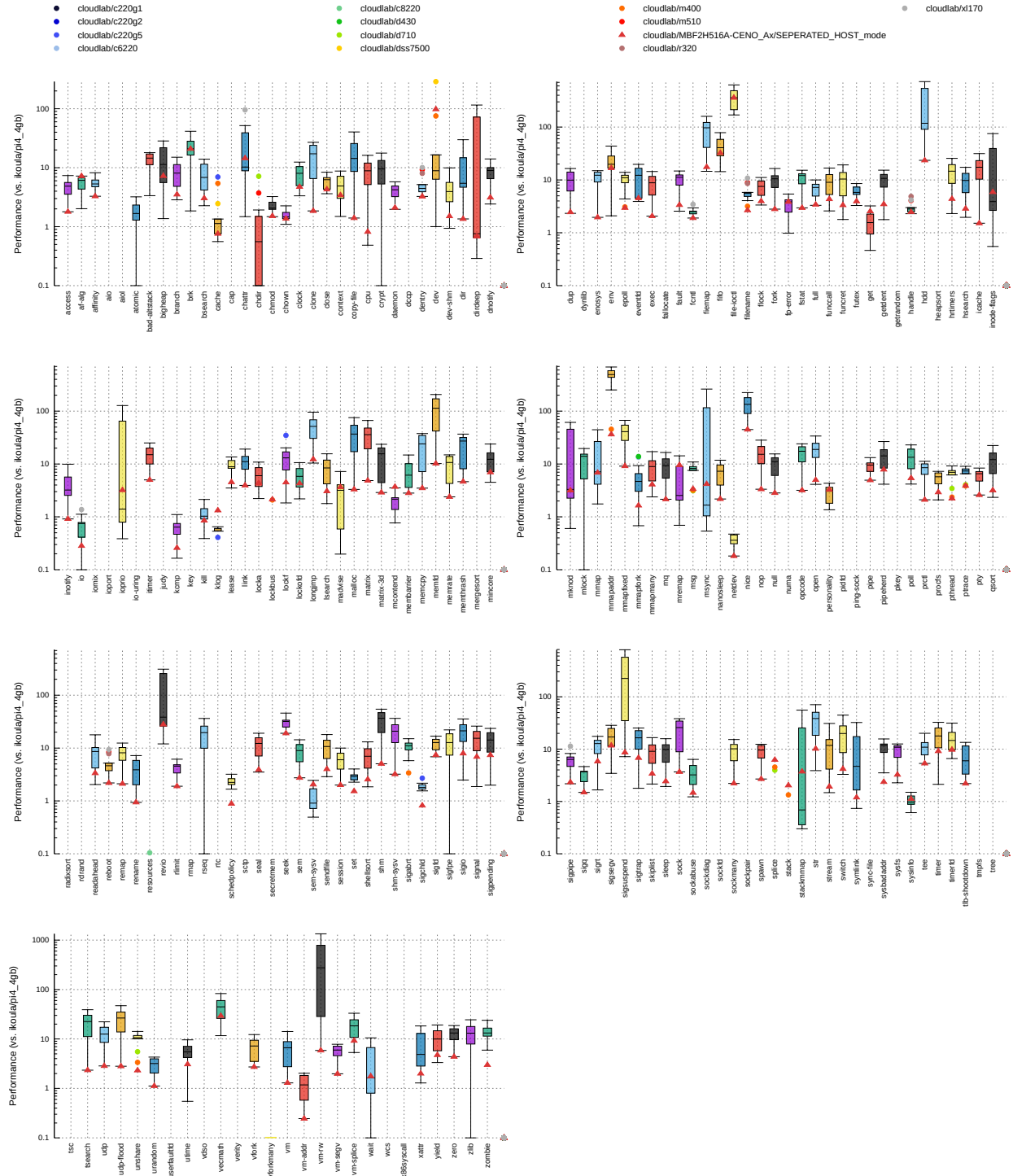
Figure 3-1. Box Plotting the Relative Performance of Different `stress-ng` stressors for 12 General-Purpose Platforms and the SmartNIC. The run time of each stressor was 60 seconds. We used the 4 GB model of the Raspberry Pi 4B as the reference platform for performance normalization. The MBF2H516A-CENO_Ax platform is the model name of the BlueField-2 SmartNIC in question. The data points of the SmartNIC are marked with triangles. The data points of the other platforms are plotted only if they are outliers (outside of the range of the corresponding whisker). Stressors without any data points are because they are not executed, hence they remain empty in the figure (e.g., aio and ioport).

41

3.1.2. *Individual Results Analysis*

As we expected, the BlueField-2 card's performance generally ranked lowest of all the systems tested except the RPi4. However, there were multiple tests where the BlueField-2 excelled and warranted a closer inspection, summarized below. The number in parentheses is the performance ranking of the BlueField-2 for a given stressor among all test platforms.

- **af-alg (#1):** AF_ALG [34, 35] is the kernel crypto API user space interface. It allows user programs to access the cipher and hash functions provided by hardware cryptographic accelerators. The BlueField-2 SmartNIC contains multiple hardware accelerators for cryptography such as the IPsec/TLS data-in-motion accelerator, the AES-XTS 256/512-bit data-at-rest accelerator, the SHA 256-bit accelerator, and the true random number accelerator. These all contribute to the outstanding performance on this stressor test.

- **lockbus and mcontend (#1):** The lockbus stressor keeps step forwarding a pointer while injecting write barriers in between. The mcontend stressor starts a couple of threads to concurrently update and read data residing in memory that is mapped to the same physical page. These two stressors simulate aggressive memory access patterns that could be more extensive than real-world programs. However, the BlueField-2 handles these memory access contentions very well.

- **stack (#1), mremap (#3), stackmmap and madvise (#5), msync (#6), mmap (#8), malloc, and vm (#13):** These stressors exercise the virtual memory subsystem of the OS running on the SmartNIC. Although we see the performance of accessing some memory interfaces on the BlueField-2 is significantly better than other servers, the most common memory interfaces, such as mmap and malloc, do not perform well on this device.

- **chattr and inode-flags (#5), ioprio (#6), file-ioctl (#7), dnotify and getdent (#12), copy-file, dentry, dir, and fstat (#13):** These stressors touch different interfaces provided by the filesystem. We see a considerable variation among these tests. It is important to note that the BlueField-2 does not show a performance advantage over other platforms when accessing other common interfaces for metadata operations, such as getdent(2) and fstat(2).

- **fp-error (#7), vecmath (#9), branch, funccall, bsearch, hsearch, lsearch, qsort and skiplist (#11), longjmp and shellsort (#12), cpu, opcode, and tsearch (#13):** These stressors focus on logic and arithmetic operations and include a variety of sorting tests. The cpu stressor performs a variety of math functions (e.g., manipulating bits, taking square roots, finding the greatest common divisor, and computing Apéry's constant). Surprisingly, the relative performance of the BlueField-2 on this stressor is less than 1, meaning its arithmetic performance is *even worse* than the RPi4. In contrast, the performance result of the vecmath stressor is interesting because the BlueField-2 performs *better* than some of the x86_64 CPUs on this test, including the D-1548 (Q4'15), the E5-2450 (Q2'12), and the Xeon E5530 (Q1'09).

- **cache (#11), icache (#13):** The cache stressor thrashes the last-level cache of the CPU by reading/writing content from/to the cache as fast as possible. In this test the BlueField-2 performs slightly better than the E5-2630 v3 (Q3'14), the E5-2660 v2 (Q3'13), and the

Xeon E5-2450 (Q2'12). However, all are still worse than the RPi4. The reason for this discrepancy is that the last-level cache on the BlueField-2 is the L3 cache, while the last-level cache on the RPi4 is the L2 cache. The icache stressor tests the instruction cache performance of the CPU by generating load misses using different sizes of memory pages. Overall, the CPU cache on the BlueField-2 does not have competitive performance over other platforms.

- **sigsegv (#9), timerfd (#10), signal, clock, and timer (#11), itimer, sigpipe, sigsuspend, and sleep (#12), nanosleep (#13):** The performance of these stressors represents the interrupt performance of the OS. These results indicate that software offloaded to the BlueField-2 should avoid using the OS's timing and interrupt interfaces when possible.

- **readahead (#11), hdd and seek (#13):** The local storage hardware of the BlueField-2 is an eMMC flash device, while the CloudLab general-purpose servers have either enterprise-class HDDs or SSDs. The relative I/O performance of the BlueField-2 is worse than most of the other platforms and suggests that offloaded functions should minimize their access to local storage.

- **sem-sysv (#2), fifo (#9), eventfd, poll (#11), futex (#12), hrtimers (#12), clone, exec, fork, nice, and pthread (#13):** System V inter-process communication (IPC) [36] is a set of mechanisms provided by Linux to simplify the communication between processes. The sem-sysv stressor measures how fast a pair of processes can safely increment and decrement a shared semaphore when mixing both legal and illegal arguments. The BlueField-2 SmartNIC's performance was better than all x86_64 platforms in this test. This result may be due to an architectural advantage of the Arm processors, as the m400 Arm and RPi platforms also ranked higher than expected. However, other scheduler-related tasks did not perform well on the BlueField-2. For example, the BlueField-2 scored poorly on the futex stressor, which uses the futex(2) [37] system call to wait until a condition becomes true.

- **sockabuse (#11), epoll, sockmany, sock, udp-flood, and udp (#13):** These stressors test the performance of the kernel network stack. The rankings of these stressors show the networking performance of the BlueField-2 using the kernel stack is worse than most of the other platforms in the comparison. This result is consistent with behaviors we observed in later tests that measured processing headroom during packet transmission.

We list the performance rankings of the SmartNIC among all test platforms for each stress test in Table 3-1. Considering the large number of stressors, we believe that the most useful way to represent this information is to show the best and worst results for the BlueField-2. This ordering provides us with guidance on the types of operations that should and should not be performed on the BlueField-2.

As a means of determining whether thermal or caching effects were impacting performance, we reduced the duration of each stressor from 60 seconds to 10 seconds and repeated the tests on all platforms. Table 3-2 lists the stressors where the BlueField-2 changed more than two positions in the overall rankings. The stressors with the biggest change are the CPU and CPU cache-related stressors. For example, the bigheap stressor exercises the virtual memory by bumping up the memory of a process with the REALLOC(3) [38] system call until an out-of-memory error is

Table 3-1. Performance Ranking of the BlueField-2 SmartNIC Based on the Results of Stressor Tests

Stressor	Stressor Classes	Ranking	Stressor	Stressor Classes	Ranking
mcontend	MEMORY	1	locka	FILESYSTEM | OS	9
splice	PIPE_IO | OS	1	lockofd	FILESYSTEM | OS	9
stack	VM | MEMORY	1	sigsegv	INTERRUPT | OS	9
dev	DEV | OS	2	vecmath	CPU | CPU_CACHE	9
sem-sysv	OS | SCHEDULER	2	chown	FILESYSTEM | OS	10
get	OS	3	env	OS | VM	10
mremap	VM | OS	3	timerfd	INTERRUPT | OS	10
chattr	FILESYSTEM | OS	5	...	...	...
inode-flags	OS | FILESYSTEM	5	...	...	...
madvise	VM | OS	5	...	...	...
personality	OS	5	bad-altstack	VM | MEMORY | OS	14
stackmmap	VM | MEMORY	5	getrandom	OS | CPU	14
sysinfo	OS	5	inotify	FILESYSTEM | SCHEDULER | OS	14
ioprio	FILESYSTEM | OS	6	netdev	NETWORK	14
msync	VM | OS	6	rename	FILESYSTEM | OS	14
brk	OS | VM	7	resources	MEMORY | OS	14
file-ioctl	FILESYSTEM | OS	7	rseq	CPU	14
fp-error	CPU	7	schedpolicy	INTERRUPT | SCHEDULER | OS	14
bigheap	OS | VM	8	sigabrt	INTERRUPT | OS	14
mknod	FILESYSTEM | OS	8	sigchld	INTERRUPT | OS	14
mmap	VM | OS	8	vforkmany	SCHEDULER | OS	14
revio	IO | OS	8	vm-addr	VM | MEMORY | OS	14

* We only show the stressors that BlueField-2 SmartNIC ranks ⩽ 10 or the last among all test platforms.

triggered. Furthermore, the rankings in the 60 second test are mostly higher than the stressors' corresponding rankings in the 10 second test. This information suggests that the Arm CPU on the BlueField-2 may need to be warmed up for optimal performance. Regarding the thermal dissipation, we have not seen a noticeable impact it caused on the performance of the BlueField-2.

Table 3-2. Changes in the Performance Ranking of the BlueField-2 SmartNIC in the 10s and 60s Tests

Stressor	Stressor Classes	10s Test	60s Test
af-alg	CPU | OS	7	1
bigheap	OS | VM	14	8
branch	CPU	14	11
brk	OS | VM	11	7
cache	CPU_CACHE	14	11
dirdeep	FILESYSTEM | OS	13	9
klog	OS	5	1
seek	IO | OS	7	13
sigfd	INTERRUPT | OS	14	11

3.1.3.   *Class Results Analysis*

`stress-ng` categorizes all stressors into 12 classes. To evaluate whether the BlueField-2 has performance advantages over other platforms in certain classes of operations, we calculate the

average relative performance of the stressors in a stressor class for each class and each platform and show the result in Figure 3-2. The relevant class definitions based on our understanding are listed below:

- DEV: Stressors that test a series of Linux device interfaces under /dev, e.g., /dev/mem, /dev/port, /dev/null, /dev/loop*.

- PIPE_IO: Stressors that test Linux pipe I/O interfaces, e.g., fifo, sendfile(2) [39], tee(2) [40], and splice(2) [41].

- MEMORY: Stressors that test the memory subsystem of a machine. Example stressors are malloc, memrate, stackmmap, and mcontend.

- INTERRUPT: Stressors that test different Linux interrupt signals. Example stressors are timerfd, sigrt, timer, and sigq.

- CPU: Stressors that thrash different functionalities of the CPU such as encryption, atomic operations, random number generation, and arithmetic calculation. Example stressors are crypt, atomic, getrandom, cpu, nop, matrix-3d, and vecmath.

- OS: Stressors that test various general system interfaces. Typically, stressors of this class also belong to some other classes. For example, the aforementioned stressor fifo, sigrt, and malloc belong to this class as well.

- NETWORK: Stressors that test the network performance and functionality of the system. Example stressors are sockfd, sockmany, rawpkt, udp-flood, and sctp.

- VM: These stressors focus on testing the virtual memory layer managed by the operating system. Even though as a process running on Linux, it is hard to test the hardware memory subsystem without testing the virtual memory; stressors in this class focus on some high-level virtual memory operations such as mlock [42], msync [43], swapon/swapoff [44], and madvise [45].

- CPU_CACHE: Stressors that can thrash the CPU cache. The previously discussed cache and icache stressors belong to this class.

- FILESYSTEM: As the name indicates, these stressors test various filesystem interfaces. Example stressors are iomix, xattr, flock, getdent, and fallocate.

- IO: These stressors try to test the raw storage performance as heavily as possible. Example stressors are revio, hdd, sync-file, and readahead.

- SCHEDULER: These stressors test the capability and stability of the system scheduler. Example stressors are zombie, nice, affinity, mq, spawn, and yield.
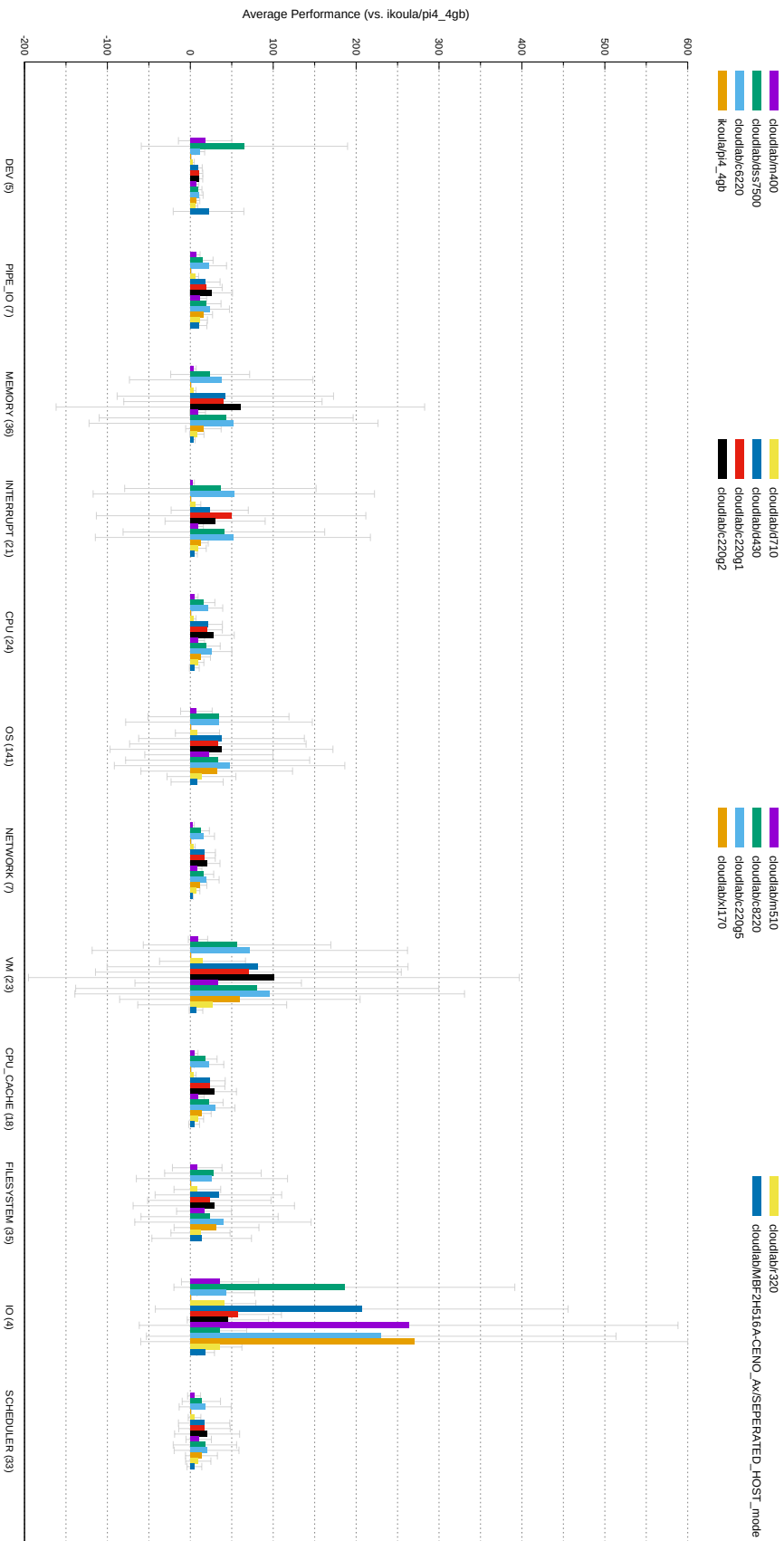
Figure 3-2. Average Relative Performance of Stressors in a Stressor Class for a Particular Platform. If a stressor belongs to multiple classes, its relative performance value will be added to each of the belonging classes in the calculation. The number beside the name of a stressor class (along the x-axis) is the number of stressors in that class. The whisker on a bar is the sample standard deviation of the average. The MBF2H516A-CENO_Ax platform is the BlueField-2 SmartNIC in question.

From Figure 3-2 we can see that the average performance of the BlueField-2 is on par with the 12-year-old x86_64 server d710 and the Arm server m400. However, since the average performance of almost all stress classes has a large variation, the comparison between them becomes less meaningful because the differences are not statistically significant. For example, the I/O stressor class only has four stressors, but on each test platform the standard deviation of the stressor results is almost as large as the average performance. These computational experiments confirm that the BlueField-2 SmartNIC platform performs worse than general-purpose x86_64 platforms, and that statistically there is not a single class of operations that the BlueField-2 Arm processors perform better than a traditional server's processors.

3.1.4.   *Summary and Insights*

To summarize our findings, the computational advantage boundary of the BlueField-2 is small. Therefore, offloading functions to the SmartNIC requires careful consideration about the resource usage and operations of the functions to ensure the design is tailored to the given embedded environment. Specifically, the function should avoid containing operations working on local storage and filesystem I/O, heavy CPU calculation, or relying on frequent software interruptions and system scheduling. Most importantly, avoid using the kernel network stack whenever possible by migrating the network stack to either user space or hardware-accelerated solutions. With that being said, some operations are profitable to be offloaded to the SmartNIC such as memory contention operations, cryptographic operations, and IPC operations. In general, executing memory-related operations is better than executing CPU, storage I/O, and kernel network-related operations on the SmartNIC. The exceptions to the CPU operations are encryption and vector calculations. The former is due to the built-in accelerators in the SmartNIC. The latter may be due to the optimization of the Arm CPU.

3.2.   Network Assessments: Packet Processing Headroom During Ethernet Transfers

The goal of our second performance evaluation is to determine how much processing headroom is available on the BlueField-2 card's embedded processors when the SmartNIC is busy transmitting network data at maximum speed using a 100Gb/s Ethernet network fabric. For this experiment we run a packet generator on the SmartNIC and then insert a variable amount of delay between packet bursts to model a scenario where computations are applied to a network stream. Finding the maximum amount of delay that can be tolerated before the network's performance drops gives an estimate of the amount of time the hardware has available for offloading computations. Given that we are seeking an upper bound on this time, it is useful to select a packet generator that yields maximum network bandwidth with minimal overhead. After our initial experiments with user space utilities such as iPerf [46], nuttcp [47], and Netperf [48] yielded suboptimal performance, we transitioned to the Linux `pktgen` [49, 50] traffic generation tool. `pktgen` is a kernel space tool that injects UDP packets directly into the kernel IP network stack.

3.2.1. Benchmark Considerations

There are multiple benefits to using `pktgen` in this evaluation. First, we observed that its performance was roughly 15% higher than the aforementioned user space benchmark utilities in resource-restricted environments. Second, `pktgen` has built-in support for symmetric multiprocessing and can start a kernel thread on each CPU core. This feature is important in 100Gb/s Ethernet environments where it is nearly impossible to saturate the network with a single core (e.g., a single-instance of iPerf achieved less than 40Gb/s in a previous study [51]). Finally, `pktgen` provides multiqueue support that allows a socket buffer's transmission queue to be mapped to the running thread's CPU core. This optimization reduces the overhead of cross-core communication and improves throughput.

There are three main options that we supplied to `pktgen` to adjust the settings of our experiments. First, the "delay" option was used to regulate the minimum amount of time allocated to send a batch (or burst) of packets. Varying this option provides us with a way to determine how much additional processing the SmartNIC can perform before network throughput drops. Second, the "clone_skb" option controls how often a network packet is reused for transmission. Setting this option to zero removes memory allocation times. Future tests may use larger values to increase allocation costs for communication. Finally, the "burst" option specifies the number of packets queued up before enabling the bottom half of the network stack. This option can be used to adjust interrupt coalescing.

3.2.2. Experiment Setup

`pktgen` threads use an infinite loop model such that when a `pktgen` thread is enabled, the associated CPU core will be fully occupied. Therefore, measuring the processing headroom with this tool requires two steps. In the first step, we need to measure the minimum configuration with which the SmartNIC can achieve the highest possible bandwidth. To be specific, we set the "clone_skb" to 0, and gradually increased the number of threads and the value of "burst" while recording the throughput changes. Once we have found the minimum configuration, the second step is to modify the "delay" setting to inject an artificial delay for each burst of packets. We find the maximum delay the SmartNIC can withstand before throughput drops for a particular number of threads. Thus, the maximum processing headroom available when transmitting a given batch of packets can be calculated by subtracting the time spent on sending the batch of packets without delay evaluated in the first step from the maximum delay evaluated in this step. Note that we kept the default MTU value (1500B) in all experiments as this should be the most common scenario. The experiments were conducted with the BlueField-2 card in both the separated host mode and the embedded function mode.

3.2.3. Evaluation in the Separated Host Mode

Conducting the first step of the experiment involved performing a sweep of packet sizes to find the minimum configuration settings that would generate the maximum bandwidth. Packet sizes

ranged from 128B to 10KB. Packets larger than exactly 10KB caused the test process to crash. Throughput measurements are presented in Figure 3-3.
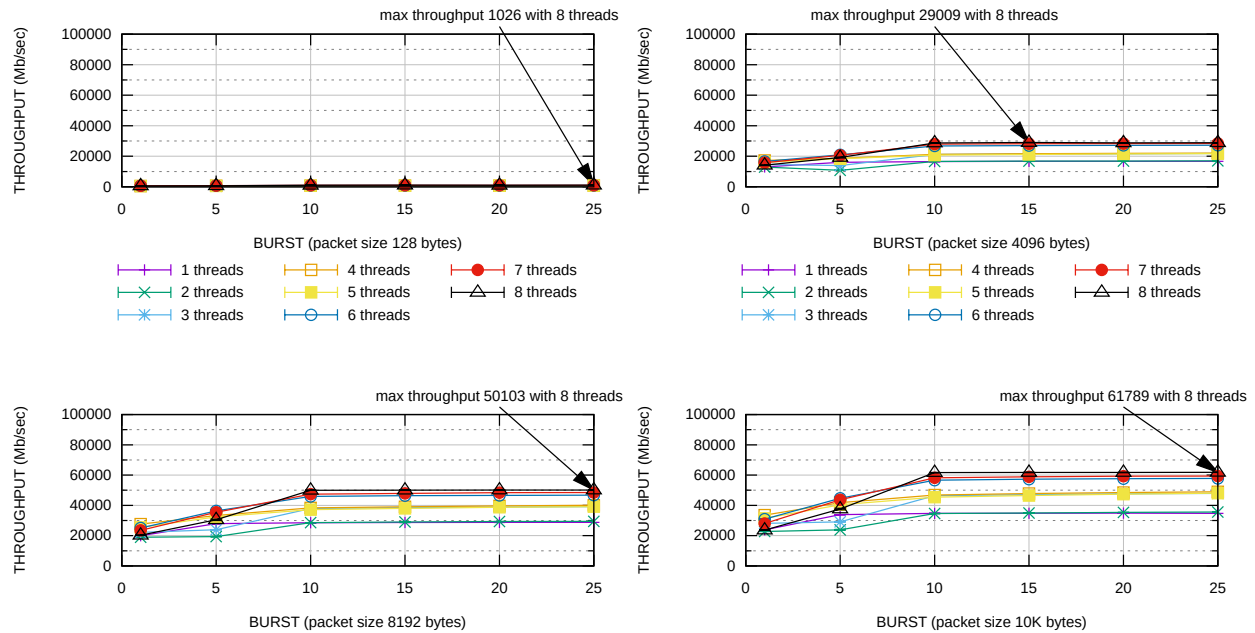
Figure 3-3. Throughput Results from the BlueField-2 SmartNIC in Separated Host Mode

The results for the BlueField-2 surprised us. While we knew that the BlueField-2 card's embedded processors would have trouble saturating the 100Gb/s network link, we expected that many worker threads generating large packets would be able to fill a significant portion of the available bandwidth. At best we found the card could only generate approximately 60% of the total bandwidth. On the one hand, this evaluation shows that the packet processing task itself is resource-intensive, even for the most advanced SmartNICs available nowadays. On the other hand, we see that offloading functions to this SmartNIC without changing the traditional network processing path used by these functions cannot achieve the best performance.

After accepting that the BlueField-2 can only realistically support approximately 50 Gb/s of traffic, the next step in our evaluation was to measure how much delay can be added to packet generation to determine how much processing headroom exists for this data rate. As presented in Figure 3-4 the maximum delay before bandwidth degradation is approximately 320μs. If we subtract the time required for each burst of transmission without delay from this delay, and convert the result to the available CPU percentage per core per burst of transmission (10KB x 25 = 250KB), we get 22.8% of CPU time left for application logic on the Arm cores if we only aim to use up to 50% of the full network bandwidth.

For comparison with a general-purpose processor, we ran a similar set of tests on the host where the BlueField-2 card resides (CloudLab machine type r7525). The host has two 32-core AMD 7542 CPUs (2.9 GHz) and 512GB DDR4-3200 memory. We varied packet sizes from 128B to 1KB and saw that the 100Gb/s link could be saturated with a packet size of only 832B. The results are presented in Figure 3-5. Thanks to the host's much more powerful system resources, saturating the network can be accomplished with only 5 threads (corresponding to 5 vCPU cores)
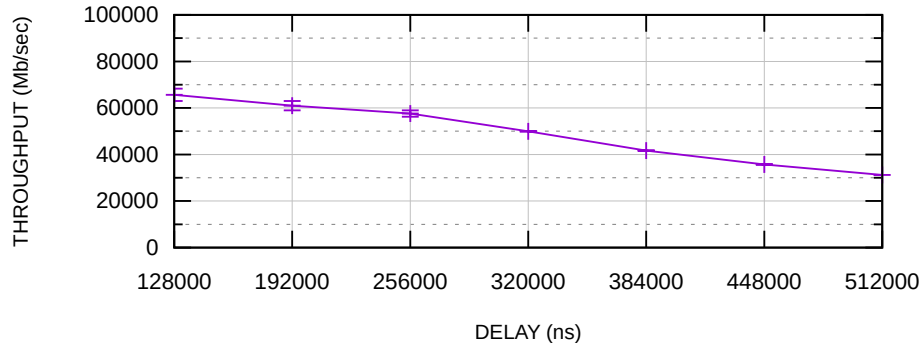
Figure 3-4. Throughput Results from the BlueField-2 SmartNIC with Different Delay Configurations (8 Threads, 10KB Packets, 25 Packet Bursts)

and a burst size of 25. While additional threads had a minimal impact on throughput for the 832B packet case, we observed a performance drop in larger packet sizes such as 1KB when more threads were added. We believe this drop is due to resource contention.
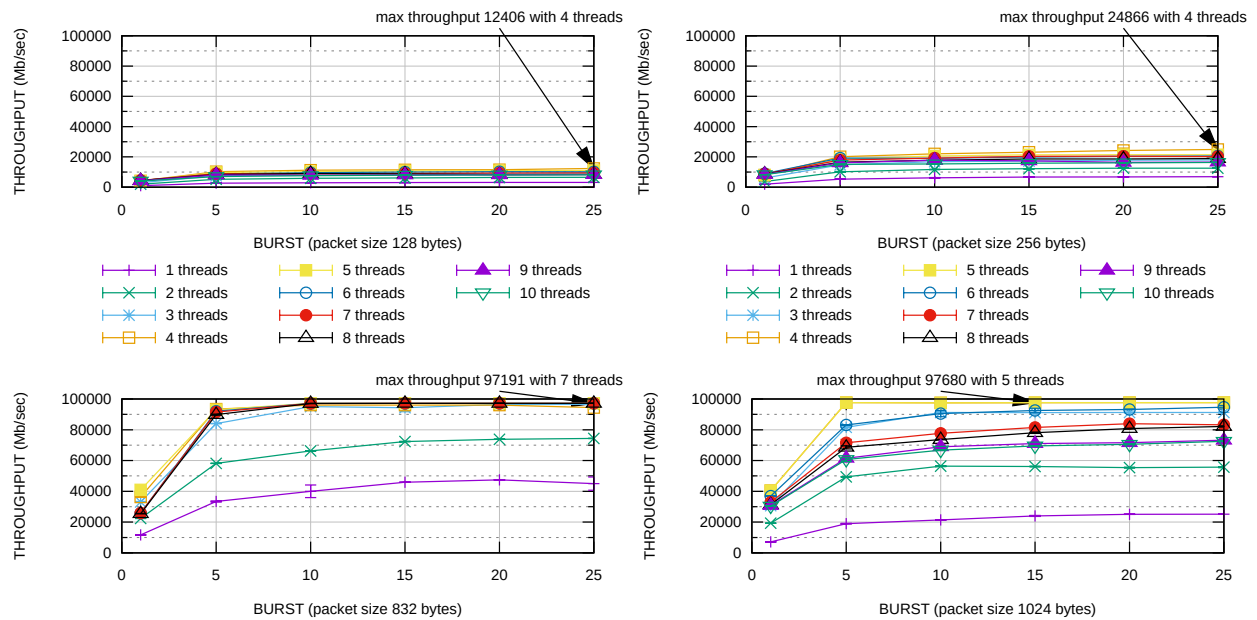
Figure 3-5. Throughput Results from the r7525 Machine in Separated Host Mode

Using the minimum configuration (5 threads and a burst of 25), we can then inject a small delay per burst to see how well the full bandwidth can sustain on the host (Figure 3-6). Based on the results, the host can afford 8μs delay per burst without using additional threads. This delay is equivalent to <1% of CPU time available for handling application logic on these five cores. However, this savings is not significant given that the host has an additional 123 vCPU cores available for use by applications.

50
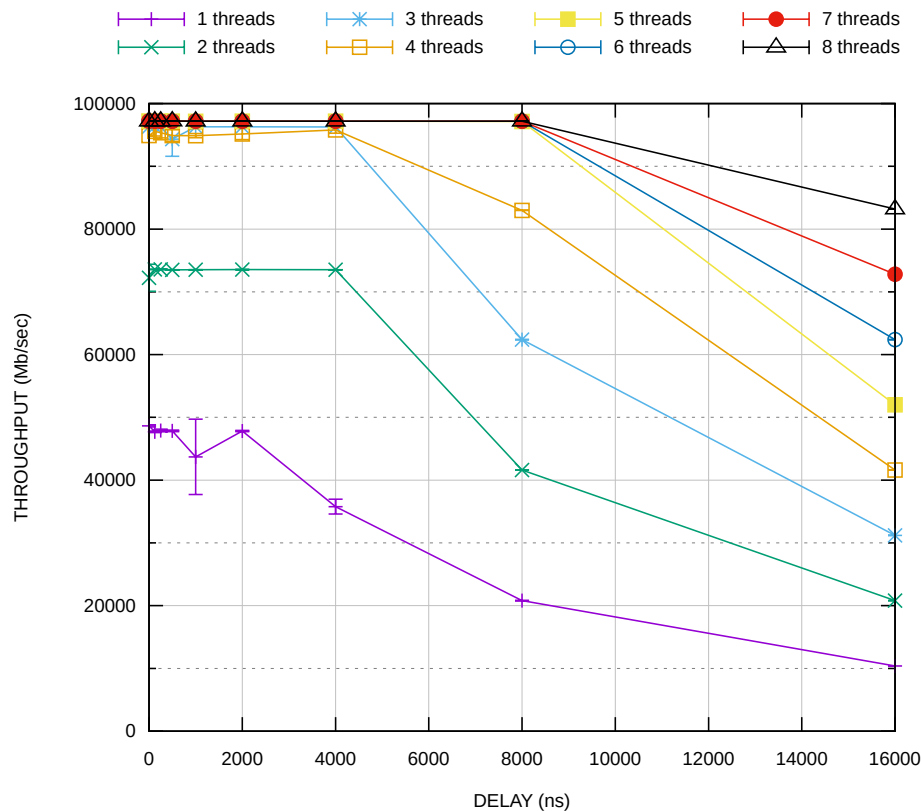
Figure 3-6. Throughput Results from the r7525 Machine with Different Delay Configurations (832B Packets, 25 Packet Bursts)

3.3.　　　Network Assessment: InfiniBand RDMA Performance

While 100Gb/s Ethernet performance is important for data centers and clouds, HPC users require a high-speed interconnect such as InfiniBand to ensure that their parallel computing applications minimize the amount of time spent exchanging data. InfiniBand hardware features low-latency message queues and high-bandwidth remote direct memory access (RDMA) engines to allow a low-level communication library to move data between user space applications and the network fabric efficiently. While message passing libraries such as MPI must wrap a great deal of functionality around an RDMA communication library to make InfiniBand usable to application developers, it is useful to characterize the performance of the RDMA hardware because it gives an upper bound for what the network is capable of performing.

This section characterizes RDMA performance for InfiniBand and focuses on three fundamental questions. First, what are the raw RDMA transfer rates between different pairs of hosts and SmartNICs in the network? These measurements provide insight into how quickly the hardware can execute network operations and help us estimate whether the Arm processors have sufficient compute power to saturate an InfiniBand network link. Second, is there a performance advantage for a host to interact with its local SmartNIC compared to a remote host or SmartNIC? Substantial locality benefits would motivate researchers to examine how communication software could

51

better exploit a tight coupling between a host and its local SmartNIC. Finally, what are the consequences of the BlueField-2 SmartNIC's host and Arm processors sharing the same network connection? While SmartNICs provide a cost effective way to supplement a platform with additional processors, it is important not to impede the general communication performance of the host processors.

3.3.1.   Experiment Setup

A great deal of information can be determined about InfiniBand RDMA performance through the use of the `ib_send_bw` tool. `ib_send_bw` is designed to move large amounts of data between a pair of network endpoints and includes options for users to vary parameters such as the message size and transmission method. For our first two questions we configured `ib_send_bw` to perform a bandwidth test for a wide range of message sizes using different pairs of hosts and Arms. These tests measure how quickly (1) a host can RDMA data to processors at the local SmartNIC, a remote host, and a remote SmartNIC and (2) how quickly an Arm's processors can RDMA data to processors at the local host, remote host, and remote SmartNIC. For our third question we constructed an experiment where host-to-host communication is performed at the same time SmartNIC-to-SmartNIC communication takes place. The hosts in this experiment perform an `ib_send_bw` sweep while the Arms on the SmartNICs run continuous bandwidth transfers with a fixed message size. Adjusting the message size used in the Arm transfers allows us to scale the amount of contention there is for the outgoing network link.

3.3.2.   Fundamental RDMA Performance

The results of the first experiment are presented in Figure 3-7. As expected RDMA performance starts at approximately 5MB/s for small messages and then ramps up to GB/s speeds when messages are larger than 512B. In all six experiments the maximum bandwidth was reached when messages were only 4-8KB in size. In contrast to our experiments with Ethernet, the SmartNIC Arm processors had no difficulty saturating the outgoing network link. These performance curves highlight how efficient RDMA is at transferring data from one application's memory space to another.

These measurements also illuminate the performance differences between local and remote transfers. Once messages were large enough to saturate the network link, the host was able to transmit data to the Arm processors on the local SmartNIC 6.99% faster than when it communicated with remote host processors. Similarly, the Arm processors achieved 6.98% more bandwidth when they communicated with the local host. These measurements indicate that there is a slight performance advantage for intra-host communication.
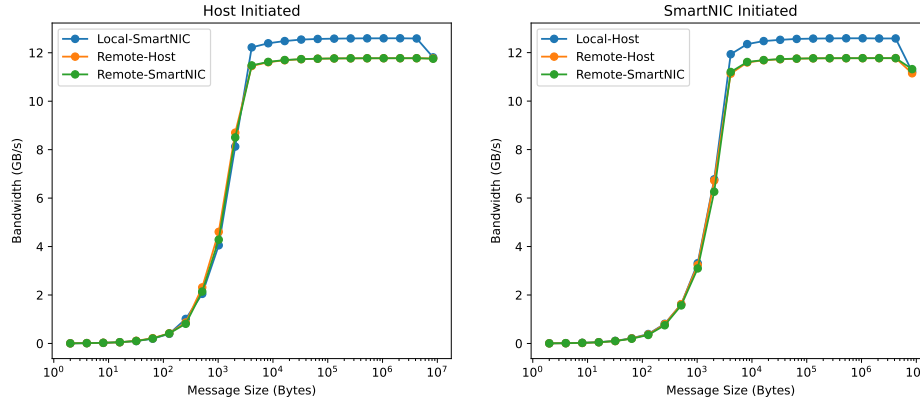
Figure 3-7. RDMA Bandwidth between Different Pairs of Hosts and SmartNICs

3.3.3.   *Consequences of Sharing a Network Link*

The second RDMA experiment focuses on characterizing how the BlueField-2 SmartNIC behaves when the host and Arm processors both need to transmit data on the network link at the same time. We configured the Arm processors in a pair of compute nodes to use `ib_send_bw` to continuously transmit data with a fixed message size and then ran an `ib_send_bw` sweep test on the corresponding host processors. We repeated the experiment with four different message sizes on the Arm processor to adjust the amount of contention the host and Arm processors would have for the network link. As the results indicate in Figure 3-8, the communication performance of the host processors decreases when the Arm processors increase their network traffic. When the Arm processors generate enough traffic to saturate the entire link (4-8KB), the host is only able to deliver 5.86GB/s of bandwidth, which is approximately half of the maximum data rate observed in the previous host-to-host transfer measurements.
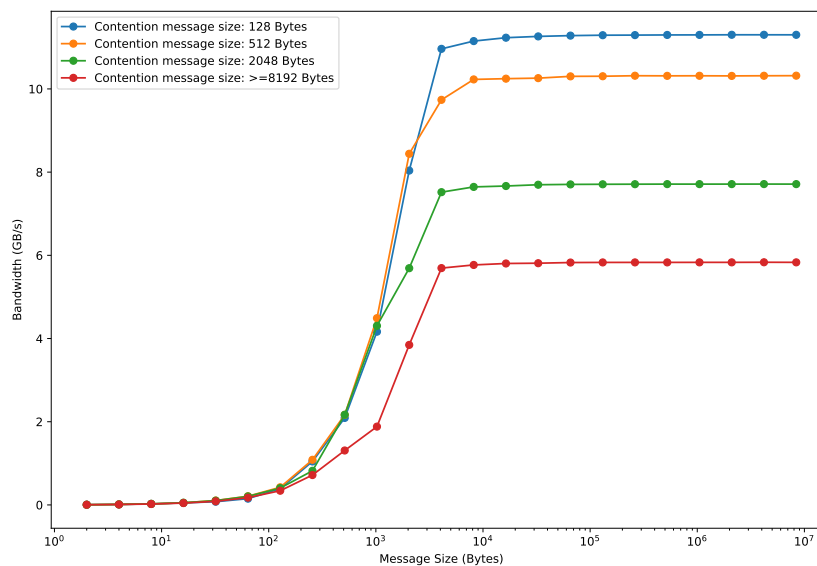
Figure 3-8. Impact of SmartNIC Communication on Host Communication

This experiment indicates that the BlueField-2 SmartNIC fairly distributes available network bandwidth to communication endpoints on demand. From one perspective this is useful because it enables applications to benefit from the network fabric whether they run on the host or the SmartNIC. However, HPC users should be aware that services offloaded to a SmartNIC can interfere with the communication performance of applications that run on the host. Future work should explore whether users can schedule their host and service operations to ensure that services do not delay latency sensitive operations on the host.

3.4. Hardware Accelerators

The BlueField-2 architecture features multiple hardware accelerators that may be relevant to different data services. The following is a summary of our experiences with the different units:

- **DEFLATE Compression**: The BlueField-2 SmartNIC includes a hardware compression unit that enables users to compress and decompress data at high speeds using the DEFLATE algorithm. This hardware unit can be accessed from both the host and the Arms and is most effective when a large amount of data is split into pieces that are then streamed through it. Given the importance of performant compression in data services, we examine the tradeoffs of working with this hardware in detail in Chapter 5.

- **Regular Expression**: Network security researchers often need an efficient mechanism for applying a regular expression on a moderate amount of data. The BlueField-2 provides a regular expression (RegEx) hardware unit that applies a precompiled rule in the rof2 format to a data stream. Software for interfacing with this hardware is available in both DPDK and DOCA. While we see great value in this hardware, we were unable to access it. We believe that it is not available in separated host mode.

- **SHA-2**: The BlueField-2 includes a hardware acceleration unit for the secure hash algorithm (SHA) defined in the FIPS 180-4 specification [52]. This hardware is capable of generating cryptographically-sound hashes for data using the SHA-1, SHA-256, and SHA-512 algorithms. While we did not explore the use of this hardware, researchers may find it useful for quickly hashing a memory region to determine if contents have changed.

- **Erasure Coding Offload**: Mellanox began integrating Reed-Solomon erasure coding offload hardware into NICs starting with the ConnectX-4 architecture. Storage services often leverage erasure coding techniques to generate parity blocks that help a system repair data when there is corruption. Given that the ability to offload this task to SmartNIC hardware would be valuable, we explored different options for leveraging the BlueField-2 erasure coding hardware. Unfortunately, the previous ibverbs interface for accessing erasure coding hardware has been deprecated in recent versions of the software and we were unable to test the hardware. NVIDIA has indicated that erasure coding support will be available in upcoming releases of the BlueField-3 card.

54

3.5.   Summary

In this chapter we have explored the fundamental characteristics of the BlueField-2 SmartNICs. Key findings of the work include the following. First, the Arm processors are able to perform a wide variety of tasks, but users should recognize that they are embedded processors that are roughly an order of magnitude slower than host processors at computational tasks. Second, it is challenging for the BlueField-2 SmartNIC's Arm processors to fully saturate a single 100Gb/s Ethernet stream because of the overhead involved in managing TCP/UDP data in the OS stack. In contrast the Arm processors can easily saturate a 100Gb/s InfiniBand connection due to the efficiency of the RDMA hardware. Third, the ConnectX-6 on the BlueField-2 mixes traffic from the host and Arm processors on demand and exhibits fairness when both endpoints generate data at the same rate. Additional quality of service (QoS) mechanisms may be necessary to ensure SmartNIC traffic does not impact host operations that are latency or bandwidth sensitive. Finally, the BlueField-2 offers multiple hardware accelerators. While some of these devices may be difficult to leverage in HPC applications, others such as compression are especially useful for data management services.

4.   AN ENVIRONMENT FOR HOSTING DATA SERVICES IN SMARTNICS

The fact that SmartNICs add processing, memory, and storage resources to the network fabric has motivated multiple research communities to investigate how the hardware can be applied to solve problems in different application spaces. Researchers have discussed using SmartNICs to improve infrastructure security in clouds [13], manage hardware resources in disaggregated architectures [27], execute tasks in computational frameworks [29], and apply computations to in-transit data in HPC applications [31]. A key challenge in all of these efforts is creating an environment that makes it easy for users to take advantage of SmartNIC hardware for a specific use case. This environment is the result of many design factors, including system access policies, standards for representing data, the mechanisms by which user-defined computations are executed, and the communication interfaces that enable data to be exchanged between host applications and SmartNICs.

In this chapter we examine the operating environments that other SmartNIC researcher have created for their use cases and define fundamental requirements that we believe are necessary for creating an environment where data management and storage services can be offloaded to SmartNICs. We then focus on adapting two existing software libraries to provide an environment that meets our requirements. For computations we leverage Apache Arrow because it features a robust data model and a flexible compute engine. For communication we employ Sandia's Faodel library because it includes simple but powerful primitives for transferring data and invoking remote computations in a heterogeneous HPC platform.

4.1.   Existing SmartNIC Application Environments

Researchers have proposed using SmartNICs to solve a variety of challenges that arise in different problem spaces. Each of these efforts involves creating a suitable software environment for supporting a particular workload. The following examples highlight how SmartNIC application environments vary depending on the use case:

- **Infrastructure Security**: Over the last decade cloud vendors such as Amazon and Microsoft have integrated SmartNICs into their platforms to help prevent compromised hosts from being able to disturb other infrastructure in the platform. The SmartNICs in these systems typically function as in-line network filters (e.g., via Embedded Function Mode) and are not exposed to end users. As such, the SmartNIC application environment in these systems is largely focused on executing packet-processing rules on streaming data efficiently, distributing rule updates securely, and capturing runtime metrics in a way that can be queried by remote management systems.

57

- **Disaggregated Resource Management**: Multiple vendors have discussed using a SmartNIC as an embedded server for managing collections of GPU or NVMe devices. For example a BlueField-2 SmartNIC can be configured to serve as the PCIe root complex for a headless PCIe expansion chassis. Similar to the infrastructure security example, users are unaware of the SmartNIC's presence in the system and simply acquire access to resources over the network through management software. While the SmartNICs in these systems could implement user-defined functions to reduce query results, application environments for these systems have largely focused on tunneling data back to the user through protocols such as NVMe-oF.

- **In-Transit Computations**: Researchers in the HPC community have proposed adding compute hardware to NICs to allow computations to be applied to data as it moves through the network [53, 31]. These approaches benefit from an environment where the SmartNIC performs on-path processing (e.g., the BlueField-2's Embedded Function Mode) and focus on extending communication library standards to enable developers to designate how computations are applied to data.

- **Task Offload**: Many researchers over the last decade have focused on task-processing frameworks that enable a runtime system to determine when and where particular tasks should execute in a distributed system [54]. In iPipe [29] researchers created an actor-based framework that includes support for tasks to be offloaded to a BlueField SmartNIC. These systems require an environment that allows the host to exchange work requests and data objects with the SmartNIC, as well as a mechanism for relaying state information to assist in scheduling. While users do not need to be aware of SmartNICs, they must define their operations in terms that fit the framework's programming model.

While each of these items is an appropriate solution for its respective problem space, none provide an ideal environment for executing the data services that are the focus of this project. The first two environments target infrastructure management and hide the SmartNIC from the end user. Environments from the in-transit computations work are relevant but focus on shifting targeted operations into the network path as opposed to hosting distributed, autonomous services in the network fabric. Task-offloading efforts offer the most attractive environment for our research, as they enable a way for service developers to generate work that can be scheduled on host or network processors. However, a criticism of this approach is that it depends on the existence of a task-processing framework that is actively supported and suitable for SmartNICs. We are not aware of a task-processing framework that currently satisfies these two conditions. Rather than build a new framework, we focus on creating an environment for hosting services that is composed of existing computation and communication libraries.

4.2.   Environment Requirements for DMSSes

While previous SmartNIC research has focused on accelerating simulation code performance, the Offloading Data Management Services to SmartNICs project targets a broader perspective that seeks to improve the manner in which complex, parallel *workflows* execute on a platform. Working at the macro level simplifies some aspects of our work, as there is less urgency to find

and offload tasks that are on the critical path of a single, complex application. However, the challenge of targeting workflows is that by definition, data must be transitioned from the mind space of one application to the mind space of another. These transitions may involve data transformations and explicit handoffs that come at irregular intervals. As such, we advocate a *service-based* approach to integrating SmartNICs into HPC environments, where applications use services with well-defined APIs to orchestrate the data flow between applications. Based on our previous experiences, we identify five basic requirements ($R1 - R5$) we expect from an environment where services execute in embedded devices distributed throughout the architecture:

$R1$ **Common Data Representation**: A common data representation is necessary in order to enable producers, consumers, and SmartNICs to be able to read and process in-transit data. This representation must be able to support a variety of complex data structures and be serializable into a contiguous memory allocation for network transmission.

$R2$ **Data Parallel Computations**: The environment must provide an easy-to-use computing framework that allows users to define computational functions that are applied at remote endpoints. This framework must be able to automatically map data-parallel computations to available resources to maximize performance without burdening developers.

$R3$ **System-wide Accessibility**: The environment must provide communication primitives that make it easy for any endpoint in the system (host or SmartNIC) to be able to directly communicate with any other endpoint. Approaches that use the host system as a proxy for its SmartNIC place unnecessary strain on the host and complicate communication operations for developers.

$R4$ **Resource Pools**: Complex data services often use a collection of endpoints to implement work in a parallel manner. The environment must provide a simple means of grouping host and SmartNIC endpoints into named resource pools to simplify development.

$R5$ **Dispatching Computations**: Services may execute in a variety of ways. While some services trigger work at fixed time intervals, others may operate by reacting to incoming data or remote computation requests. Therefore a data service environment must provide multiple mechanisms for users to dispatch computations at remote endpoints.

4.3. Resolving Computational Requirements

Our first two requirements for creating an environment for offloading data management and storage services focus on our computational needs. Researchers in the HPC and data science communities have independently constructed advanced, data processing libraries that greatly complement the functionality of composable data service libraries. These libraries define robust data structures for organizing information and are designed to exploit the parallel-processing capabilities of modern CPUs and GPUs. Popular data processing libraries in this space include VTK-m [55], Kokkos [56], and Apache Arrow [57]. We selected Apache Arrow for this project because it implements a rich set of primitives for representing, processing, and transmitting tabular data.

4.3.1. Apache Arrow

Apache Arrow is an open-source[1] project centered around an in-memory format specification and serialization protocol for column-based table data. It is SIMD [58] and vectorization friendly and relocatable, enabling zero-copy access in shared memory. The Arrow project includes a number of libraries for efficiently processing this data that are implemented in multiple languages (including C++) running on multiple platforms. In C++ an Arrow table is a two-dimensional data structure with chunked arrays for columns and a schema. Tables can be processed without copying using reference-counted record batches that hold contiguous portions of the data. Record batches enable work to be spread across multiple processors. Moreover, because of the contiguous property within a record batch, data processing can further take advantage of data-level parallelism using SIMD instructions that are generally available on modern x86 and Arm processors. Apache Arrow has been adopted by many research and commercial projects such as Apache Spark [59], Dask [60], and Polars.

Specific aspects of Arrow that meet our requirements follow.

R1 **Common Data Representation**: Arrow's tabular data model is suitable for describing many kinds of scientific datasets and provides a useful standard for data exchange. In addition to efficient, in-memory data structures for storing and processing tabular data, Arrow includes serialization software for converting data to a standard, on-wire format. This software simplifies development and improves interoperability with other libraries.

R2 **Data-Parallel Computations**: One of the benefits of Arrow's robust, tabular data model is that users can specify high-level queries that can be processed efficiently with parallel-processing techniques. Specifically, Arrow includes a streaming data processing engine named Acero [61] that processes complex user queries on tables. Acero extracts a computational graph from a query and then maps the data flow to local processing cores.

4.3.2. Arrow's Compute Performance

As a means of validating that Apache Arrow's runtime does map work to parallel resources, we constructed three computational kernels and measured their execution performance in different hardware configurations. Each kernel performed a specific task on a large block of particle data. A row of this data contains the identity, position, and velocity of a particle in a three-dimensional space, with values randomly generated to reside in a unit cube. The first kernel ("maximum") is an aggregate function that computes the squared magnitude of each particle's velocity and returns the maximum value observed in all the data. The second kernel ("normalize") is a projection function that computes the magnitude and normalized form of each particle's velocity. The third kernel ("bounding box") is a filter operation that removes all particles that are outside of a bounding box. In terms of data flows, these kernels respectively reduce the input to a single value, supplement the input by a constant amount, and reduce the input based on a filtering parameter.

---

[1] https://github.com/apache/arrow

We conducted performance measurements on the Glinda platform that measured the amount of time required for Arrow to process 2^{20} particles using a range of threads. As illustrated in Figures 4-1 and 4-2, Arrow improves in performance as the number of threads increases, with the largest gains taking place at 2 threads. As expected the host offers better performance than the SmartNIC for the same number of cores. However, host improvements diminish at approximately 8 threads. For the best performing configuration of each architecture, we observed that the host was 4.55x and 7.91x faster than the SmartNIC for the maximum and normalize kernels.
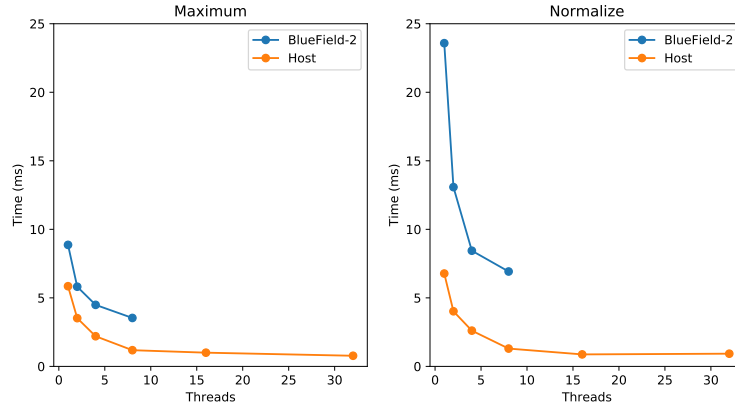
Figure 4-1. Arrow Performance for the Maximum Value and Normalize Kernels

For the bounding box kernel experiments, we adjusted input parameters to vary the amount of data returned from 100% to 1%. As illustrated in Figure 4-2, increasing threads helps performance because the algorithm still needs to scan all particles and make a simple filtering decision. However, reducing the size of the bounding box improves performance in all but the 100% case because there is less data to manage. The 100% case is faster than the other scenarios because the input table can be returned as the output table via smart pointer updates. The host was 9.92x faster than the SmartNIC in the 100% case, and 5.57x to 6.05x faster in other cases.
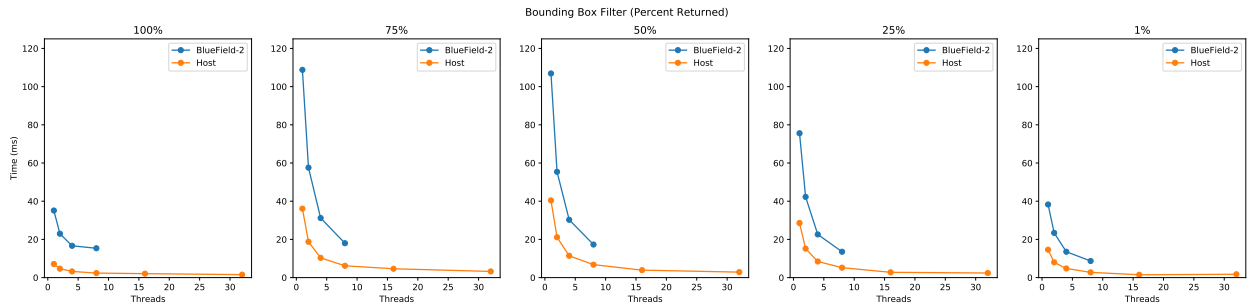
Figure 4-2. Arrow Performance when Filtering by Bounding Box

4.3.3.   *Comparing Arrow and Kokkos*

Given that Apache Arrow can map computations to parallel hardware resources, it is beneficial to compare it to other performance portability libraries to evaluate its value to end developers. We

focus on Kokkos for this work because Kokkos is widely used within Sandia and provides an easy-to-use programming paradigm for expressing data parallel computations in C++. Users store multidimensional data in a Kokkos "view" and specify kernels in lambda expressions that are dispatched through parallel for, reduce, and scan functions. While Kokkos lacks Arrow's I/O, schema, and serialization features, it is straightforward to migrate data between Kokkos and standard scientific computing file formats such as HDF5 or ExodusII.

We ported the three kernels from the previous section to Kokkos implementations (see Appendix A) and represented our tabular particle data in a Kokkos view. Performance measurements for the Arrow and Kokkos implementations on the BlueField-2 SmartNIC's processors are presented in Figure 4-3. The maximum kernel contains a single `parallel_reduce` loop. While the Kokkos implementation is slightly faster, neither library received much benefit beyond four threads due to the simplicity of the operation. The Kokkos implementation of normalize requires an explicit allocation for output data and uses a `parallel_for` loop to convert every input to an entry in the output. Arrow performed 1.98x faster than Kokkos with eight threads.
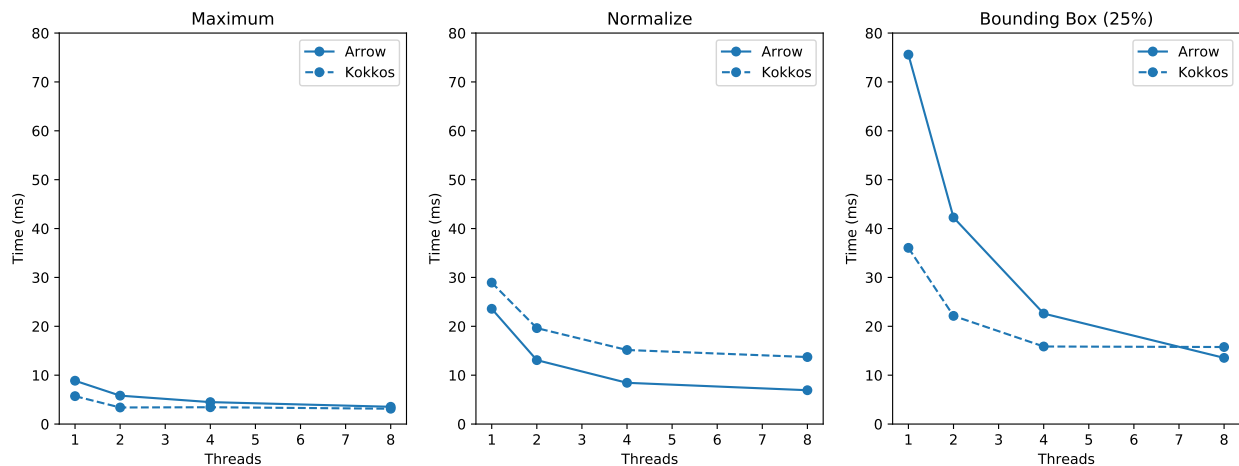
Figure 4-3. Arrow and Kokkos Performance on SmartNICs

Finally, the bounding box implementation for Kokkos requires three steps: a `parallel_scan` to build an index of the particles to keep, a memory allocation for the output particles, and a `parallel_for` to copy retained particles to the output view. The Kokkos implementation typically yields better performance than Arrow when using a small number of threads. However, Arrow typically provides the best overall performance when all threads are employed. Arrow's advantage is proportional to the amount of data returned by the query. Arrow is 1.94x to 1.16x faster when 100% to 25% of the data is returned. However, Kokkos is 1.07x faster when only 1% is returned.

Examining the Arrow and Kokkos implementations reveals other important distinctions between the libraries. First, in situations where an algorithm is specified at compile time, it is significantly easier to implement it in Kokkos. The Kokkos implementations of the three kernels involved constructing simple lambda expressions and selecting the proper parallel function to dispatch the work. In contrast the Arrow implementations involved constructing a graph of operators from

Arrow's C++ classes and applying the graph to an input table. Our second observation is that Arrow is a superior choice if the system is designed to process queries that are defined at compile time. Arrow enables users to write their query plans in a JSON format that can be consumed by Arrow's compute engine. Additional information about implementing complex queries for Arrow is discussed in Chapter 7. Finally, in terms of data structures, Kokkos views are easy to access but potentially dangerous, while Arrow tables are cumbersome but robust. Arrow's ability to house complex data tables and efficiently pass the data structures to other libraries without copies is particularly appealing in this project.

4.4.   Resolving Communication Requirements

The remaining set of environment requirements focus on communication challenges. A number of research groups have constructed *composable data service* libraries for HPC platforms to improve how data flows between workflow tools. These libraries include DataSpaces [3], Mochi [1], and Faodel [12]. Composable data service libraries provide flexible communication software that makes it easier to route data from one application's memory space to another's without using the file system. An important aspect of this work is that users are presented with higher-level primitives than are normally found in communication libraries. In addition to low-level RPC and RDMA facilities, composable data service libraries include key/value stores, REST API engines, and I/O drivers for interacting with external data repositories. These features simplify development and enable users to reason about their data at higher levels of abstraction. We selected Faodel for this project because it aligned with our communication requirements and has a native C++ implementation that is easier to integrate with Arrow than other C-based composable data service libraries.

4.4.1.   Faodel

Faodel provides an example of a composable data service library that supports multiple HPC platform architectures. Faodel is open-source[2] C++ software that includes drivers for InfiniBand [62], RoCE [63], and Cray Aries [64] network fabrics. Faodel is composed of several components: an RDMA portability library (NNTI) for low-level communication; a state-machine engine (OpBox) for managing asynchronous tasks; a memory-management library (Lunasa) for tracking memory allocations for network-accessible objects; a directory service (DirMan) for maintaining workflow configuration information; a key/blob service (Kelpie) for safely transferring objects between servers; and a lightweight web server (Whookie) to allow users to query a remote service. In prior work we have used Faodel for I/O staging and checkpointing [10], coupling visualization applications to simulation codes, and insulating users from platform-specific storage issues [65].

---

[2]https://github.com/sandialabs/faodel

The following describe how Faodel meets our project's environment requirements.

R3 **System-wide Accessibility**: Faodel assigns a unique identifier to each endpoint in the system that any other endpoint can use to communicate with the endpoint. This identifier is an IP address and port number for the endpoint's Whookie HTTP server. The HTTP server can be used to relay simple commands or establish RDMA communication channels with the endpoint. Faodel's Kelpie library provides an easy-to-use mechanism for safely transferring key-labeled objects between endpoints using asynchronous RDMA operations. Users can put, get, list, and delete objects on local or remote endpoints.

R4 **Resource Pools**: Kelpie uses a simple pool abstraction for grouping multiple endpoints together for related work. A pool is defined by a name, a list of endpoint members, and a distribution policy that determines how key labels are mapped to pool members (e.g., distributed hash table, map by producer rank, or map by an explicit identifier in a key). Pool information is maintained in the DirMan server for a workflow and can be updated and queried by the workflow's owner. Endpoints can participate in multiple pools with the caveat that developers must take care to avoid collisions between keys. An advantage of these resource pools is that users can supply different configurations at runtime to modify the behavior of their data flows.

R5 **Dispatching Computations**: While Kelpie is agnostic about data formats and computations, it provides two methods for users to invoke their own computations at endpoints. First, an endpoint may run its own main loop that periodically inspects state and reacts to changes. Second, users may invoke computations on objects at remote endpoints through user-defined functions. A remote computation may result in the remote endpoint retrieving zero, one, or more local objects for the function, but will always return a single object to the requester through RDMA mechanisms.

4.4.2.   *Faodel Stress Tests*

As a means of comparing how well the BlueField-2 SmartNIC's Arm processors perform bookkeeping tasks for network operations, we conducted multiple experiments with Faodel's built-in stress-test tool. Similar to `stress-ng`, this tool implements a series of simple microbenchmarks that represent common bookkeeping tasks that are needed by network layers. Each test implements a single task that is repeated as many times as possible over a fixed period of time to determine the rate at which a number of threads can perform the task. Faodel's LocalKV test uses a workload that employs multiple threads to put, get, and delete objects from a local, in-memory, 2D hash map. Key names are intentionally picked to either seek or avoid collisions. This test exercises common data processing tasks, such as hashing, reference counting, lock handling, and managing memory allocations.

We executed the LocalKV test on a diverse set of platforms to observe how the BlueField-2's processors performed compared to other architectures. The processors included: a 32-core AMD EPYC 7543P (Zen3) processor, a 68-core Knights Landing (KNL) processor, and BlueField-1 and BlueField-2 SmartNICs with 16 and 8 Arm cores respectively. As depicted in Figure 4-4, aggregate performance (decreases/increases) as thread counts increase in the collision
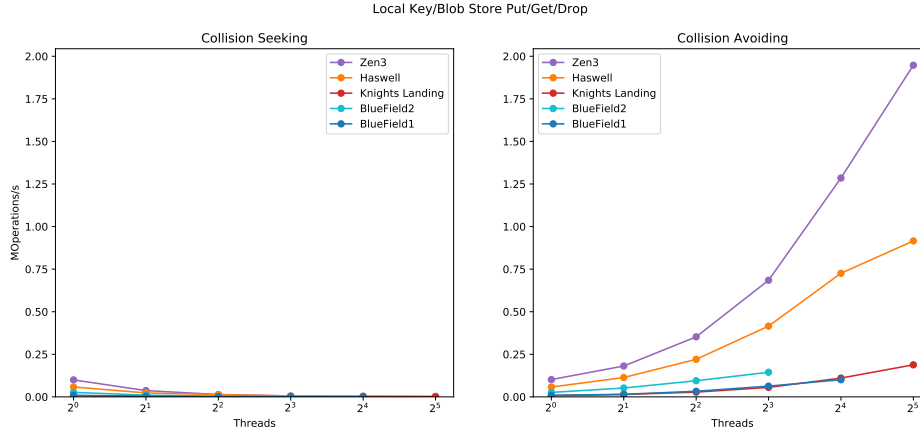
Figure 4-4. Aggregate Performance for Different Architectures in Faodel's Key/Blob Stress-Test Tool

(seeking/avoiding) experiments. Current server processors are roughly four times faster when using the same number of threads, and an order of magnitude faster when using all cores. Interestingly, the BlueField-2 outperforms the data-parallel KNL processors, which were employed in the previous generation of HPC platforms and had known performance limitations [66].

4.5.   Integration Challenges

We faced two integration challenges when combining Arrow and Faodel together to create our environment for hosting data management services on SmartNICs. First, small portions of Arrow and Faodel target processor-specific features that can be problematic in heterogeneous architectures. For example, Faodel uses `tcmalloc` for memory allocations, which includes a small amount of assembly code for critical functions. Fortunately, both libraries had previously been ported to run on both x86 and Arm architectures. A larger question for us was determining whether there would be compatibility problems when working in a heterogeneous environment with different processor architectures. We conducted extensive testing to validate that data handoffs between the two architectures functioned correctly.

Our second integration challenge involved finding a means of transporting Arrow data using Faodel's native objects. Our solution has been to construct a wrapper class that enables one or more serialized Apache Arrow tables to be stored in a contiguous Faodel data object that is suitable for RDMA network transfers. This class includes function that call Arrow's IPC serialization functions to convert between the two representations. A powerful feature of Arrow is that there is minimal overhead to convert a serialized table into an in-memory form when working with uncompressed data, as Arrow simply creates pointers into the original IPC data during deserialization. As discussed in Section 7.2.1 this trait results in a constant deserialization time for data and is particularly useful in situations where a consumer needs a way to quickly read a dataset.

65

4.6.   Summary

A key challenge in integrating SmartNICs into a platform is creating an environment where users can easily map their work to available hardware. For this project we focus on a service-based approach to connecting different applications in a workflow and define compute and communication requirements that we expect from our environment. We focus on Apache Arrow for resolving our computational needs because it leverages a robust, tabular data model that can be applied to multiple applications and is being actively developed by a large number of people in the open-source community. For our communication needs, we have selected Faodel because it is based on primitives that make it easy to create distributed data flows on HPC platforms. We note that other communication libraries such as Mochi would also be applicable in this space.

5.   LEVERAGING COMPRESSION HARDWARE

Data compression is important in data-intensive applications because it reduces the amount of data that needs to be transmitted through the network, cached in memory, and stored on disk. Most big data I/O libraries (e.g., Avro [67], Parquet, ORC [68], and Arrow IPC) feature built-in compression support for a variety of codecs. As such, any application that processes this data must be capable of decompressing and compressing the data in a manner that is compliant with the library's data format.

The BlueField-2's compression accelerator can efficiently compress and decompress data using the DEFLATE algorithm [69]. DEFLATE is widely used and is a key part of standards such as PNG [70], HTTP [71], TLS [72], and SSH [73]. The BlueField-2's hardware implementation of DEFLATE is compatible with the zlib library, which means that when needed, data compressed by the hardware can be decompressed by software and vice versa. This interoperability is critical in data management services because it enables data products to be consumed by tasks anywhere in the system. The BlueField-2's compression hardware is currently accessed through the Data Plane Development Kit (DPDK) [74, 75], which is a library for constructing high-performance data-plane applications on top of a variety of network hardware devices. The compression hardware is designed to process a stream of individual data packets in an efficient manner and includes DMA hardware to facilitate the movement of data between the accelerator and memory.

From our experiences, it is difficult to stream application data through the DPDK compression API in a way that achieves peak performance. In this chapter we discuss a software library named *Bitar* that we constructed to maximize throughput of the compression hardware when working with Apache Arrow data. Performance experiments involving three reference particle datasets confirm that the hardware offers significant speedups over software.

5.1.   Accessing the Compression Hardware

The BlueField-2 SmartNIC features multiple hardware accelerators that are attached to the Arm complex of the SoC. These units are referred to as Generic Global Accelerators (GGAs) and employ a memory-to-memory interface that streams a block of data from a source address through an accelerator unit and back out to a destination address.

The BlueField-2's compression hardware can be accessed through the Data Plane Development Kit (DPDK) library. Unfortunately, this library is highly tuned for network operations and is organized around a packet-processing model that can be cumbersome for other types of applications. We faced several challenges in adapting DPDK's compression functions to process our Arrow data. First, individual data packets have a maximum size of 64KB. To compress larger

amounts of data, developers must split input and output buffers into packet-sized segments and then generate a packet that contains a list of compression commands for processing each segment. Second, converting between contiguous and segmented data representations can result in extra memory allocations and copies that disrupt the throughput of the data flow through the compression hardware. Optimizing the pipeline requires a detailed understanding of both DPDK and the hardware, and is tedious for users that simply want to (de)compress large blocks of data. Third, embedded hardware environments have limited resources. Therefore, recycling resources after each compression operation (while still managing errors) is extremely important. Finally, a single Arm CPU core may not be sufficient for maximizing the performance of the compression accelerator. As such, it is valuable to construct a pipeline that pre-allocates memory and divides work among cores as needed.

5.1.1. *Implementation: Bitar*

To simplify accessing the compression hardware for data compression, we implemented the *Bitar* [76] library on top of DPDK and Arrow. Bitar provides a convenient (de)compression API and features zero-copy processing, synchronous and asynchronous operation, and multicore/multidevice support. It is specifically designed to operate without root privileges, which is uncommon in DPDK-based applications. Bitar also allows users to access the BlueField-2's compression hardware from either the host's or BlueField-2's processors.

5.2.   Reference Particle Datasets

As a means of exploring the performance characteristics of the compression hardware with scientific data, we obtained three large particle datasets from different sources:

- **TrackML Particle Tracking Challenge** ("Particles") [77]: CERN supplied a particle simulation dataset for a machine learning competition hosted through Kaggle in 2018. This dataset contains 10 numerical fields per particle.

- **OpenSky Network** ("OpenSky Planes") [78]: The OpenSky Network collects worldwide ADSB information about the global positions of airplanes from thousands of volunteers with radio receivers. An individual entry in this dataset is defined by 17 fields that are a mix of numeric and text data.

- **NOAA Maritime** ("Ships") [79]: NOAA provides historical AIS position data for ships near the US coastline. Daily data was converted to a particle format that contained 17 fields composed of a mix of numerical and string values.

Raw data was converted to an Apache Arrow format using `PyArrow` and stored in a Parquet file format. Given that the BlueField-2 SmartNIC operates with 16GB of DRAM we set a 1GB limit for the size of uncompressed data to use in our experiments. We decompressed each dataset, selected the number of rows that would be closest to 1GB in size, and then recompressed the data to serve as input to the experiments.

5.3.   Experiments

As a means of evaluating the value of the BlueField-2's compression hardware we constructed experiments to answer three fundamental questions: How much of an impediment is software-based compression when working with serialized Apache Arrow data? How fast can software- and hardware-based methods compress/decompress data in a threaded environment? How does the compression ratio of the Bitar implementation compare to software-based methods? All experiments in this section were carried out on a CloudLab [80] host that has two AMD EPYC 7542 CPUs (a total of 64 cores), 512GB of DDR4 memory, and a BlueField-2 SmartNIC connected with PCIe 4.0 x16 lanes. Each experiment was run on all three reference datasets with a maximum outstanding data window size of 160MB due to memory constraints imposed by DPDK and the pipelined nature of the compression hardware.

Since Bitar has not yet been fully integrated into Arrow, our experiments compress Arrow tables differently depending on whether software- or hardware-based compression is measured. The software-based approach relies on Arrow's existing compression mechanisms, which serialize and compress each column independently before writing the final output buffer (i.e., "inner compression"). In contrast, the hardware-based approach serializes the entire table and then streams the data through the compression hardware (i.e., "outer compression"). While the former is preferred, the latter is sufficient for network transfers. Furthermore, comparing the performance of these approaches can help determine the benefits of integrating hardware compression into Arrow.

5.3.1.   *Software Compression Overhead for a Single Thread*

Our first research question focuses on whether software-based compression overhead is significant enough to justify hardware acceleration. To answer this question, we constructed an experiment that measures the amount of time required for a single thread to serialize and deserialize Arrow data in software using different codecs. We intentionally excluded the memory allocation time in this experiment given that it can be preallocated using historical knowledge of output buffer sizes.
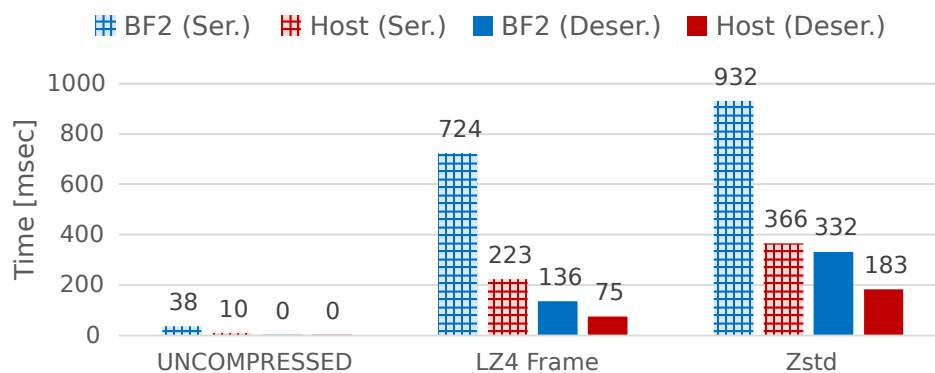
Figure 5-1. Single-thread Serialization/Deserialization Time for Different Codecs

69

The timing results presented in Figure 5-1 for the "Particles" dataset indicate that (de)serialization in Arrow *without* compression is efficient, thanks to the zero-copy buffer design of Arrow's IPC format. However, involving either LZ4 Frame or Zstd compression introduces significant CPU overhead and increases time consumption by one to two orders of magnitude. For example, serialization without compression on the host takes 10 milliseconds, while adding LZ4 Frame compression to the serialization increases the time to 223 milliseconds. We observed similar results using the other two reference datasets. Given that compression is a significant impediment to performance, we conclude that acceleration is worthwhile in performance-sensitive applications.

5.3.2.   Throughput in a Threaded Environment

Our second question focuses on how well the software- and hardware-based compression methods perform in a threaded environment. One advantage of Arrow is that it automatically parallelizes the packing and unpacking of tables by dispatching each column's work to its own thread. In Bitar's case, multiple threads can be used to maximize the amount of work supplied to the compression hardware. Since (de)compression is part of the (de)serialization process in Arrow, we conducted experiments to observe how the (de)serialization throughput improves when scaling (de)compression to use an optimal number of worker threads.
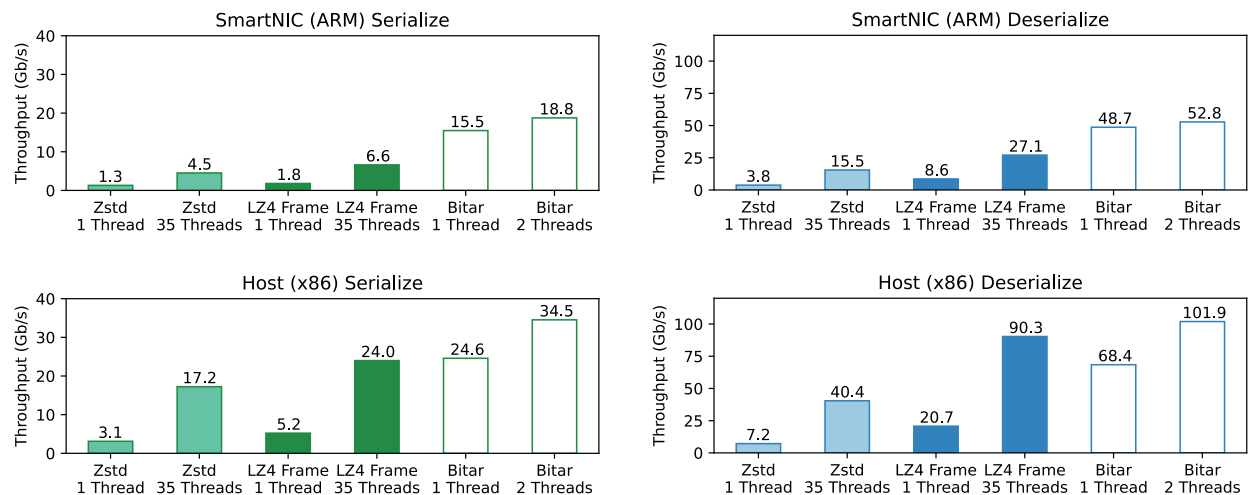
Figure 5-2. (De)serialization Throughput with Different Codecs and Degrees of Parallelism

Figure 5-2 shows the throughput measurements for the "OpenSky Planes" reference dataset. Without limiting the number of threads in the experiment, both LZ4 Frame and Zstd used 35 threads during compression and decompression. In contrast, the hardware compression throughput with Bitar was maximized when using only two threads, as we did not see higher throughput with more threads. Note that, due to the slower memory subsystem of the SmartNIC, the serialization throughput with Bitar on the host is higher than that on the SmartNIC. In general, for this dataset Bitar outperformed software-based compressions in all cases.

70
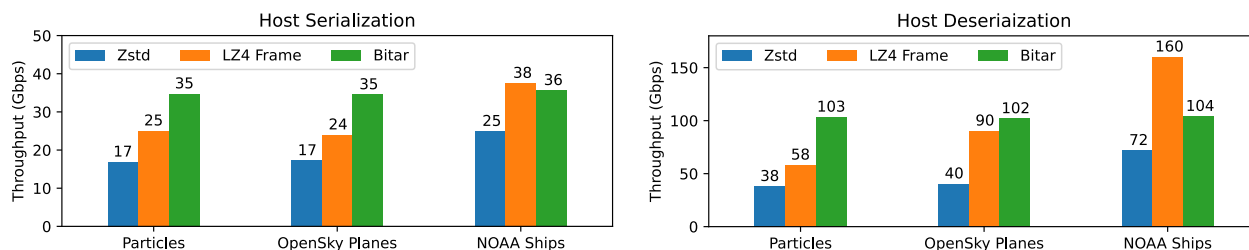
Figure 5-3. Maximum Throughput Performance on the Host for All Three Datasets

The maximum throughput on the host with different codecs for each of the three datasets is summarized separately in Figure 5-3. To better illustrate the advantages of using the hardware accelerator for (de)compression, we list the (de)serialization speedup with Bitar in Table 5-1 and 5-2. For compression with a single thread on the host, serialization with Bitar can achieve between 4.6-8.6x higher throughput than serialization with software-based compressions depending on the codec and dataset used. For compression with multiple threads, the use of Bitar can speed up the serialization throughput on the host by 1-2x.

Table 5-1. Serialization Speedup with Bitar on the Host

	Particles	OpenSky Planes	NOAA Ships
LZ4 Frame (single thread)	4.61	4.71	4.71
Zstd (single thread)	7.55	7.89	8.58
LZ4 Frame (multiple threads)	1.39	1.44	0.95
Zstd (multiple threads)	2.06	2.00	1.43

For decompression with a single thread on the host, using Bitar can speed up throughput by 3.3-10.8x. For multithreaded decompression, Bitar outperformed Zstd in all cases, but was observed to fall behind LZ4 Frame in the case with a wide dataset that loaded in many columns (i.e. 19). This is because the wider the dataset is, the more cores it can leverage during the (de)compression phase. However, since deserialization with Bitar can already achieve greater than 100 Gbps throughput (the maximum network speed), the marginal increase in throughput for the software-based approach is minimal considering the heavy load placed on the CPU cores.

Table 5-2. Deserialization Speedup with Bitar on the Host

	Particles	OpenSky Planes	NOAA Ships
LZ4 Frame (single thread)	4.46	3.30	4.59
Zstd (single thread)	10.84	9.51	10.20
LZ4 Frame (multiple threads)	1.78	1.13	0.65
Zstd (multiple threads)	2.72	2.52	1.45

Conservatively speaking, based on these results, the throughput of the compression accelerator rivals that of a software implementation that consumes all the cores of a modern CPU socket. For example, although Bitar's performance is lower than that of LZ4 Frame with 42 threads in the case of testing with the "Ships" dataset, it is greater than the same codec's performance with 35 threads when testing with the "OpenSky Planes" dataset.

5.3.3.   *Impact on Compression Ratio*

Our third question focuses on quantifying how the compression ratio changes when switching between different configurations of the software- and hardware-based compression methods. The compression ratio is computed by dividing the compressed IPC buffer size for a particular configuration by the uncompressed IPC buffer size. We expect the ratio to change in the Bitar hardware implementation because (1) a different compression algorithm is used and (2) the implementation applies compression on the entire table instead of individual columns.

The compression ratios for different configurations are presented in Figure 5-4. Results listed for Bitar are presented for one and two threads to illustrate that splitting the work into multiple threads does not have a significant impact on output size. The hardware-based compression using the DEFLATE algorithm provides a compression ratio that is between that of the LZ4 frame and Zstd codecs in all three datasets. These measurements confirm that offloading computations to the BlueField-2's compression accelerator does not result in a significant sacrifice in the compression ratio.
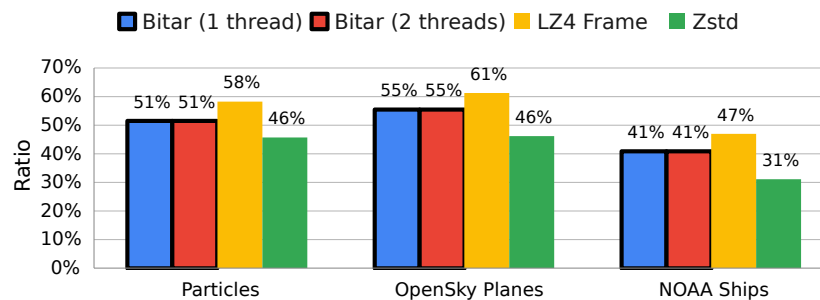
Figure 5-4. Compression Ratios for Different Compression Approaches. Black borders indicate hardware-accelerated results.

5.4.   Discussion

These performance results reveal that general-purpose CPUs are not particularly efficient in (de)compression tasks as the single-thread performance is far lower than that accelerated by compression hardware. Moreover, (de)compression using general-purpose cores cannot effectively scale the performance with the degree of parallelism. In the database arena, recent applications have begun to advocate the use of specialized storage devices that can perform

transparent (de)compression to optimize throughput and latency [81, 82]. We believe that similar efforts should be made to improve the performance of in-transit data processing. That is, instead of occupying an entire modern CPU socket to gain optimal (de)compression performance, applications can benefit more from running complex logic on these general-purpose cores and offloading compression tasks to hardware accelerators deployed along the data path. For distributed data analytics, having the ability to (de)compress data at near network speeds and with only a fraction of the system's available compute cores is essential for streaming data across nodes.

5.5.   Summary

While reducing the size of in-flight data may be valuable to many services, developers often avoid compression due to software overheads and added operational complexity. Apache Arrow resolves the latter by including compression as part of the serialization process and performing per-column compression to spread work across multiple threads. In this chapter we have described how the Bitar library enables us to leverage the BlueField-2's DEFLATE accelerator to compress Apache Arrow data. Using only two threads Bitar can (de)compress Arrow data faster than a software-based version running with 35 threads. In studies with real-world particle data, Bitar achieved similar compression ratios to the current codecs included in Arrow.

6.   REORGANIZING DISTRIBUTED DATASETS

HPC simulations operate on massive datasets that are distributed across hundreds to thousands of compute nodes. For efficiency, it is imperative that a workflow retains this parallelism as much as possible when migrating data between workflow stages and executing required data transformations. The limited compute and memory resources of SmartNICs only amplify this urgency. As such, offloading data management services to SmartNICs requires us to consider *parallel* services that not only move data between host applications and SmartNICs, but also between SmartNICs that work together on a common task.
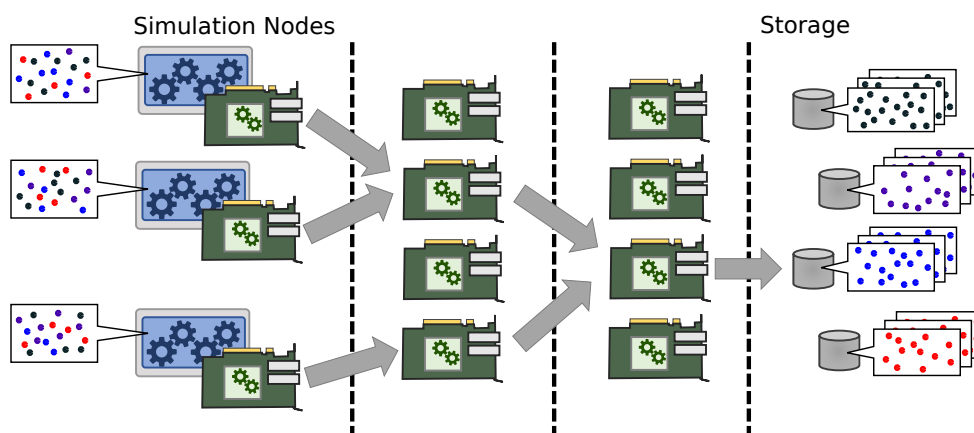
Figure 6-1. An Array of SmartNICs Reorganizes Simulation Results in Multiple Stages

In this chapter we explore an example of how an array of distributed SmartNICs can work together to reorganize simulation results into a form that is easier for analysis applications to consume. As illustrated in Figure 6-1, an array of SmartNICs implements a multistage data flow that gradually sorts a particle dataset until all related items are hosted at the same SmartNIC. Sifting behavior can be controlled by passing configuration information to the service at start time. We discuss design details for this particle sifting service and present performance measurements from experiments conducted on 100 nodes of the Glinda cluster.

6.1.   Reorganizing Particle Data

Scientists employ Particle-in-Cell (PIC) methods [83] in simulation codes to model a wide variety of phenomena [84, 85]. PIC codes track the discrete state of billions of particles as they move about and interact with a model of a physical environment. The sheer size of the particle data precludes users from writing continuous snapshots to disk or maintaining more than a single time step of data in the simulation's memory. The ability to rapidly sample and export sizable portions

of this data would provide users with an opportunity to apply external analytics in a workflow to inspect how the state of individual particles evolves over time. One of the obstacles in exploiting this data, however, is reorganizing it from the simulation's perspective (i.e., *temporal snapshots*, where data is sorted by time step and simulation rank) to a form that analysis applications can leverage (i.e., *particle tracks*, where data is sorted by particle ID and time step). It is therefore useful for a workflow to include data management services that can transform the data from one representation to another.

(a) Temporal Snapshot of Positions   **(b) Particle Tracks**
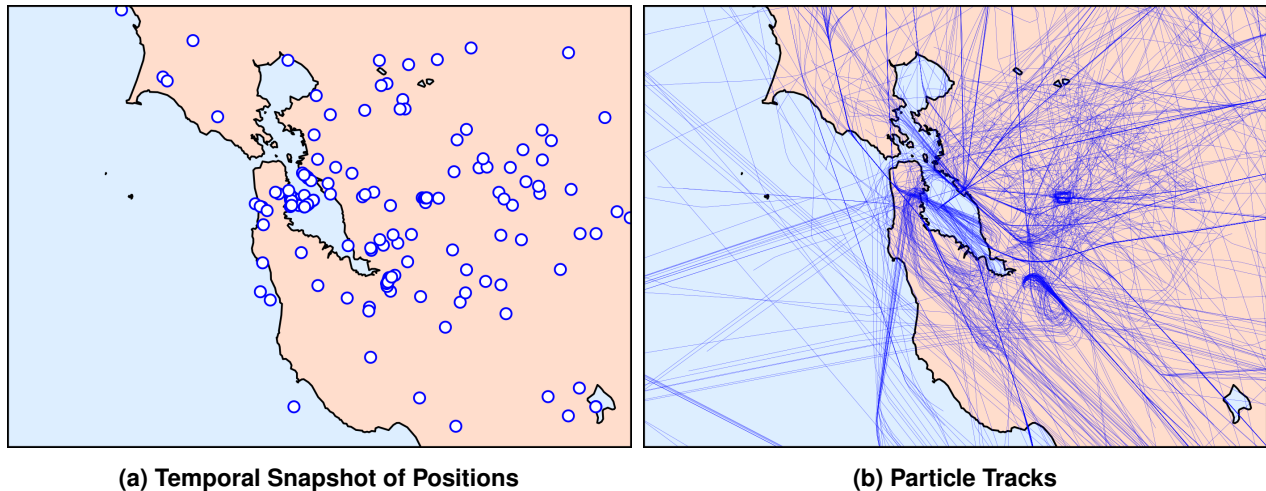
Figure 6-2. Analytics may Require Particle Data to be Adapted from Temporal Snapshot to Particle Tracks

The challenges associated with refining large particle datasets can be illustrated with an example that uses the OpenSky Planes dataset discussed in Chapter 5. Consider a system that tracks a large number of particles (e.g., airplanes) as they move about a geographic region, as depicted in Figure 6-2(a-b). Each particle maintains a small amount of state information (e.g., airplane ID, type, position, velocity, and tailwind speed) that is updated every time step. While examining temporal snapshots of position data (a) may yield insight into particle densities for key areas and the typical spacing between particles, looking at the trajectories of a subset of particles over time (b) reveals more about the rules governing their behavior. Performing this conversion in real systems is challenging for two reasons. First, systems that track large numbers of particles are often memory constrained and require users to distribute data across many compute nodes. Services that reorganize this data must be designed in a distributed manner that does not result in individual nodes being overloaded with data. Second, particles migrate between compute nodes in these systems as time progresses due to their movements in space or by load balancing in the system. As such, assembling a single track may involve contributions from multiple compute nodes over time.

In response to these challenges, we define multiple requirements for building a particle sifting service. First, the service must be implemented in a distributed manner that spreads the data and work across available resources to ensure efficient execution and memory utilization. Second, processing elements (PEs) must be able to accumulate data and operate asynchronously to allow the system to react to dynamic runtime characteristics. Finally, the service must minimize the amount of time required for a simulation to inject a new wave of data.

6.2.   Distributed Particle Sifting Service Implementation

We constructed software on top of Faodel and Arrow to implement a multistage sifting algorithm that uses a collection of SmartNICs (or hosts) as PEs in a linear pipeline. As illustrated in Figure 6-3, simulation ranks sample particle data for the current time step and inject a copy of it to the PE hosted at the local SmartNIC. Once a user-defined accumulation threshold is crossed, the PE performs a *compaction* operation. During compaction, the PE (1) splits all of its accumulated data into smaller objects based on bits in each record's particle ID field and (2) transmits each output object to its corresponding PE in the next stage of processing. Particles become more sorted as they move through each of the stages.

While PEs can be mapped to any physical SmartNIC or host in the system, it is expected that multiple, neighboring PEs will exist at a single location to reduce communication costs. The actual steering of data between PEs is managed through a combination of a key-labeling scheme and the use of Faodel pools to determine where data is routed. The key-labeling scheme concatenates the next stage's ID and the currently-matched particle ID bits to pick a unique destination for the data. Additional source information is encoded in a separate portion of the key to avoid collisions with the data from other PEs.
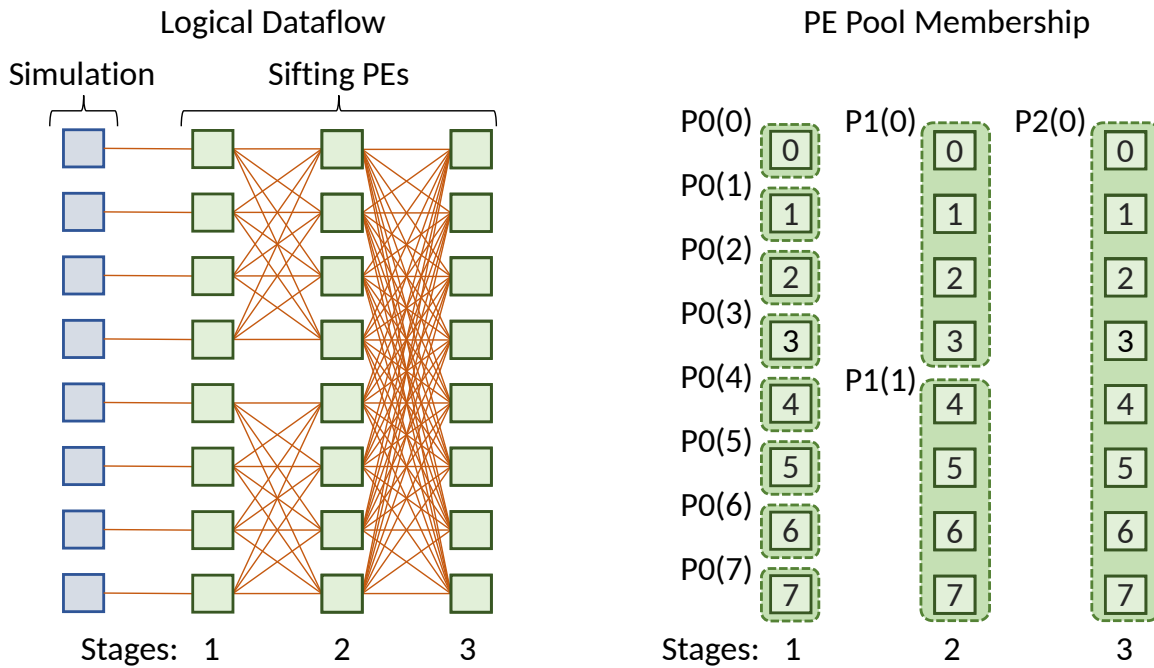
Figure 6-3. Data Flow and Placement for Sifting Particle Data

Multistage sifting systems with low PE fanout and high numbers of compute nodes can easily result in a few nodes in the system becoming overwhelmed with all the simulation's data. To mitigate this problem, we use Faodel's pool abstraction to limit the number of nodes to which a PE can distribute data. At start time, software generates a collection of pools in the cluster that correspond to where different PEs reside. For example, the network depicted in Figure 6-3 shows three stages and PEs that can split each object into four possible outputs. The 6th PE in stage 1

uses pool "P1(1)" to route to four possible destinations, while the 6th PE in stage 2 uses "P2(0)" to route to eight possible destinations.

6.2.1.   Injecting Particles to the Local SmartNIC

The first step in the particle sifting service is for each host in the simulation to sample its current data and create a serialized Arrow object on the local SmartNIC. While there are multiple means by which this task can be accomplished, our initial approach simply relies on the host to convert the data to an Arrow format, serialize the data to a Faodel object in host memory, and then use Faodel to transmit the object to the SmartNIC. We constructed a benchmark to quantify injection overheads and varied the input data sizes from 1M–64M particles (37MB–2.4GB). Performance measurements are presented in Figure 6-4. While the conversion and serialization portion of this process could achieve rates of up to 5.24GB/s, Faodel's single-object transfer performance of 1.77GB/s resulted in the overall injection rate being 1.32GB/s.
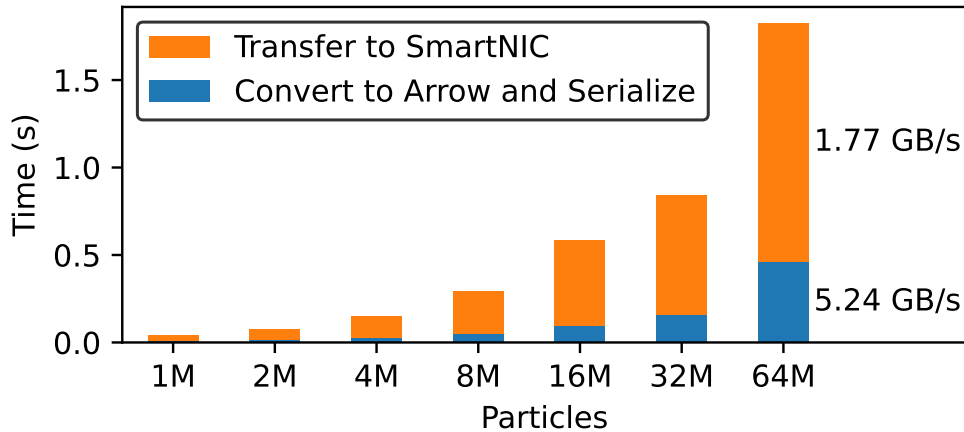
Figure 6-4. Data Preparation and Injection Overhead

There are two significant shortcomings of this approach that hinder performance. First, the data flow operates in a store-and-forward manner that does not allow for any kind of pipelining or overlap. A better approach for a system involving network transfers would be to split the input data into smaller pieces to allow multiple transfers to take place at the same time. Second, there are two allocations along the critical path: one on the host to serialize the data and another on the SmartNIC to receive the data. Consolidating the allocations and performing them in advance can help streamline the data flow. We examine optimizations for injecting data into SmartNICs in greater detail in Chapter 8.

6.2.2.   Partitioning Particle Data

The second step in reorganizing the particle data is for a PE to perform a compaction operation that partitions a collection of particles into smaller, related pieces that can then be transmitted to

the next stage's PEs. The partitioning algorithm examines a table and uses a small number of bits in the particle ID field to determine which output table should hold each particle. We implemented the partitioning as a multistep algorithm that executes a select query to generate each output table via Apache Arrow. While far from ideal, this approach is acceptable in the particle sifting service because of the low-fanout requirements of the distributed algorithm.
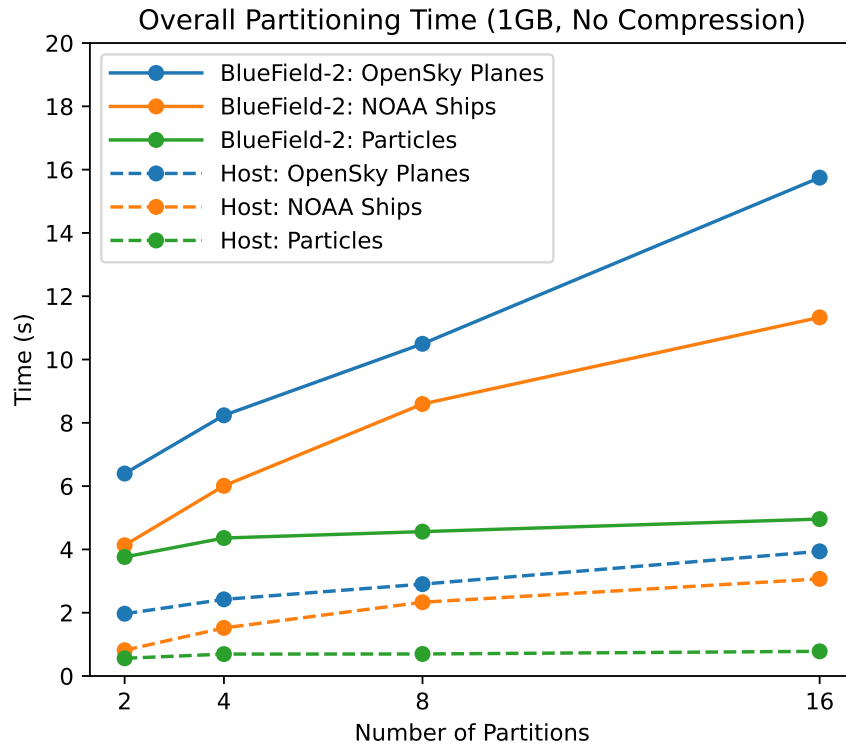
Figure 6-5. Overhead for Partitioning without Compression

Performance experiments were conducted on a Glinda compute node that features a 32-core AMD EPYC 7543P processor and a BlueField-2 VPI card. In the first experiment, we measured the overall amount of time required for the host or SmartNIC to unpack, partition, and repack the particle datasets from Chapter 5 using different compression algorithms implemented in software. The partitioning algorithm split an input table into 2 to 16 output tables based on 1 to 4 bits of the particle ID. As depicted in Figure 6-5, the host operates roughly four times faster than the BlueField-2 when processing uncompressed data. Increasing the number of partitions increased the processing time in most cases. A closer inspection of the "Particles" dataset revealed an ID address space issue that resulted in a distribution imbalance. These issues can be mitigated by hashing the ID or selecting ranges that are more meaningful to the application.

The second experiment examines the impact of Apache Arrow's built-in software compression mechanisms on performance. These tests vary whether the input and output objects are serialized with no compression, LZ4 Frame compression [86], or Zstd compression [87]. Figure 6-6 provides the timing breakdowns for unpacking, partitioning, and repacking 1GB of particle data when performing a 4-way split. As expected, uncompressed data is significantly faster to read than compressed data because Arrow can simply create pointers into the original serialized data and avoid copies. Repacking the data, however, is similar in all cases. This overhead highlights the fact that serialization by itself is an expensive operation.
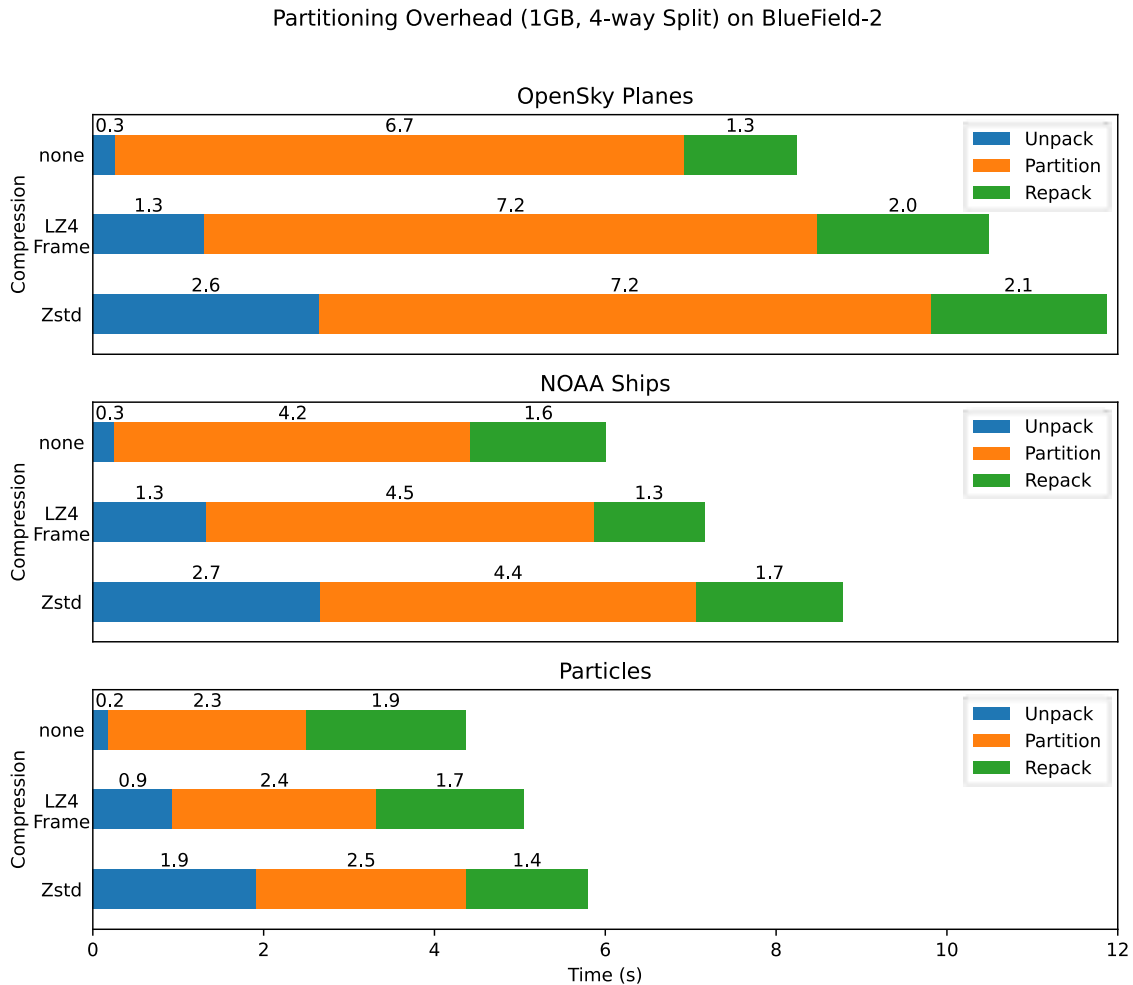
Figure 6-6. Timing Breakdown for a 4-way Split on the BlueField-2 with Software-Based Compression

Examining the output sizes of the individual, serialized partitions generated in the second experiment provides greater insight into how partitioning affects compression results. Figure 6-7 provides a breakdown of how large each output partition is when using Zstd compression and the lowest 1 to 4 bits of the particle ID to split the three input datasets. In the OpenSky Planes dataset, the lower bits of the ID are diverse and yield equally-sized output partitions. There is a slight decrease in the aggregate size of the output data as the number of partitions increases because the individual partitions have more data redundancy that the compression algorithm can exploit.
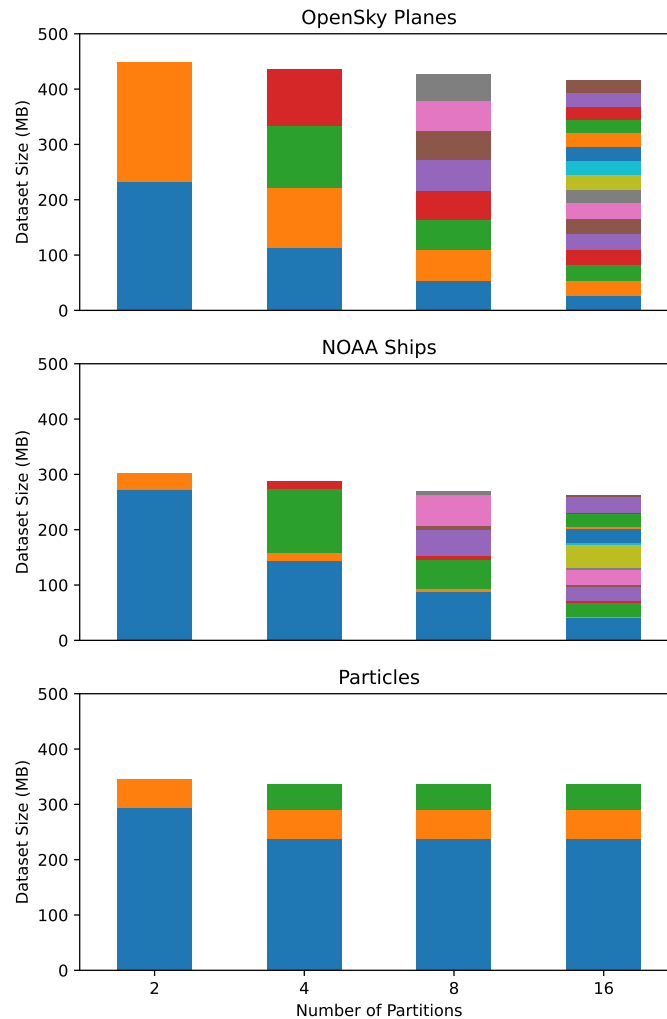
Figure 6-7. Aggregate Dataset Sizes when Varying the Number of Partitions and Compressing with Zstd

In contrast, the NOAA Ships and the Particles datasets have less diversity in the lower bits of the particle ID field. As such, the partitioning algorithm splits the data into uneven portions. This property is undesirable because it may create load balancing issues with downstream consumers of this data. While the aggregate size of the NOAA Ships dataset improves as the number of partitions increases, the Particles dataset does not as its IDs can only be split into three partitions. These examples indicate that it is worthwhile for architects to understand the characteristics of their data and select partition address bits that will result in balanced outputs.

81

6.2.3.   Distribution Challenges

The initial version of the particle sifting algorithm used Faodel's built-in *distributed hash table* (DHT) pool to distribute one PE's output tables to the next stage's PEs. While this approach worked, we observed significant imbalances within different pools. Figure 6-8 illustrates how load imbalances can impact the overall performance of the system. In this four-stage example, several SmartNICs in the third and fourth stages are assigned significantly more data to process than other SmartNICs in the system. A closer inspection revealed that our key-labeling scheme for controlling routing in the merge tree generated keys with very similar names. Unfortunately the DHT's default hashing algorithm did not have enough entropy to ensure the keys would properly be distributed across the pool's members.
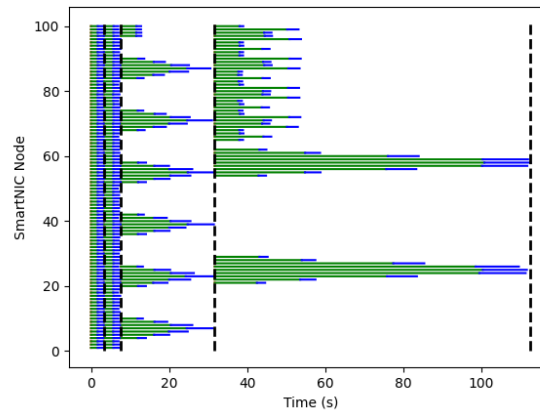
Figure 6-8. Key Hashing can Create Distribution Imbalances when Using a DHT

We solved this problem by creating a new pool type in Faodel that uses an explicit tag encoded at the end of a key to control which node in the pool is responsible for hosting a particular key. Users assign an integer value to keys to group related keys together. A new *tag-folding table* (TFT) pool extracts the value from the tag and does a modulo operator to select the node in the pool that owns the data. This strategy gives users a great deal of control over how data is placed but still uses keys that are compatible with other pool types in Faodel. The results in the remainder of this chapter are all based on the TFT pool.

6.3.   Overall Performance Evaluation through an Impulse Response

The particle sifting service is designed to operate in an asynchronous manner that bubbles data through the PEs as dictated by compaction policies. As a means of comparing the tradeoffs of different architectural configurations, we constructed an *impulse response* benchmark that injects a uniform amount of data to each of stage 1's PEs and then measures the amount of time required for each stage to perform a synchronous compaction of data. As such, this benchmark measures the overall amount of time for one wave of data to flow through the entire system and be binned in such a way that each particle ID will land on a particular SmartNIC no matter where it originated. For these experiments we varied the distribution architecture and measured the amount of time to fully distribute the data across 100 SmartNICs.

6.3.1. Distribution Architecture

The end goal of the particle sifting service is to ensure that particles are distributed in such a way that particles with the same ID for a given number of bits are binned together and that there are as many output bins as there are SmartNICs in the system. The partitioning algorithm processes k bits of the address space at each stage (i.e., 2^k splits per stage). As listed in Table 6-1, the value of k determines the minimum number of stages required to sift the data into at least 100 output bins. Increasing the number of output bins makes data in a bin more specific, but comes at the cost of increasing the bookkeeping the system must perform to track all the different bins.

Table 6-1. Determining the Minimum Number of Stages to Distribute to 100 Nodes

k	Splits/Stage	Minimum Stages	Output Bins		
2	$2^2 = 4$	4	$2^2 \cdot 2^2 \cdot 2^2 \cdot 2^2$	=	256
3	$2^3 = 8$	3	$2^3 \cdot 2^3 \cdot 2^3$	=	512
4	$2^4 = 16$	2	$2^4 \cdot 2^4$	=	256
5	$2^5 = 32$	2	$2^5 \cdot 2^5$	=	1,024
6	$2^6 = 64$	2	$2^6 \cdot 2^6$	=	4,096
7	$2^7 = 128$	1	2^7	=	128

Figure 6-9(a-c) depicts the different pathways for moving data between the 100 SmartNICs in three different distribution schemes. These schemes split data (a) four ways (k=2) in four stages, (b) 16 ways (k=4) in two stages, and (c) 128 ways (k=7) in one stage.

(a) 4 Stages, 4-Splits/Stage   (b) 2 Stages, 16-Splits/Stage   (c) 1 Stage, 128-Splits/Stage
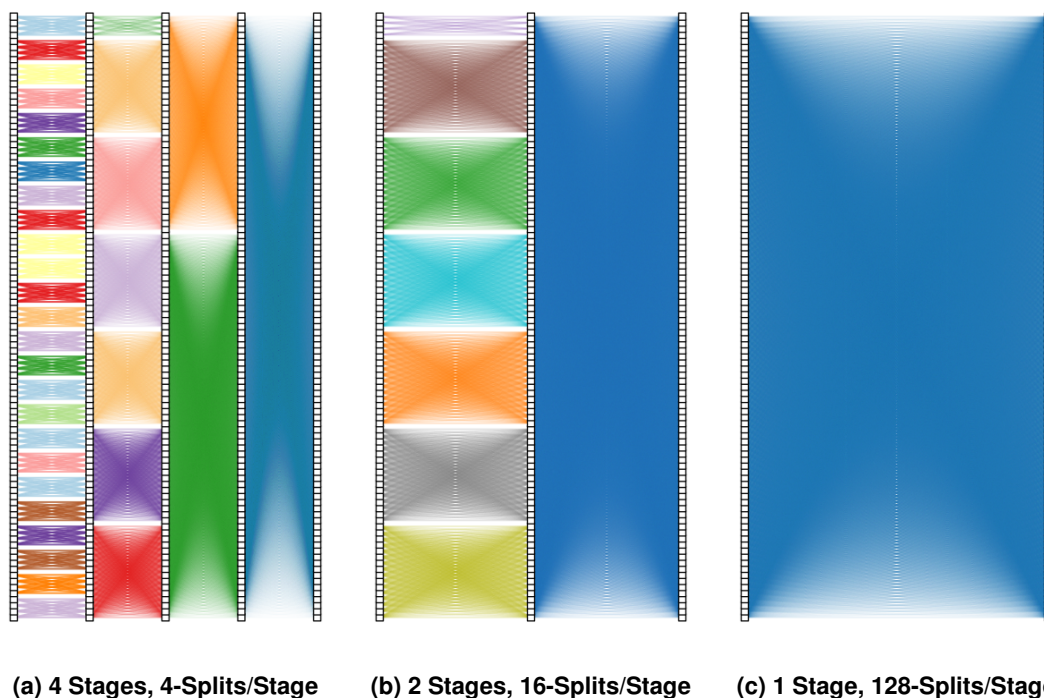
Figure 6-9. Particle Distribution Strategies

6.3.2. Overheads for Different Split Sizes

Figure 6-10 presents the split and publish timings required to process 100M particles on 100 SmartNICs. While performing 128 splits allows the work to be completed in a single pass, doing so is slightly slower than doing 4-way splits over 4 stages of work. Our experiments indicate that 16 splits per object yielded the best solution for the SmartNICs. In most cases, split time was more expensive than the publish time. Overall, the current implementation provides a relatively uniform distribution of work and data across the nodes.
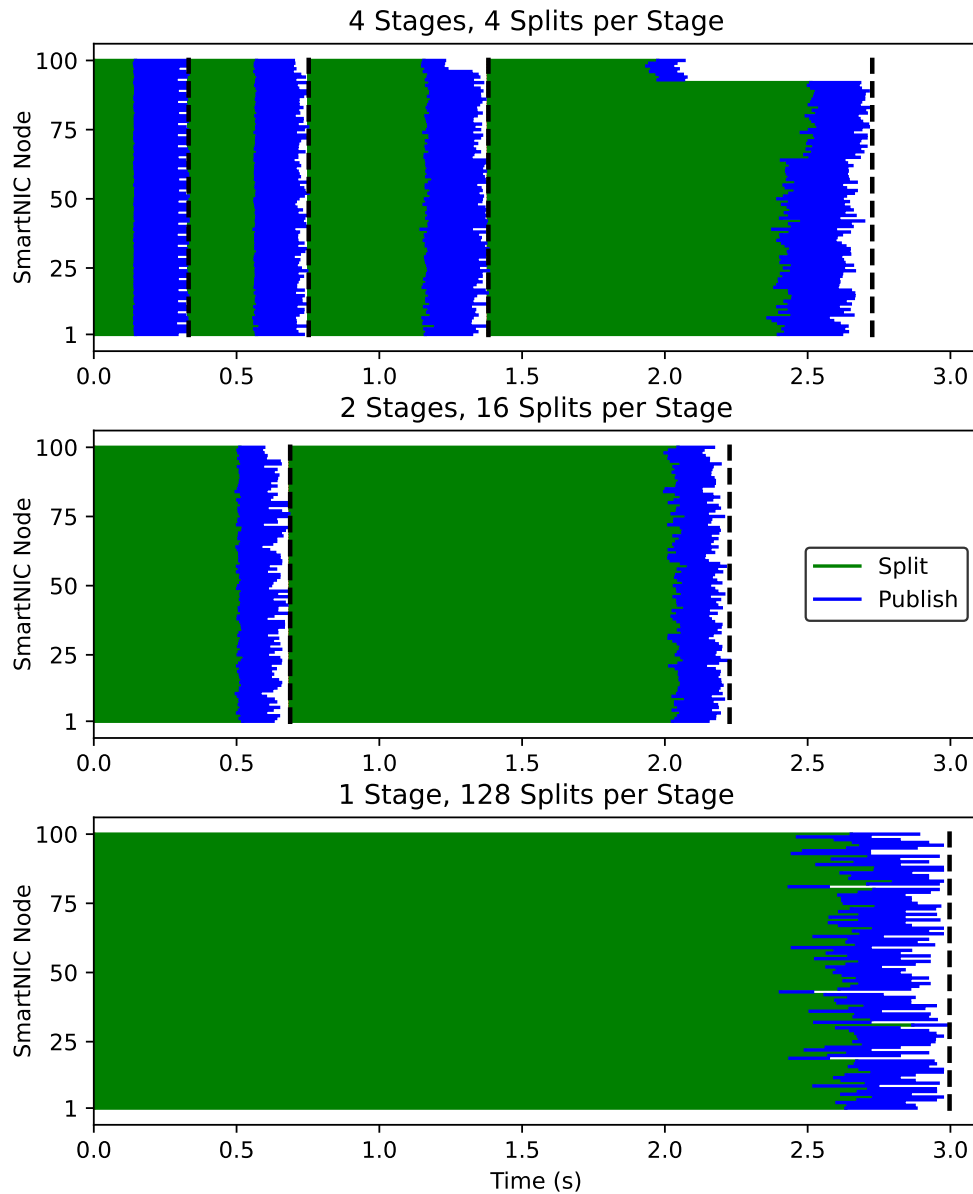
Figure 6-10. SmartNIC Sifting Time for 100M Particles

Reducing first-stage overhead is important as it makes the sifting network more responsive to injected data. We repeated the previous experiment on 100 EPYC 7543P Zen3 server nodes to measure the first-stage performance for a range of splits. As depicted in Figure 6-11, the 32-core host processors were roughly four times faster than the 8-core Arm processors.
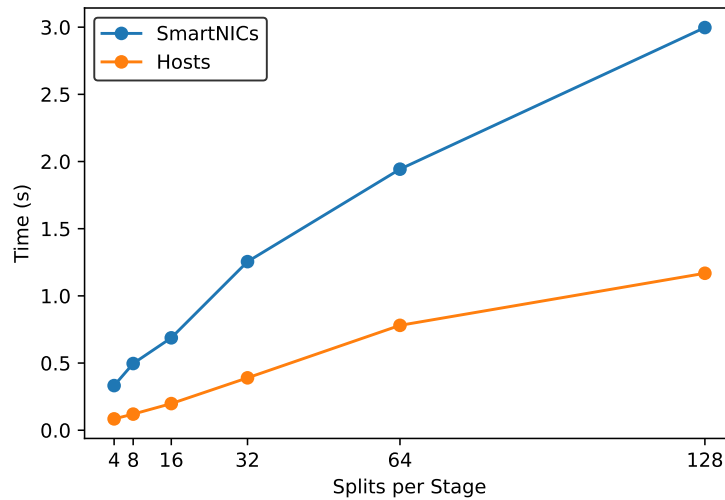
Figure 6-11. First Stage Overhead for 100M Particles

6.3.3.   Scaling Particle Dataset Size

In the final set of measurements, we conducted impulse response tests for 10M, 100M, and 1,000M particles. The overall sifting times for 100 SmartNICs and 100 host systems are presented in Figure 6-12. Performance scaled linearly in both cases. The host systems were again roughly four times faster than the SmartNICs.
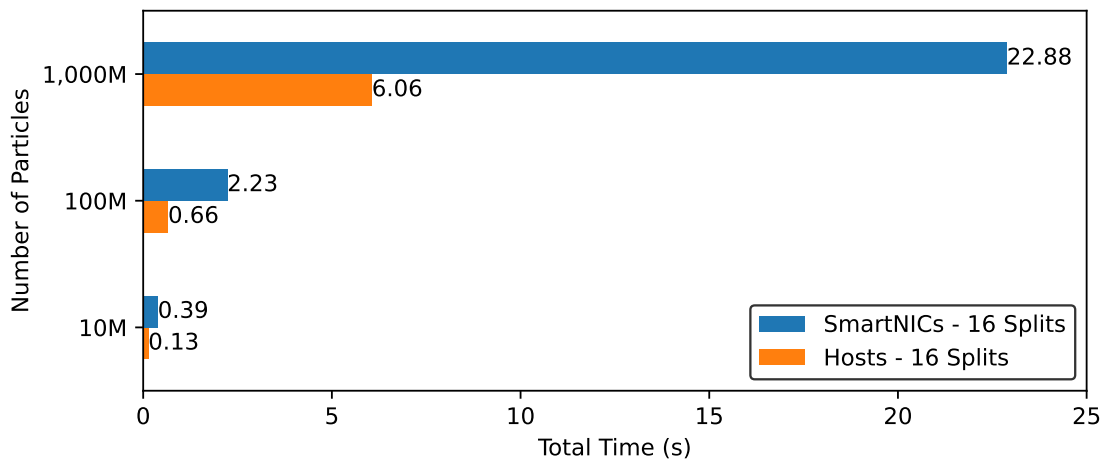
Figure 6-12. Total Sifting Time for Different Input Datasets

6.3.4.   *Discussion*

Implementing the partitioning operation with Apache Arrow highlighted its development advantages. Arrow's well-reasoned data primitives and existing support for serialization, compression, and processing greatly simplified the implementation effort. Our implementation worked with multiple datasets without modification, even though each dataset had different data components and ID bitwidths. Although the current version of Arrow does not have all the primitives of a higher-level library such as Pandas [88], it contains adequate primitives to implement a variety of operations.

In terms of raw performance, the hosts are noticeably faster than the SmartNICs at sifting the particle dataset in a distributed manner. However, there are multiple scenarios where lower performance is acceptable, such as when time step snapshots take place infrequently or host memory is highly constrained. In these examples it is valuable for the host to be able to rapidly pass data to the SmartNIC, reclaim memory, and return to the simulation.

6.4.   Summary

The particle sifting services provides an example of how a collection of distributed SmartNICs can offload a data processing task from the hosts and add value to a workflow. SmartNIC memory is used to stage result data and make asynchronous progress towards reorganizing data to a format that is better aligned for analytic consumption. Faodel is valuable for this work because it provides a basic set of primitives for moving data between groups of SmartNICs. The ability to change the behavior of the system by supplying a configuration with different pool definitions enabled us to fine-tune the implementation without having to rebuild the software. This work also demonstrates that Apache Arrow is particularly useful for executing operations on in-transit data. Arrow defines in-memory and on-wire representations of data and compute primitives for transforming the data that automatically map to data-parallel hardware. While the current SmartNICs are approximately four times slower than the hosts at reorganizing the particle data, they are appropriate in situations where the host infrequently generates output snapshots and the goal is to minimize the amount of overhead for I/O observed by the simulation.

7. QUERYING IN-TRANSIT DATA

As demonstrated in the previous chapter, scientific computing workflows pass a large amount of intermediate data between different tools and services that run in parallel on an HPC platform. While this in-transit data may contain valuable information, workflows typically only save a small fraction of the data to disk due to capacity and performance limitations of the parallel file system. As more workloads are adapted to run in distributed network and storage devices, there is an opportunity for researchers to embed analytics in the data path and harvest new insights from in-transit data. What is needed is a mechanism that allows users to remotely inspect the dynamic content that individual devices manage.

We advocate adding a robust and flexible *query interface* to embedded devices to enable users to interact with their in-transit data. We see multiple use cases for query interfaces, including capturing live statistics about data content in a workflow, troubleshooting platform load balancing issues, verifying the correctness of workflow components, and debugging. In this chapter we focus on the design of a query interface for SmartNICs that leverages Apache Arrow's Acero library to execute query plans on in-transit data. As illustrated in Figure 7-1 this interface implements a decision engine at the SmartNIC to estimate whether it would be more profitable to execute a query locally (i.e., *push-down*) or simply return the raw data back to the client for evaluation (i.e., *push-back*). Characterizations of different parts of the end-to-end query path allow the decision engine to make predictions about query execution that would not be feasible by the client alone.
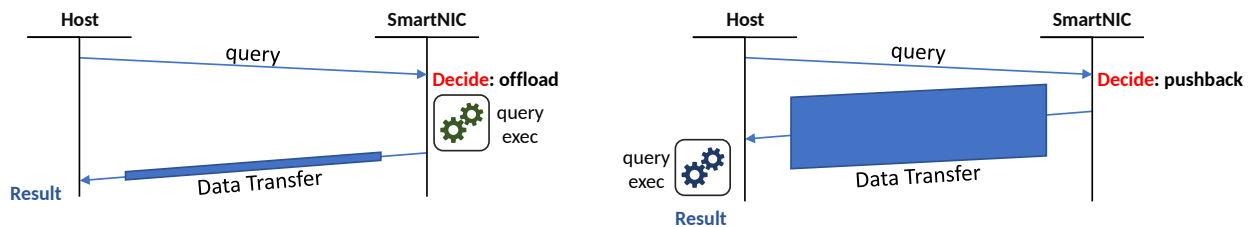
Figure 7-1. Query Execution can be Pushed Down or Pushed Back

7.1. Query Interface for In-Transit Data

The intent of our query interface is to provide a simple mechanism by which users can interactively probe a platform's distributed resources and extract essential pieces of information about the data they maintain. We define three specific requirements for this interface. First, users must be able to express a wide array of complex queries that are dynamically defined at runtime. Static approaches such as remote procedure calls (RPCs) that only allow users to define a fixed set

of queries at compile time are insufficient. Second, the query interface must have interoperability with existing tools and standards. The database and data science communities have well-established standards and extensive tools for interacting with data stores. It is therefore inappropriate to impose new paradigms or languages on users. Finally, the query interface must be designed in a way that allows the system to optimize execution based on runtime constraints. Given that complex queries may overwhelm an embedded device, it is imperative that the system be able to offload or defer work as needed.

Our approach to building a query interface for SmartNICs is to construct a traditional query pipeline that leverages existing standards and libraries as much as possible. This pipeline encompasses three steps. First, users supply a SQL query that is translated to a logical query plan by a query front-end. Second, software translates the logical query plan to a physical execution plan by a query planner and optimizer. Finally, the execution plan is delivered to the proper SmartNIC (or in the future, a collection of SmartNICs) and executed by a query engine. The main contribution that the Offloading Data Management Services to SmartNICs project makes in this area is an initial exploration in dynamically determining where execution should occur– on a remote SmartNIC or on the local host. The decision is made using the physical execution plan produced in the second step, and is discussed in more detail in Section 7.2.

7.1.1. *Software Components*

In our work, we leverage a number of open source libraries to implement remote queries:

DuckDB: DuckDB [89] is a popular, open-source analytical database that is capable of translating a SQL query into a logical query plan.

Substrait: Substrait [90] is a cross-language specification for data computing operations and a standardized format for query plans. This facilitates the transformation of logical query plans into binary substrait plans in formats like protobuf or JSON, which are suited for network transfer.

Apache Arrow: Apache Arrow provides a data standard for the environment that allows developers from multiple realms to represent, process, and query their data. It utilizes a tabular data model, which is suitable for many of our applications due to its thriving open-source community, and includes built-in compute operators with SIMD optimizations that can map to parallel processors. Most importantly, the Acero component of Apache Arrow provides semantics for constructing and executing composable logical query declarations in C++.

Faodel: Faodel's Kelpie library provides a key/blob API that enables the exchange of RDMA-transportable objects across a diverse set of communication endpoints. Furthermore, it has the ability to trigger computations on objects located at remote endpoints using user-defined functions. This feature facilitates the transmission of query workloads to SmartNICs and the return of either complete or intermediate results depending on the execution decisions made by the SmartNICs.

7.1.2.   End-to-End Data Flow

The end-to-end data flow for a push-down query is illustrated in Figure 7-2. First, a user's SQL query is converted to a binary Substrait plan. Second, Faodel's remote compute operation is used to dispatch the query to the communication endpoint that is responsible for housing the desired data. This request contains a key for referencing the desired objects at the endpoint, an identifier for a user-defined function (UDF) that processes Arrow data, and the binary Substrait plan. Third, upon receiving this request, Faodel retrieves the relevant objects from the endpoint's local in-memory store and passes them to the specified UDF. The UDF we have constructed for this project deserializes data from the objects into a single, in-memory Arrow table and then uses Arrow's Acero library to execute the Substrait plan on the table. The UDF serializes the tabular results into an object that Faodel returns to the client's remote compute operation. Finally, the object is deserialized into an in-memory Arrow table that can be consumed by the user.
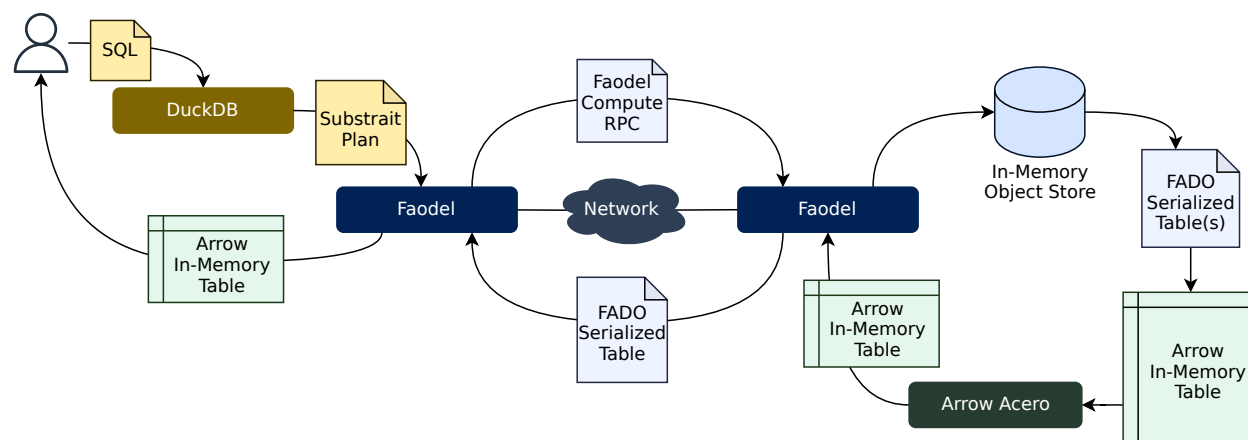
Figure 7-2. The End-to-End Data Flow for a Push-Down Query using DuckDB, Arrow, and Faodel

7.2.   Dynamic Query Decision Engine

In this section, we share our design of a *decision engine* that uses predictive scheduling to determine if a query should be offloaded (*push-down*) or delegated back to the client (*push-back*). The motivation for this work comes from the fact that queries vary in complexity and can return different amounts of data. In queries that return a small amount of data it may be beneficial to execute the query on the SmartNIC to minimize data transfers when the SmartNIC is not saturated with other requests. Conversely, in situations where a query is complex or returns a substantial amount of data it may be advantageous to defer execution to the client. A decision engine is therefore required to assess current conditions and predict the best course of action.

A fundamental design choice in this work involves determining where the decision engine should be placed in the end-to-end flow. Implementing the decision engine early on in the request at the client side simplifies development at the expense of having limited information available to assist in the decision. In contrast implementing the decision engine at the SmartNIC increases the server's complexity but enables the decision engine to take advantage of local information about

the data and the SmartNIC's current load. We advocate placing the decision engine in the SmartNIC to allow data content and runtime information to be factored into the decision.

Our decision engine generates statistics about queries, data, and platform execution times to make predictions about whether push-down or push-back execution will be faster for individual queries. The decision engine uses the Apache DataSketches library [91] to rapidly generate characterizations of in-transit data. One of the advantages of DataSketches is that it includes a streaming mode that enables statistics to be updated as new data arrives. Furthermore, this library provides parameters that enable the fine tuning of the estimation's accuracy. This feature allows us to evaluate the trade-offs between estimation performance and system resource consumption.

An inspection of the push-down and push-back data paths depicted in Figure 7-3 reveals that there are three dominant overheads that must be factored into the decision process: serialization, network transfers, and query execution. In the following subsections we describe each operation in greater detail and present the methods by which we characterize and predict overheads in the decision engine.
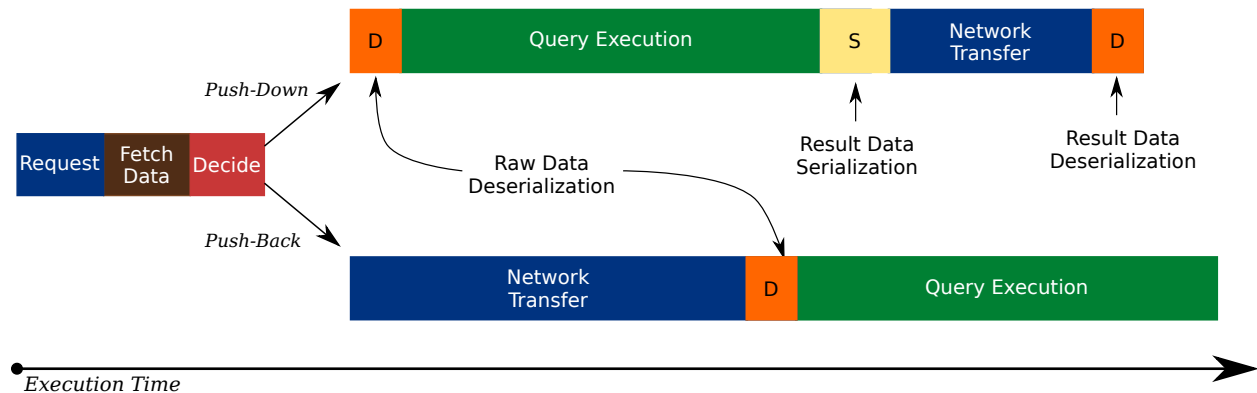
Figure 7-3. Conceptual Time Breakdown for Push-Down and Push-Back cases

7.2.1.   *Serialization Time*

Each SmartNIC in the system manages a collection of in-transit data objects in on-card memory. A data object holds one or more Apache Arrow tables that have been serialized to the Arrow inter-process communication (IPC) format. As such the data must be deserialized into an Arrow table before a query can execute, whether the work is done on the SmartNIC or at the client. Fortunately, Arrow's IPC format is designed to allow zero-copy reconstruction of an Arrow table from an (uncompressed) IPC buffer simply by establishing reference pointers into the buffer. In the case where a query is offloaded to a SmartNIC, it is also necessary to serialize the results into an IPC buffer in order to allow it to be transmitted to the client.

Characterizing: As a means of characterizing serialization overheads, we constructed a synthetic corpus of particle data tables and then measured the amount of time required to serialize and deserialize the tables on both a host system and the BlueField-2 SmartNIC. Tables employed 9 fields and ranged in length from 1 to 2^{25} rows. As depicted in Figure 7-4 (left), deserialization performance is independent of table size, remaining nearly constant for both hosts and

SmartNICs. In contrast, Figure 7-4 (right) indicates that serialization time depends on table size.
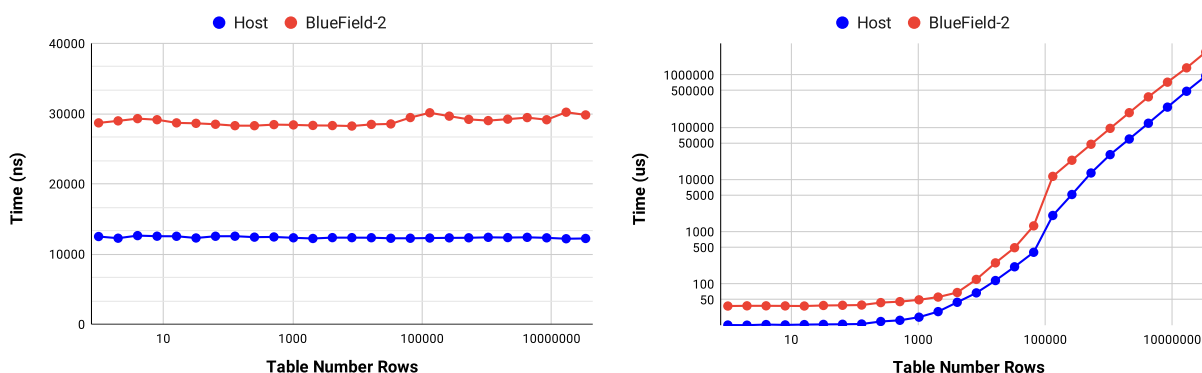
Figure 7-4. Arrow Table Deserialization and Serialization Times

Predicting: The decision engine employs different strategies for predicting deserialization and serialization overheads due to the performance characteristics of the two operations. Deserialization time is estimated by using the constants that were observed from empirical measurements. For predicting serialization overheads based on table row counts, we used our synthetic corpus to train a random forest regression model. As indicated in Figure 7-5 this model achieves a prediction accuracy with an error rate of less than 7%.
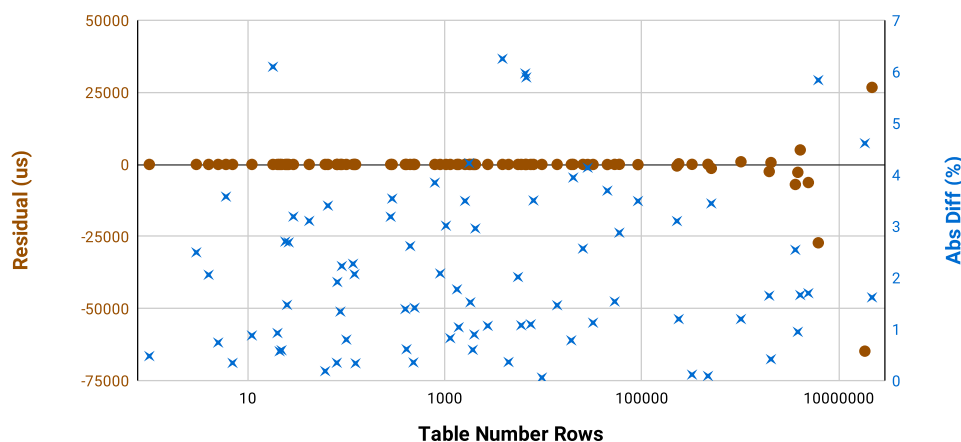
Figure 7-5. Error Rates for Serialization Time Predictor

7.2.2.   Network Transfer Time

The second significant overhead in the system is the amount of time required to transfer table data back to the client. Transfer times can be challenging to estimate because they involve multiple tasks that may not be obvious to users. For example, returning data from a UDF in Faodel involves three operations: the server sends a message with pointers to the result data; the client allocates memory for the new object; and then the client performs an RDMA pull.

91

Characterizing: To create an estimate of network overhead, we measured the round-trip time for a client to request varying sizes of data from the SmartNIC using Faodel. As shown in Figure 7-6, overhead is constant until tables are larger than 64KB.
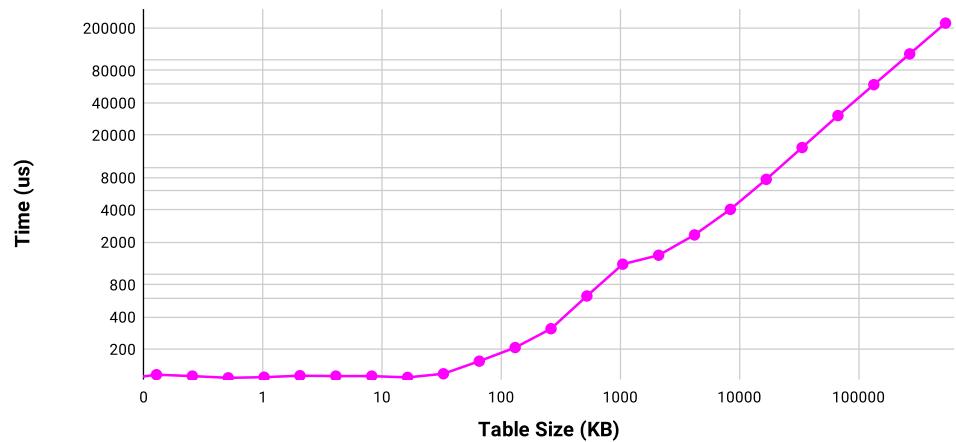
Figure 7-6. Network Transfer Times for Retrieving Objects from a SmartNIC with Faodel

Predicting: Predicting the network transfer time requires us to estimate (1) how large (in bytes) a table with a given number of rows will be when serialized and (2) how long it will take Faodel to transmit a serialized table. In both cases empirical measurements were used to train a random forest regression model. As illustrated in Figure 7-7 the model could infer the serialized size from a particle table's row count with an error rate of less than 6%. Similarly, Figure 7-8 indicates that network transfer times could be estimated from the serialized data size with error rates within a single-digit percentage.
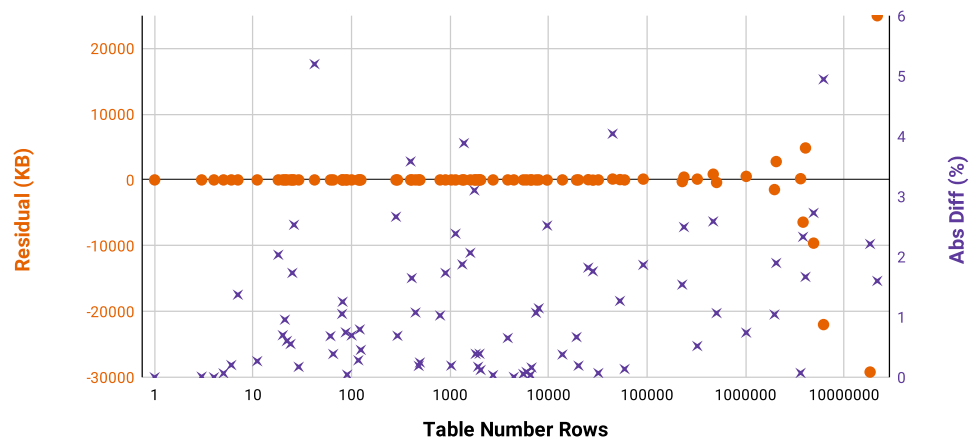
Figure 7-7. Error Rates for Serialization Size Predictor
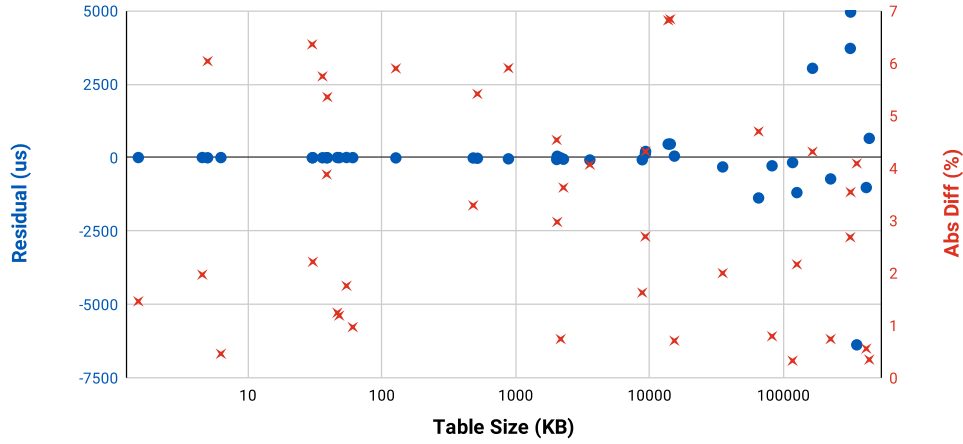
Figure 7-8. Error Rates for Network Transfer Time Predictor

7.2.3. *Query Execution Time*

In database systems, cardinality estimation [92] is used to measure the cost of processing a query. This approach estimates, or predicts, the count of every operation in a query and the result size of each operation based on statistics describing the data (e.g., histogram and distinct counting), *without actually executing the query*. For our use case, in-transit data on the SmartNIC, it is necessary to progressively update and revise each histogram as new data is received, merged, or migrated. Once the cost of a query workload is captured, we can combine the resource availability (e.g., available thread count) of the target system (e.g., SmartNIC or host) to predict the execution time of the workload. We also note that cardinality estimation on a SmartNIC requires efficiency due to limited resources compared to typical instances of mature database systems (e.g. PostgreSQL [93] or SparkSQL [94]).

Characterizing: For each query workload, we inspect the physical execution plan and generate an operation vector to encapsulate the cost of the query workload, as illustrated in Figure 7-9. For each batch of in-transit data, we use two data sketch algorithms from the Apache DataSketches library [91] to derive the distinct counting and histogram statistics–Theta Sketch [95] and KLL Sketch [96], respectively.

Our cardinality estimator currently supports queries that involve filtering, projection, aggregation, or any combination of these operations. The estimator is also capable of estimating reducible conditions to maximize the value of statistics generated for individual columns. For instance, the condition `ABS(vx) < 30` can be equivalently transformed to `vx > -30 AND vx < 30`, and can utilize the histogram statistics for the `vx` column to estimate the cardinality satisfying this condition. Moreover, the estimator can estimate queries that depend on multiple data sources. This capability is particularly significant as it allows for the estimation of workloads querying multiple data table partitions simultaneously on a single SmartNIC.
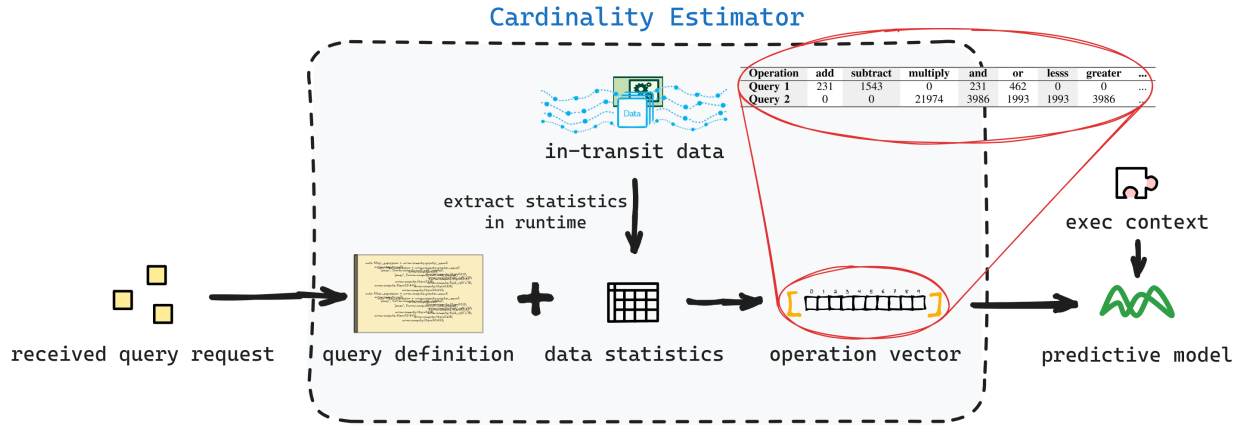
Figure 7-9. Predicting Query Execution Time by Constructing an Operation Vector for a Query

Predicting: To convert an operation counts vector (see Figure 7-9) into execution time, we trained a random forest model. Our training data consisted of query templates, representing C++ logical query plans, where placeholders were filled with randomly generated constants (i.e. numerical predicates) to produce concrete logical plans. Each record in our training dataset included operation counts, the number of rows in the queried table, the thread counts, and the actual time for workload execution.

(a) Row Output Estimation

(b) Prediction Time vs. Query Execution Time
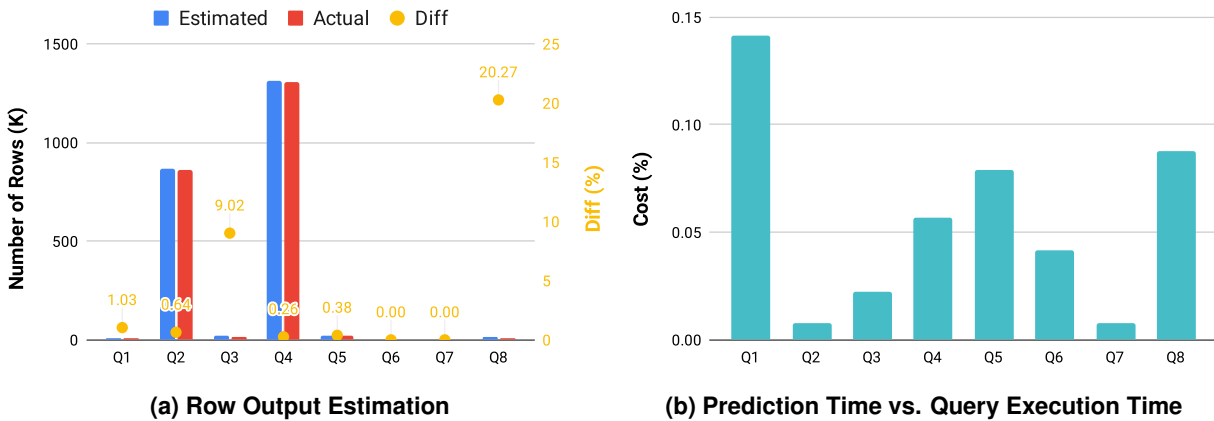
Figure 7-10. Cardinality Estimation Performance for Eight Sample Queries

Figure 7-10(a-b) illustrates the performance of the cardinality estimator using randomly generated test queries on data tables of varying sizes. The first evaluation (a) indicates that the majority of predictions have an error rate that is within 1% of the true cardinality. However, some queries such as Q8 have a higher error rate due to the aggregation of multiple statistics' biases. For the Q8 query, the prediction uses histogram statistics for three filtering conditions and distinct counting statistics for one aggregation condition. Furthermore, this query requires applying the distinct counting statistics to a subset of the table resulting from the filtering, which can introduce significant errors if the data itself is biased. Figure 7-10(b) measures the efficiency of conducting the cardinality estimation on the BlueField-2 SmartNIC. These measurements indicate that a prediction can be made in a fraction of the time that the query would take to execute. This efficiency is essential for implementing the decision engine on the SmartNIC.

A test suite of 16 different queries was constructed to measure the difference between predicted and actual query execution times. The percentage difference for each query is presented in Figure 7-11, with the number of threads used in each query listed at the base of each bin. It is worth noting that our estimator supports operation counting for sub-conditions, which enables fine-grained execution time predictions for complex, composable query workloads.
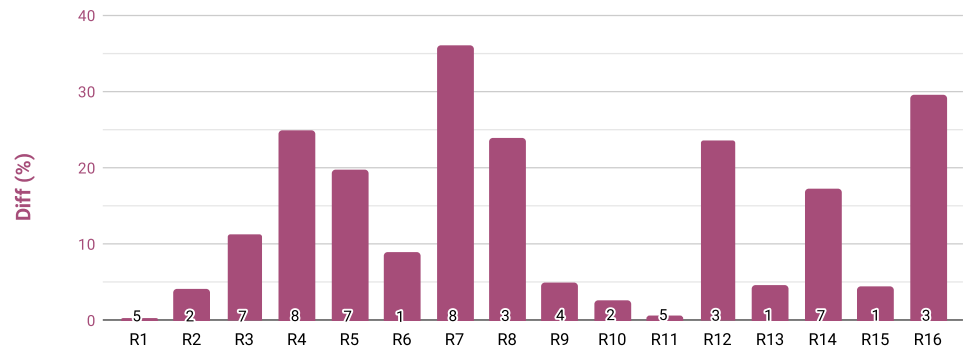
Figure 7-11. Discrepancy between Actual and Estimated Execution Times for Test Queries

7.3.   Performance Measurements

Let us proceed with two case studies to exemplify the application of our decision engine in estimating the beneficial execution location for a query workload. We began by using our prediction models to estimate the time consumption associated with each factor shown in Figure 7-3. This process considered both offloading and pushing back the workload to a host equipped with two Intel Xeon 16-core E5-2698 CPUs running at 2.30GHz and 512 GB of memory. We then measured the actual time consumption of the query workload by executing it on both the SmartNIC and the host system. By comparing these data sets, we could evaluate the effectiveness and accuracy of our decision engine, which makes scheduling decisions only based on aggregating all the estimated time consumption factors. Note that in both case studies, source data was contained in a single data object, thereby removing the need to merge multiple objects in the push-back case.

In the first study, we analyze a query applied to a particle dataset comprised of 6,177,731 rows:

$$\text{SELECT * FROM particles} \atop \text{WHERE x >= 0.7 and y < 0.3 and z <= 0.1}$$ (CQ1)

The actual execution of this query results in 55,517 rows, or just 0.9% of the original data. The cardinality estimator predicts 55,036.7 rows, showing a difference of 0.865% from the actual row count. For this particular query, the operations vector is generated as follows:

Table 7-1. The Operations Vector Produced for Case Study Query CQ1

and_kleene	filter	greater_equal	less	less_equal	select	table_rows
12355500	6177730	6177730	6177730	6177730	55036.7	6177731

95

The second study uses a query slightly different from the first one to examine the crossover point where offloading and pushing back result in similar execution costs:

$$\text{SELECT * FROM particles}$$
$$\text{WHERE x >= 0.5 and y < 0.55 and z <= 0.67} \quad \text{(CQ2)}$$

Executing this query on the same dataset yields 1,136,847 rows, which represents 18.4% of the original row count. The cardinality estimator predicts a return of 1,152,860 rows, indicating a minor discrepancy of 1.41

The estimated and actual time consumption, aggregated from the time factors for each of the two scenarios, are depicted in Figure 7-12. The query execution time for the SmartNIC is both measured and estimated using six threads, whereas, for the host, it is measured and estimated utilizing 32 host threads. This bias is intentionally introduced to account for the host's superior availability of computing resources. Despite this adjustment, the comparison reveals that for the first query, choosing offloaded execution significantly reduces execution latency by 74.64% due to low-percentage selectivity. This outcome can be attributed to the high network transfer cost that dominates the total execution latency in the scenario of pushed-back execution. As for the second query workload, execution latency is comparable whether conducted on the SmartNIC or the host. Specifically, while the estimation slightly leans towards pushing back, keeping execution on the SmartNIC reduces latency by 1.38% due to higher-percentage selectivity.
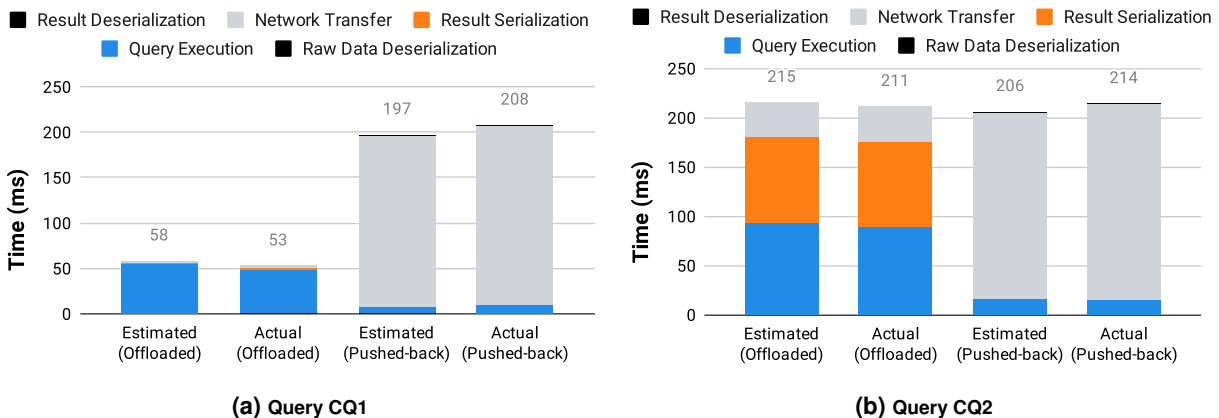
Figure 7-12. Analysis of Time Consumption for Offloaded vs. Pushed-Back Execution with Case Study Queries.

7.4.   Summary and Future Work

Optimizing data query workload execution via dynamic offloading across systems of different architectures is challenging. However, data management techniques open avenues for developing a workload placement decision engine for SmartNICs, tailored to the HPC data processing landscape. Importantly, our decision engine's predictive approach may extend to embedded systems with similar hardware capabilities (e.g., computational storage devices) to exploit data service offloading benefits. It is worth noting that several factors, including network bandwidth variation, system resource fluctuation, and performance interference among different data services, could impact dynamic offloading performance. Addressing these factors remains vital in our future work to enhance the efficiency of dynamic queries on SmartNICs.

96

8.   OPTIMIZING HOST-TO-SMARTNIC DATA TRANSFERS

As discussed in section 6.2.1, one of the challenges of leveraging SmartNICs in real application flows is simply transferring data as efficiently as possible from the host to its local SmartNIC. While RDMA benchmarks in section 3.3.2 indicate that there are performance advantages for local communication, it is easy for these advantages to be lost when application flows must factor in bookkeeping tasks such as memory allocation and registration, serialization, and synchronization. What is needed is a simple communication library that provides a more direct path for the host to exchange data with the SmartNIC in a customizable way. In this chapter we describe the design, implementation, and performance evaluation of the SmartNIC Data Movement Service (SDMS), which is a service that is designed to create a tight coupling between a host and its SmartNIC. SDMS implements an asynchronous conduit for passing data to and invoking operations on a SmartNIC.

8.1.   Overview of the SmartNIC Data Movement Service (SDMS)

To facilitate the offloading of data management tasks to the SmartNIC, we have built the SmartNIC Data Movement Service (SDMS). Applications running on the host use the Client API of SDMS to send requests to an SDMS Server process running on the SmartNIC. The request includes the information necessary to execute an RDMA READ operation from the SmartNIC (i.e., memory address, number of bytes to transfer, and remote key). The SDMS server acknowledges the request and transfers the data asynchronously.

SDMS is built on the low-level primitives provided by `hodcarrier` [1], a simple library built on the InfiniBand Verbs interface that enables high-speed transfer of data from host to SmartNIC memory. SDMS adds functionality (e.g., support for buffer caching, serialization) to provide a purpose-built service for transferring application data to the SmartNIC as part of offloading data management services. The basic theory of operation is that applications will use the SDMS Client to transfer raw output data to the SDMS Server running on the SmartNIC to facilitate offloaded data management and analysis.

---

[1] `https://github.com/sandialabs/hod-carrier`
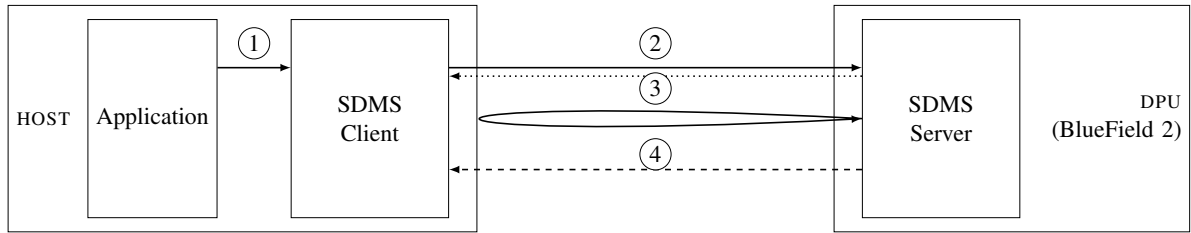
Figure 8-1. Data Flow for the SDMS library

Figure 8-1 depicts the basic sequence of operations for transferring data from the host (client) to the SmartNIC (server). A detailed description of the steps involved in performing a data transfer is provided below:

① When the application has data that is ready to transfer, it calls the `request_transfer` function of the SDMS Client API. The request includes: (i) the memory addresses of the buffers to transfer; (ii) the size of the buffers to transfer. and (iii) the remote keys for the RDMA transfer of the buffer.

② The SDMS Client sends the request to the SDMS Server running on the DPU (represented by the solid arrow pointing from left to right). The SDMS Server running on the DPU will acknowledge the request to the Server (represented by the dashed arrow pointing from right to left).

③ Based on the contents of the request, the SDMS Server acquires a destination buffer for the transfer and prepares an InfiniBand Verbs work request for executing an RDMA Read operation (`IBV_WR_RDMA_READ` opcode).

④ After the buffer has been transferred and processed, the SDMS Server notifies the SDMS Client that the transfer is complete and the source buffer can be reused. Completion notification is done lazily; the notification is piggybacked on the next message sent to the SDMS Client (e.g., a transfer request acknowledgement).

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

8.2.   Performance Characterization of the SDMS

To understand the performance of our implementation of SDMS, we characterized the data transfer bandwidth and host latency of `hodcarrier` using a simple benchmark. These results demonstrate that SDMS provides high-performance data transfer from host to SmartNIC.

8.2.1.   Comparisons with `qperf`

Using `qperf` [97] enables us to evaluate the extent to which SDMS is able to use the full network bandwidth. Figure 8-2 compares the bandwidth of SDMS measured with two different configurations of `qperf`. The SDMS data points represent the average over 10 trials. The `qperf` data points represent the average reported by the `qperf` tool. Bandwidth benchmarks, including `qperf`, initiate large numbers of transfer operations at once in an attempt to saturate the network and ensure high bandwidth even for modestly sized messages. The number of requests posted at once in `qperf` is controlled by the value of `NCQE`. By default, the value of `NCQE` is 1024 and is not modifiable at runtime. Initiating a large number of transfers establishes a valuable upper bound, however, users commonly want to transfer a small number of buffers at a time and are thus not able to achieve this upper bound (especially for small message buffers). As a result, to get a more realistic upper bound on network bandwidth, we modified `qperf` to allow the value of `NCQE` to be set by the user at runtime.
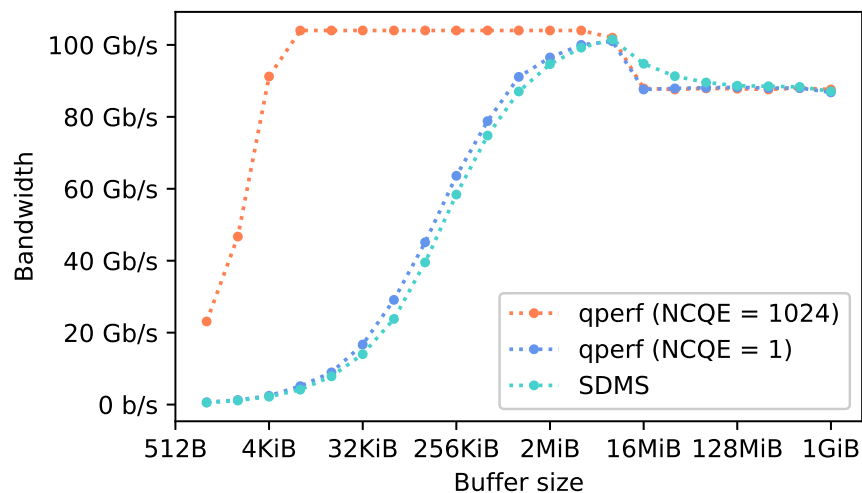
Figure 8-2. Comparison of Bandwidth Measurements for `hodcarrier` and the `qperf` Benchmark

The orange line in Figure 8-2 shows the bandwidth measured by the default `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 (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 19%. The blue line shows the bandwidth measured by `qperf` with the value of `NCQE` set to 1, to approximate the case where a user is sending one buffer at a time. These data show that the measured bandwidth is significantly lower for single message buffers, especially for small message buffers. We expect that, in most cases, applications will request the transfer of a small numbers of buffers at a time (e.g., once per timestep). As a result, the measurements with `NCQE` equal to 1 represent a fairer baseline to compare the performance of SDMS against. The turquoise line represents bandwidth measured using a simple SDMS benchmark. To ensure a fair comparison with `qperf`, we implemented this benchmark to ensure that buffers are registered and reused to minimize that impact of memory allocation and registration costs. The data in this figure show that, for messages that are 128 KiB or larger, the bandwidth measured with SDMS is greater than 85% of the bandwidth measured with `qperf` (`NCQE` = 1).

99

8.2.2.   *Impact of Memory Alignment on Bandwidth*

The SDMS server uses two principal optimizations to ensure high performance. The first is to allocate memory buffers with `memalign` to align buffers to page boundaries. The second is to pre-post and cache registered server buffers. Additionally, another potential optimization is to ensure that the memory buffers passed to the SDMS client are page-aligned (i.e., allocated with `memalign`). Figure 8-3 characterizes the impact that each of these optimizations have on the data transfer performance (i.e., bandwidth). These two subfigures evaluate eight different combinations of these three different optimizations.

(a) `memalign` Server Experiments   **(b) `malloc` Server Experiments**
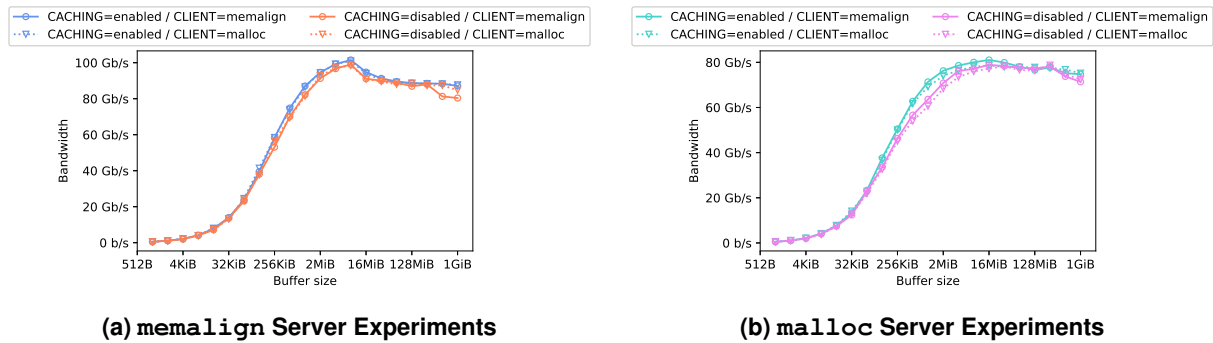
Figure 8-3. Impact of Memory Alignment on Bandwidth

The left figure (Figure 8-3a) shows the results when server buffers are allocated on page boundaries with `memalign`. The right figure (Figure 8-3b) shows the results when server buffers are allocated with `malloc` and are not guaranteed to be page-aligned. Within each figure there are results of four combinations of the client buffer allocation strategy (`memalign` or `malloc`) and buffer caching strategy (enabled or disabled). These data show that the critical optimization is to use server buffers that are page-aligned. The peak bandwidth achieved for trials with page-aligned server buffers is more than 25% higher than the peak achieved with non-page-aligned server buffers. For page-aligned buffers, these data also show that buffer caching may yield an additional improvement. For 1 MiB server buffers, buffer caching can increase the achieved bandwidth by nearly 6%. For non-page-aligned server buffers, using page-aligned client buffers may improve bandwidth by 5% or more.

8.2.3.   Impact of Page Alignment on Host Overhead

The SDMS client was expressly designed to minimize the host overhead incurred by data transfers. Figure 8-4 shows the time (minimum, maximum, and average) required on the host to initiate the transfer. The left subfigure (Figure 8-4a) shows the data for the case where client buffers are page-aligned and the right subfigure (Figure 8-4b) shows data for the case where client buffers are not guaranteed to be page-aligned. Overall, these data show that the average time required to initiate a data transfer is approximately 300 μs. These data show that the host overhead is (unsurprisingly) independent of the size of the buffer being transferred. The value of the Pearson correlation coefficient is 0.107 (very weak positive correlation) for experiments that use page-aligned client buffers and -0.332 (weak negative correlation) for experiments that use client buffers that are not guaranteed to be page-aligned.

(a) `memalign` Client Experiments   (b) `malloc` Client Experiments
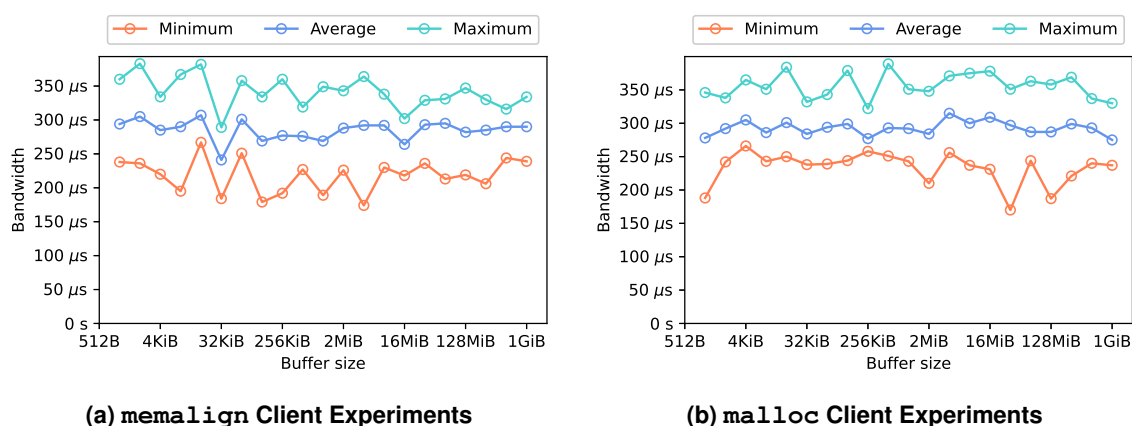
Figure 8-4. Host Overhead Based on Whether Client Buffers are Page-Aligned or Not

8.3. Serializing Application Data

The output data generated by many applications will need to be serialized in order to be moved off-node. Within the context of this report, we are interested in understanding how offloading parts of the serialization process to the SmartNIC may affect performance. Figure 8-5 contains graphical depictions of four different approaches to serialization for a simple example data layout that contains three arrays located in host memory:

(a) **No Serialization**: No serialization is performed to provide a baseline for the serialization approaches examined in this subsection.

(b) **Host Serialization**: The application output data is serialized by the host and a serialized buffer is transferred to the SmartNIC.

(c) **SmartNIC Serialization**: The raw application data is transferred directly to the SmartNIC and the SmartNIC performs the serialization operations.

(d) **Inflight Serialization**: The server buffers are managed to allow the application data to be serialized as part of the transfer from the host.

(a) No Serialization   **(b) Host Serialization**   **(c) SmartNIC Serialization**   **(d) Inflight Serialization**
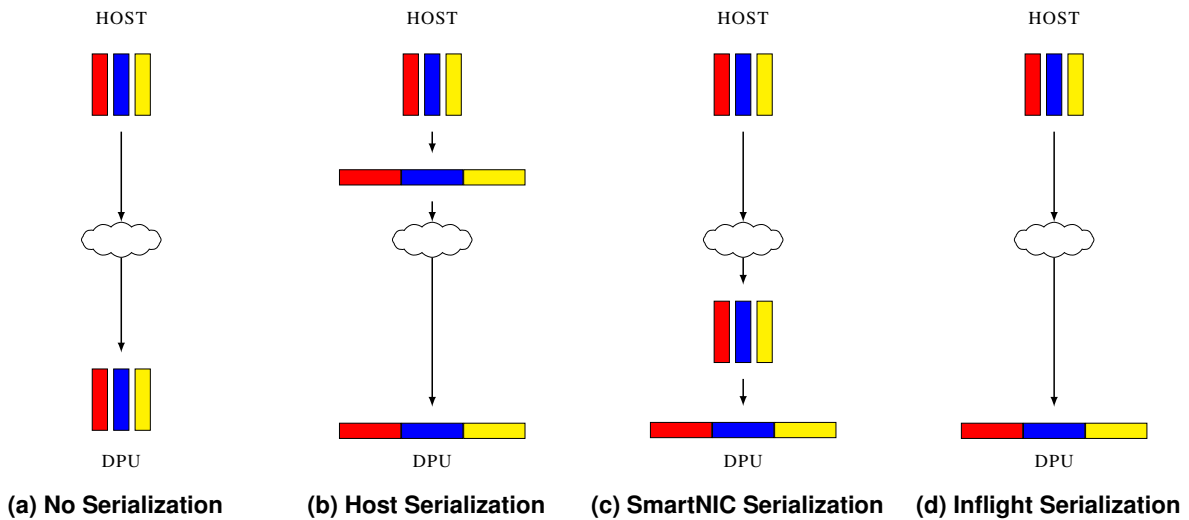
Figure 8-5. Four Approaches to Serializing and Transferring Three Arrays to a SmartNIC

8.3.1. Performance Comparisons

To understand the costs associated with these serialization approaches, Figure 8-6 shows the results of our serialization benchmark for four different approaches to serialization. Based on the results of our experiments in Section 8.2, we use page-aligned server and client buffers and cache memory buffers to enable reuse to the greatest extent possible. This figure presents our measurements of the effective bandwidth (i.e., the mean number of bytes that are transferred to a serialized buffer per unit time of server execution) achieved and the host overhead (i.e., the amount of host processor time required for the transfer) for three simple data layouts: serializing
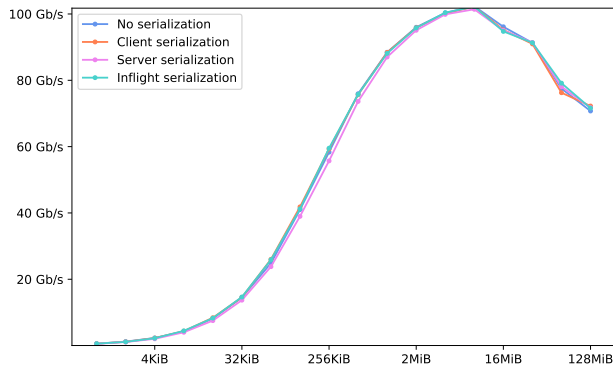
(a) Transfer bandwidth / 1 buffer
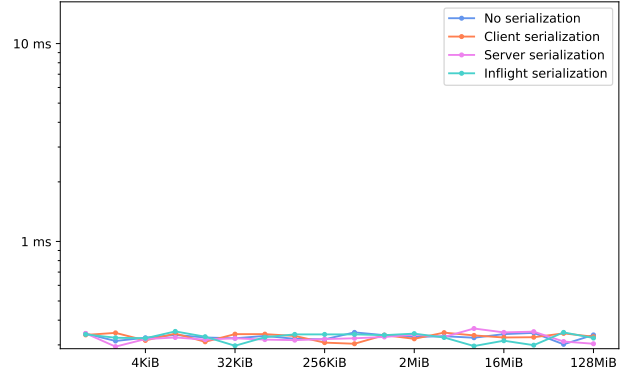
(b) Host overhead / 1 buffer
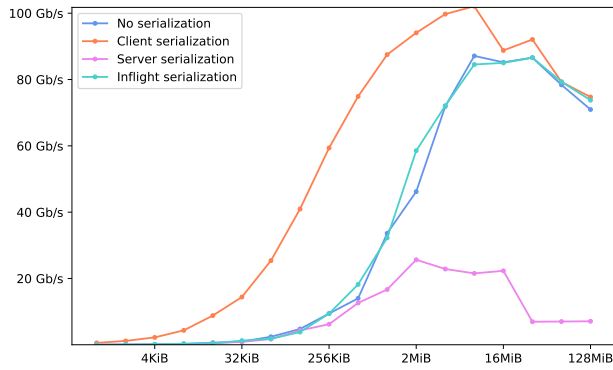
(c) Transfer bandwidth / 2 buffers
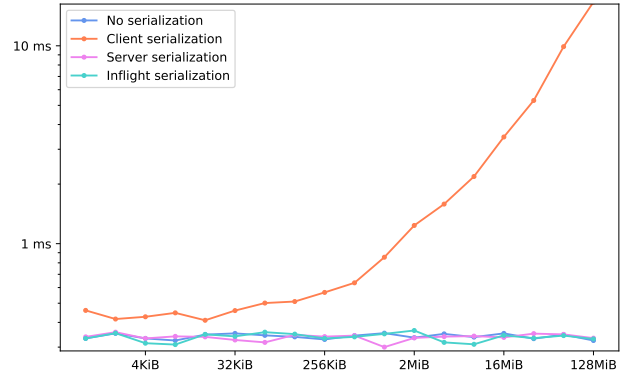
(d) Host overhead / 2 buffers
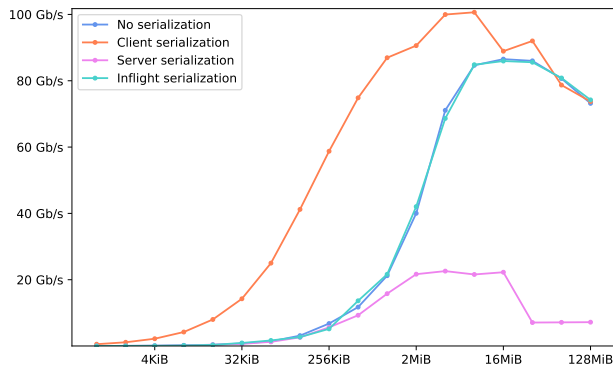
(e) Transfer bandwidth / 4 buffers
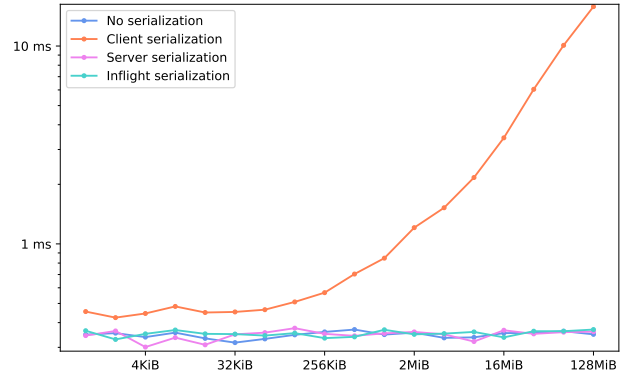
(f) Host overhead / 4 buffers

Figure 8-6. Comparing the effective bandwidth and host overhead of four different approaches to serializing application data consisting 1,2, and 4 output memory buffers

1, 2, and 4 equally-sized memory buffers. Figure 8-6a and Figure 8-6b show the results for "serializing" a single memory buffer. A single, standalone memory buffer is already serialized for our purposes so these data function as a baseline for our other experiments. As a result, all four serialization approaches yield nearly the same effective bandwidth and host overhead.[2].

The data for 2 and 4 memory buffers (*see* Figures 8-6c to 8-6f) show that for the no serialization experiments, transferring multiple smaller buffers instead of one large buffer reduces the effective transfer bandwidth but has little impact on the host overhead. Additionally, these figures show that the highest effective bandwidth is achieved for client serialization, nearly matching our baseline measurements. However, client serialization imposes significant overhead on the host that increases as a function of the total memory buffer size. Similarly, server serialization transfers the serialization costs to the SmartNIC. As a result, the effective bandwidth for moving data from the host is significantly reduced since the SmartNIC has to spend time serializing the buffers after they have been transferred. The data from our inflight serialization shows that this approach is able to closely match the effective bandwidth measured for the no serialization experiments without noticeably affecting the host overhead. This is a promising result that shows that we can significantly reduce serialization overheads if we are able to manage the data so that they are serialized as part of the transfer to the SmartNIC.

8.4. Summary

While RDMA benchmarks inform us that the hardware can transfer a contiguous block of data efficiently from the host to SmartNIC, it is difficult to realize peak performance levels in real world applications because user data structures involve multiple components that are not contiguous as a whole. SmartNICs offer an opportunity for us to offload the burden of serializing these components into a network transportable buffer. As a means of improving application coupling to and data exchange with a SmartNIC, we have constructed SDMS. This service excels at orchestrating a small number of data transfers into the SmartNIC in an asynchronous manner and can be used to perform inflight serialization.

---

[2]These results are also consistent with our earlier experiments, *see* Figure 8-3a

9.   STORAGE SERVICES

While many workflows can be constructed in a way that enables data to be streamed from one job's memory space to another's, users will always need more capacity than memory can offer and persistence for post-workflow analysis. As such it is valuable to consider ways in which SmartNICs can help system architects improve the manner in which storage systems are integrated into HPC platforms. In this chapter we discuss the importance of *job-local storage* and examine how compute node SmartNICs can be used to host parallel filesystems in a way that minimizes disturbances to host applications. Given that on-card SmartNIC flash storage is too small and slow to be of use in this space, we investigate NVMe-over-Fabrics (NVMe-oF) methods to allow a SmartNIC to leverage the host's NVMe devices. Experiments confirm that heavy I/O loads on the SmartNIC do not impede computational tasks on the host.

9.1.   Motivation for I/O Isolation

HPC platforms traditionally share a large parallel filesystem cluster amongst all compute nodes in the system to enable users to store their workflows' input, output, checkpoint, and intermediate data. Unfortunately, the shared nature of these filesystems means that I/O activity from one job in the platform may negatively affect the I/O performance of other jobs. These disturbances increase performance variability in the compute nodes and may have significant consequences in latency sensitive, parallel jobs. Some platforms have moved to include *node-local storage*, i.e. direct-attached SATA or NVMe storage on each compute node. This allows direct access to private storage on each node, but does not allow for shared access to files across a job. As a means of providing better I/O isolation than shared parallel file systems while preserving the ability to share data between nodes in a job, system architects have proposed and deployed *job-local storage* [98] that uses node-local NVMe and SSD devices in the compute nodes to create private parallel file systems close to computation.

9.1.1.   *Advanced Storage in Recent Platforms*

Recent HPC systems have used node- and job-local storage for the benefits listed above. The Summit [99] system, operational in 2018 at Oak Ridge National Laboratory, was designed with a two-tiered storage system [100] – a traditional PFS as well as node-local burst buffers. The burst buffer system is designed to support checkpoint-restore and provides multiple interfaces for applications to interact with the system. The Burst-Buffer API (BB-API) supports N-to-N checkpointing patterns with asynchronous draining of data to the PFS, while the Burst Buffer Shared Checkpoint File System (BSCFS) is a log-structured filesystem built on FUSE that

supports N-to-1 checkpointing. Applications can also use Scalable Checkpoint Restart (SCR) [101] to drain data from the SSDs to the PFS. All three of these solutions require modifications to application code, making each relatively difficult to use. Furthermore, BSCFS is not POSIX-compliant, raising other hurdles to application adoption. Data movement to the PFS is done via NVMe-oF. While this reduces compute-node CPU burden, node memory and network access can still be affected, and are controlled via QoS and throttling.

Two other systems were developed for Summit that allowed more transparent access of the burst buffer storage layer: Spectral and SymphonyFS [99]. Spectral supports N-N checkpointing to the burst buffer with no application code changes. Applications write directly to XFS on the local SSD. When the application closes the file, Spectral detects the `close()` system call and moves the data to the PFS without application involvement. SymphonyFS is another FUSE-based solution for N-1 checkpointing. It aggregates local SSDs into a single namespace and decomposes metadata and data operations. Metadata operations are passed through to the PFS. Data operations are cached locally on XFS and then drained to the PFS later. While SymphonyFS appears to provide a single namespace and is transparent to application developers, it assumes nodes work on non-overlapping regions of a file, and thus has a much different consistency model than POSIX, which limits its scope to certain checkpointing workloads.

The Fugaku supercomputer in Kobe, Japan at the Riken Center for Computational Science also uses two-tiered storage, and manages the fast NVMe layer with the Lightweight Layered I/O Accelerator (FFIO) [102] system. For every group of 16 compute nodes, Fugaku has a node equipped with NVMe storage that is designated as a storage and compute node. This node handles all higher-tier storage requests for other nodes in its group. FFIO is used for three purposes – a cache for the second-tier PFS, an area for temporary files shared between compute nodes running in a single job, and an area for temporary files exclusive to compute nodes. The temporary areas are not intended for files that will be sent to the second-tier PFS.

9.1.2. The Potential for Noise Interference by Filesystem Daemons

Existing research on job-local storage does not study the performance impact of running filesystem daemons on compute nodes. Tightly-coupled parallel workloads can suffer from performance degradation and variability when system noise interferes with the parallel workload [103]. A classic case of noise interference can happen when several ranks of an MPI process enter a synchronization barrier. If one rank is delayed due to a context switch, all other ranks must wait to exit the barrier. Experiments presented in this chapter show that the effect of a node-local filesystem daemon on parallel application performance is significant, variable, and disproportionate. In other words, application performance is affected by a larger margin than the CPU usage of the filesystem daemon, and the application's runtime has a greater level of uncertainty.

Current job-local solutions trades filesystem consistency or compute for acceptable performance. No system achieves transparent application access to the burst-buffer layer, POSIX-compliance, and low compute overhead. SmartNIC devices are well-positioned to help achieve all three goals. They are becoming increasingly common in HPC systems and data centers, they have their own

CPU cores and memory, and they are capable of running general-purpose Linux distributions. We investigate the utility of using SmartNICs to offload node-local filesystems, and how offload could impact compute performance.

9.2. Job-Local Filesystems

Job-local filesystems in HPC systems aggregate storage resources local to compute nodes in the HPC cluster under a single namespace. There are a number of advantages to colocating storage and compute including I/O performance and task isolation, or reducing the noisy neighbor problem. Many job-local filesystems are also *ephemeral*. That is, they only exist for the duration of an associated job, workflow, or application run.

Existing job-local filesystems are commonly used as burst buffers. Burst buffers are designed to absorb higher rates of I/O than traditional parallel filesystems. After data is written to the burst buffer, system software can asynchronously copy it out to the parallel filesystem.

To improve performance, many recent job-local filesystems relax POSIX semantics, assuming, mostly correctly [104], that HPC applications tend to manage their I/O carefully, and the application can be trusted, for example, to not write concurrently to the same file offset. Illustrative examples from the recent literature include UnifyFS [105], which uses commit consistency semantics, where writes are not globally visible until they have been committed, and GekkoFS [106], which provides eventually-consistent metadata handling by avoiding global locks during metadata operations.

Providing weaker consistency guarantees may improve filesystem performance, but could also limit these filesystems for use by existing well-behaved HPC applications. We chose to focus our efforts on BeeOND, an ephemeral job-local filesystem based on BeeGFS [107], which supports full POSIX semantics.

9.2.1. BeeOND

BeeOND is an ephemeral filesystem based on BeeGFS, a parallel filesystem designed for use in high performance computing systems. There are three main components in the BeeGFS filesystem – a *storage service*, a *metadata service*, and a *client service*. BeeGFS decouples data and metadata similarly to other distributed filesystems like Ceph [108] and Lustre [109]. The storage service stores striped BeeGFS data on standard Linux filesystems such as ext4 or XFS. Typically the storage nodes are in a RAID-6 configuration for fault tolerance. The metadata service controls the striping pattern and data placement. The client service mounts the BeeGFS filesystem on client nodes. Applications may interact with the filesystem as they do with any mounted filesystem. BeeGFS supports full POSIX semantics, so no modification is necessary for applications to use BeeGFS. When accessing data, clients first contact the metadata service, then the storage service directly to access the data. BeeGFS services may be run individually on separate nodes, or services may be colocated on single nodes.

In our setup, BeeOND startup was integrated with Slurm so that new Slurm allocations had their own private BeeOND instance, which ran for the duration of the allocation. The startup process automatically assigns the first node to host the metadata service, then distributes storage services across all nodes. It is also possible for the metadata service to be distributed over several nodes, in which case each node handles a segment of the filesystem namespace.

9.3.   Attaching Storage to SmartNICs

Current generation SmartNICs typically offer a small amount (less than 100GB) of permanently affixed flash storage. Given that the speed and capacity of this storage is inappropriate for HPC applications, it is necessary to consider other means by which we can attach storage to the SmartNIC. For this work we leverage the NVMe-over-Fabrics standard to access NVMe on the SmartNIC's host.

9.3.1.   NVMe-over-Fabrics

The shift from SATA SSDs to NVMe SSDs has caused an evolution in the Linux kernel I/O stack. Prior to Linux 3.13, with the introduction of `blk-mq` [110], a single-threaded I/O stack had acceptable performance due to the request latency of spinning disk hard drives, which were predominant. The parallelism of NVMe SSDs requires a different architecture for optimal usage of the disk. Data is written to NVMe devices using paired submission-completion queues. There may be up to 64K queues, and each queue may have up to 64K entries. Queues are typically mapped to cores on the host CPU. Queue entries are scheduled using either round robin or a weighted round robin strategy.

NVMe thus maps naturally to the send, receive, and communication queue pairs in RDMA. NVMe-over-Fabrics (NVMe-oF) is a protocol for accessing NVMe devices efficiently over a network, commonly RDMA via either InfiniBand or RoCE. When connecting an NVMe-oF initiator to a target, the user may specify the number of queues in the connection. Each queue is assigned its own core by the operating system, on both the initiator and the target. We were able to verify this behavior during our experiments.

An I/O request takes the following path in a filesystem mounted on an NVMe-oF block device. We consider a `write` request on the ext4 filesystem. First, the virtual filesystem (VFS) sends the request to ext4. The ext4 filesystem then creates a block I/O request (`bio`). The `bio` is sent to the `nvme` driver, which is presenting a block device to the initiator system. The block device can either be backed by a physical NVMe device, or an NVMe-oF connection to the target. In this case, we assume an NVMe-oF connection via RDMA, so the `bio` is routed to the RDMA subsystem.

Once on the target, the `nvmet` (NVMe-target) driver processes the request by sending the encapsulated `bio` to kernel block layer. Once finished, the target sends an acknowledgment to the initiator, which receives it as a software interrupt.

9.3.2. NVMe-oF NIC Offload

Recent Mellanox ConnectX HCAs (5 and higher) support NVMe-oF target offload [111]. We found that the standard Linux `nvmet` driver for RHEL8 did not include offload support, so we used the MLNX_OFED `nvmet` driver released by Mellanox. When target offload is enabled, the network interface is able to handle the NVMe request received over RDMA directly, without going through the kernel block layer. This reduces CPU usage on the target to nearly zero.
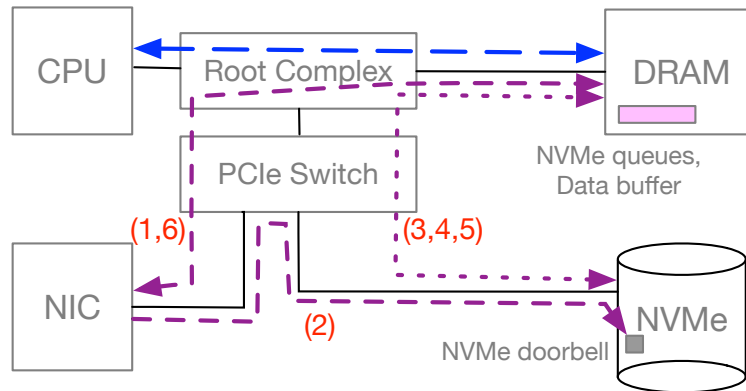
Figure 9-1. NVMe-over-Fabrics NIC Offload with Memory Accesses

In more detail, these are the steps for an offloaded NVMe `write` operation (Figure 9-1):

1. The offload engine generates a submission queue entry and does a DMA write to the submission queue for NVMe in main memory.

2. The offload engine does a P2P write, ringing the "doorbell" of NVMe device.

3. The NVMe device responds to the doorbell and fetches the submission queue entry from memory via DMA.

4. The NVMe device writes data from the memory buffer and stores it on disk.

5. The NVMe device writes to the completion queue to notify that the operation has completed.

6. The offload engine polls the completion queue to detect when the write operation is done.

Although NVMe-oF offload reduces CPU usage on the target, it still uses memory bandwidth, a source of possible contention [112]. Some NVMe drives have a controller memory buffer (CMB) [113], a region of DRAM accessible via a PCIe base address register, which may be used to store the submission and completion queues. In that case, reading and writing to the queues is done via P2P DMA, and operations can be offloaded without accessing system memory. Our NVMe disks did not have CMBs, so we did not explore this functionality.

9.4.   Interference Experiments on the Host

The goal of our first set of performance experiments is to observe how much I/O services disturb compute tasks when the two execute on the same host. Specifically, we evaluate the performance impact of BeeOND on compute node performance. Given 3*N* nodes participating in BeeOND, we run a compute-heavy workload on 2*N* nodes requiring communication between all ranks, and an I/O-heavy workload on *N* nodes. Since all nodes are running BeeOND, I/O is distributed across all 3*N* nodes. Figure 9-2 shows our experimental design and the placement of BeeOND services for $N = 2$ (there are two nodes running the I/O workload and four nodes running the compute workload). HPL, IOR, and the BeeOND storage and metadata services run in user space, while the BeeOND client is a kernel filesystem module.
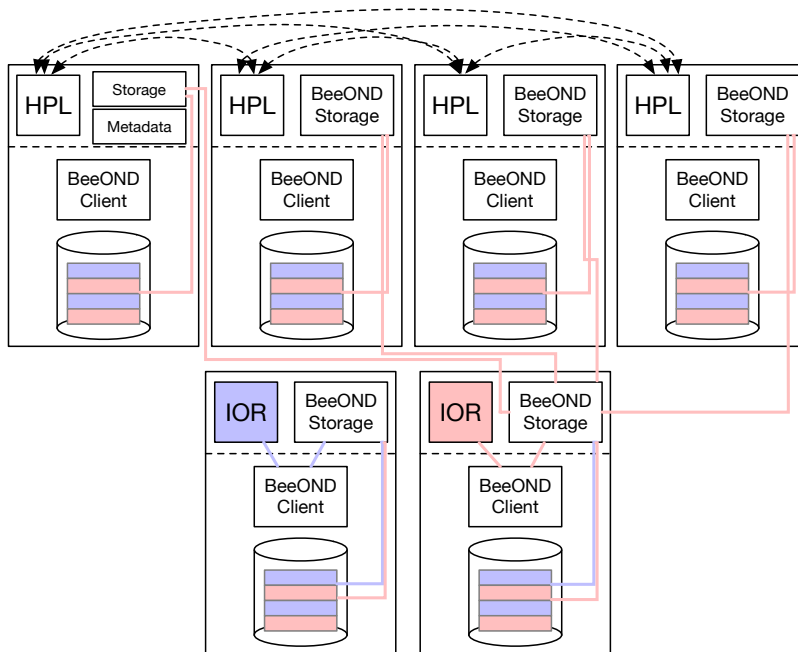
Figure 9-2. HPC allocation with BeeOND, HPL, and IOR. Blue storage lines omitted for clarity.

9.4.1.   Compute Workload

We use the High Performance Linpack (HPL) benchmark [114] to provide a compute workload that is reflective of HPC applications. HPL performs matrix factorization in a tightly coupled MPI job. It is compute intensive. Our hypothesis before running the experiment was that HPL would experience slowdowns due to its tightly coupled nature. When a group of MPI ranks enters a barrier, they may only leave the barrier once the last rank has entered. This makes barrier performance degrade as performance variation increases.

After studying HPL more closely, we found that it uses a more complicated communication scheme than a simple barrier. HPL has six different broadcast algorithms which are configurable at runtime. On every iteration of the LU factorization, ranks must broadcast across their row and

column. In the default "increasing-ring" topology, the broadcasting rank shifts to the next rank in the ring on each iteration [115]. Although this is not exactly the communication scheme we expected, it exhibits similar performance degradation to the barrier model due to inter-rank communication dependencies, as we show in the evaluation.

9.4.2. I/O Workload

We use IOR to generate I/O-intensive workloads for our experiments. IOR is a configurable benchmark suitable for evaluating parallel filesystems. IOR has a POSIX backend and uses MPI for rank synchronization which makes it particularly suitable for HPC systems.

```
srun -N ${iornum} -n $((cores*iornum)) -w ${ior_nodes} \
    ior -t 128 -T 20 -D 60 -i $((1024*1024)) \
        -e -C -w                              \
        -a POSIX -s 1024 -F -Y                \
        -o=/mnt/beeond/testfile
```

Figure 9-3. IOR Runtime

Our complete IOR runtime is given in Figure 9-3. We discuss the most important flag choices here. We perform 128 byte writes (`-t 128`) with an `fsync` after each write (`-Y`). Each process has it's own file (`-F`) to avoid contention on the file level. We repeat the test (`-i`) continuous for 20 minutes, or until the completion of HPL. This ensures that we are performing I/O for the duration of the experiment.

The small sequential writes and frequent `fsync` calls are designed to generate significant I/O activity and exercise BeeOND as much as possible. We experimented with 512B, 2K, and 8K writes and found similar results with all four sizes. Future work could investigate other I/O patterns, such as large sequential writes, more fully.

9.4.3. Impact of BeeOND on HPL Runtime

Co-running the HPL compute workload with BeeOND daemons servicing IOR write activity has a substantial impact on compute performance. Using the setup described in Figure 9-2, Figure 9-4 shows the duration of our HPL experiment, averaged over 10 runs with errors bars showing the minimum and maximum, for several different configurations. The column furthest to the right, `hpl`, has no corunning filesystem daemon. The second column from the right, `hpl-fs-daemon`, shows the runtime with BeeOND daemons running, but no I/O workload. The other four columns show run times with IOR configured for the sizes of writes labeled on the x-axis.

Notably, HPL runs with very low variance when it is the sole tenant on each node. When BeeOND is servicing an I/O workload, HPL runtime increases and becomes more variable. We measured the CPU usage of BeeOND separately, and found that per node, it uses a total of between 3-4 cores of CPU time, out of 56 cores on each node. If the slowdown in HPL runtime

111
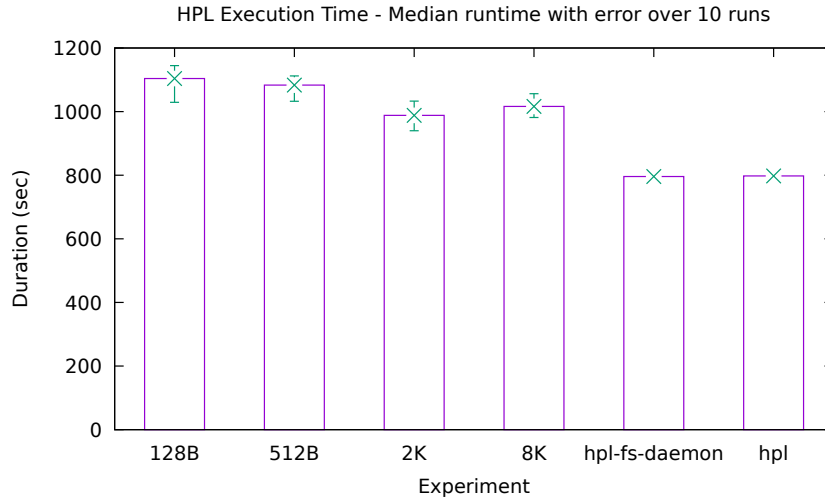
Figure 9-4. HPL Runs With and Without Corunning IOR + BeeOND

was proportional to the CPU usage of BeeOND, we would expect HPL runtime to be about 7% longer. Instead, we find that the median runtime of HPL with BeeOND and IOR is 38% slower, and it is 44% slower on the worst run. Performance improves as write sizes increase because the number of I/O operations decreases. However, even with 8K writes, HPL runtime is significantly affected.

9.4.4. Impact of BeeOND on HPL Communication

Our hypothesis was that MPI ranks performing panel broadcast in HPL were being delayed by BeeOND, which led to increased execution time. We verified this by instrumenting the HPL code to time when ranks sent broadcast messages. Our results are shown in Figure 9-5. The left plot shows rank broadcasts for HPL running alone, while the right plot shows rank broadcasts for HPL corunning with BeeOND.

The Figure 9-5 plots present the difference in time between consecutive messages sent during the broadcast phase of one HPL run, for each of the first 16 ranks. Since all durations are found by subtracting timestamps from the same rank, we avoid worrying about clock drift between ranks.

The results show that HPL broadcasts have a distinctive communication pattern which falls into two modes – a shorter time between messages sent, and a longer time. When BeeOND is running on the node with an I/O-intensive workload, the pattern is still visible, but there is a large increase in communication time variation. Since computation cannot continue until ranks have broadcasted their panels, an increase in variation leads to a longer runtime.

112

(a) HPL Running Alone (b) HPL co-running with BeeOND Services and I/O
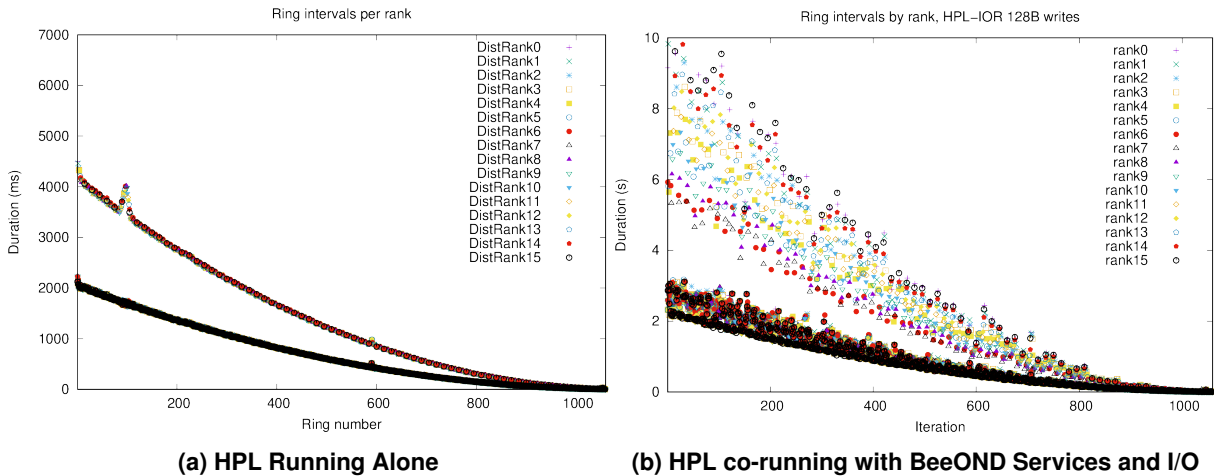
Figure 9-5. HPL Time Between Messages Sent (First 16 Ranks)

9.5.　　BeeOND SmartNIC Offload

Thus far, we have shown that co-running BeeOND on nodes in an allocation has a measurable performance impact on compute tasks. We now investigate the feasibility of offloading BeeOND to SmartNICs attached to each node.

The key mechanism we chose to make offload possible is to access host storage using NVMe-over-Fabrics (NVMe-oF). The SmartNIC can transparently access an NVMe device on the host using NVMe-oF offload which is available on modern ConnectX HCAs. Using HCA offload reduces host CPU usage to near zero while the SmartNIC is accessing storage. Alternatively, NVMe-oF can be used with non-NVMe SSDs or RAM disks, but HCA offload is not available. We investigated the following questions using two test setups, described in Tables 9-1 and 9-2.

1. Can the SmartNIC access host storage using NVMe-oF?

2. What is the overhead of accessing an NVMe device over fabrics, compared to accessing it locally?

3. Do lower-powered cores on the SmartNIC lead to a performance penalty compared to a full host when accessing storage via NVMe-oF?

CloudLab sm110p - Two nodes
16-core Intel Xeon Silver 4314
128GB ECC Memory
1TB SSD
4x1TB Samsung 980PRO NVMe SSD
100Gbps network
Dual-port Mellanox ConnectX-6 Dx, Dual-port Mellanox ConnectX-6 Lx

Table 9-1. CloudLab Test Setup

Singra node with BF-2
32-core AMD EPYC 7513
512GB ECC Memory
2x1TB Samsung 980PRO NVMe SSD
NVIDIA BlueField-2 SmartNIC via PCIe x8

Table 9-2. Singra Test Setup

113

We used the CloudLab setup primarily because it offered more administrative flexibility, although it also let us compare the performance of a SmartNIC relative to another full node. We use terminology similar to iSCSI for NVMe-oF – the "target" hosts the NVMe disk being accessed, while the "initiator" is the system accessing the disk remotely. The relationship is the same whether working with two full nodes, or a SmartNIC and a full node.

The SmartNIC on Singra is configured in separated host mode. Thus, the SmartNIC has it's own IP address, and can communicate with the target via RDMA. In this way, the setup for both systems is nearly identical.

9.5.1.   NVMe-oF Performance for a Host

First, we determine the performance of NVMe-oF compared to accessing an NVMe device locally. We use FIO for disk microbenchmarks – latency, IOPS, and throughput. We also use LevelDB as an application-level benchmark to measure expected real-world performance of the system.
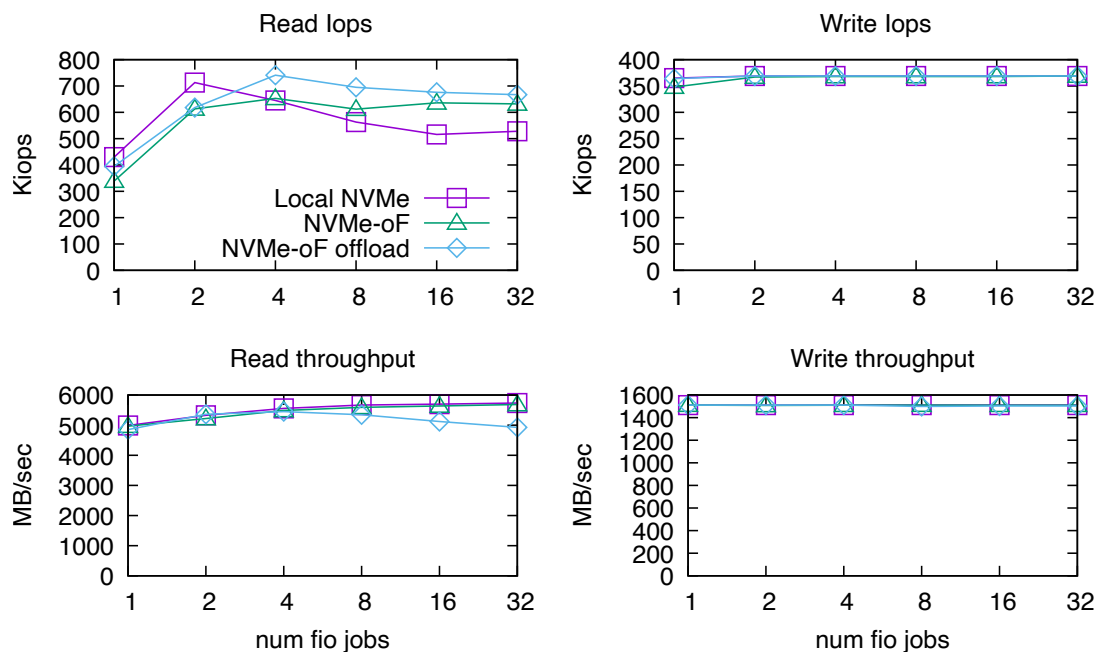
Figure 9-6. NVMe-oF FIO Performance Measurements on CloudLab between Two Hosts

Figure 9-6 shows the performance of NVMe-oF using the two-node CloudLab setup (Table 9-1). Results for NVMe-oF with and without target offload are compared to local NVMe performance. NVMe-oF is competitive with accessing an NVMe device locally for throughput and IOPS workloads. In fact, NVMe-oF outperforms local disk access in read IOPS. This was attributed to interrupt coalescing in a previous study of NVMe-oF [116], although the authors did not evaluate the interrupts generated by either the NVMe device or the network interface. We found (Figure 9-7) that fewer interrupts were generated on the NVMe-oF target per operation compared to an equivalent workload with local NVMe access.

(a) Local NVMe Access

(b) NVMe-oF Target
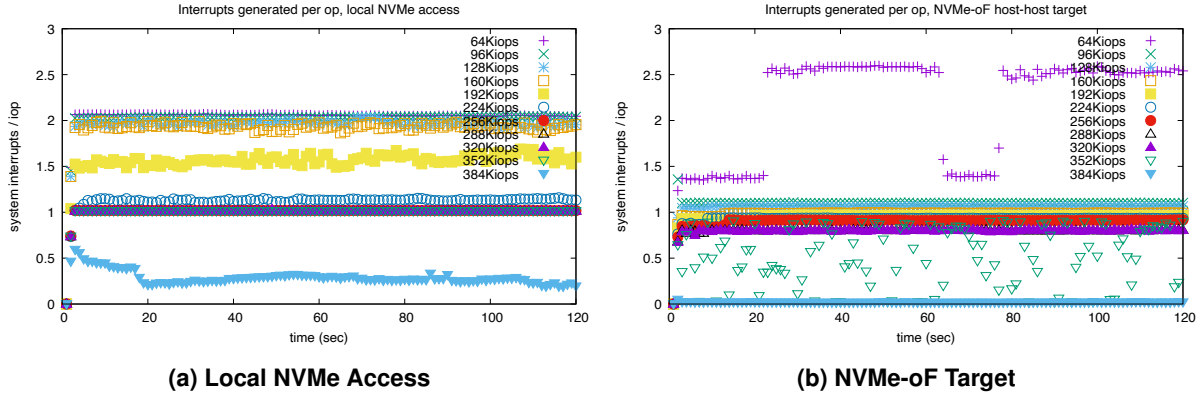
Figure 9-7. All System Interrupts Generated per Write Operation (Two Second Intervals)
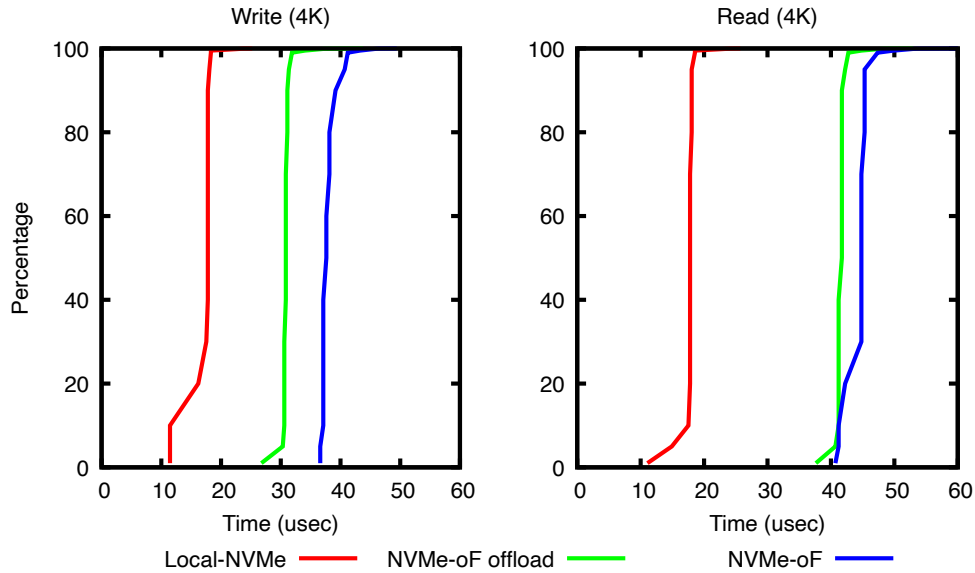
Figure 9-8. Operation latency CDF – Local NVMe and NVMe-oF

NVMe-oF has higher latency than local device access due to network overheads. Interestingly, offloading NVMe-oF reduces latency compared to traditional NVMe-oF. This is because NVMe-oF offload bypasses the kernel block stack on the target. We give a latency CDF in Figure 9-8. One other benefit we would expect to see in NVMe-oF offload is improved tail latency when the target is under load. We do see improved tail latency for NVMe-oF offload in Figure 9-8, but this measurement was conducted on an unloaded system. NVMe-oF tail latency under load has been studied in a comparison with iSCSI [116], but not with NIC offload.

We also confirm that CPU usage is reduced while using offload. Figure 9-9 shows that system interrupts on the target, as measured by `vmstat`, are at idle levels for all evaluated workloads when offload is enabled.

115
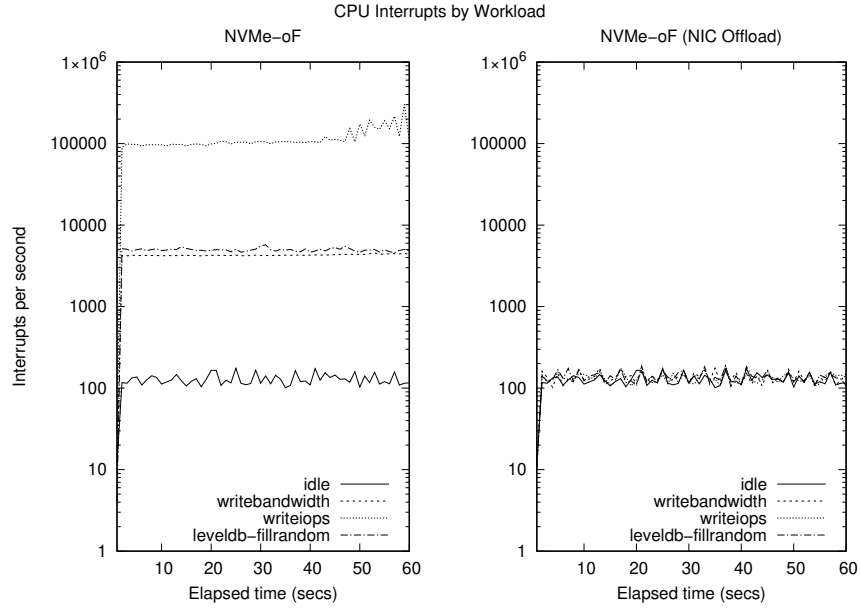
Figure 9-9. NVMe-oF Offload Target Interrupts

9.5.2. NVMe-oF Performance for a SmartNIC

Next, we evaluate NVMe-oF performance when accessing host storage from a BlueField-2 SmartNIC. These experiments used the Singra setup (Table 9-2). The host-host measurements were done on CloudLab (Table 9-1) and were used to control for the slower cores of the BlueField-2. The SmartNIC has a harder time generating a large amount of I/O operations. In the write IOPS benchmark, it needs four FIO jobs to generate the same amount of operations as a single FIO job on a CloudLab node. It is, however, capable of sustaining the same amount of write IOPS.
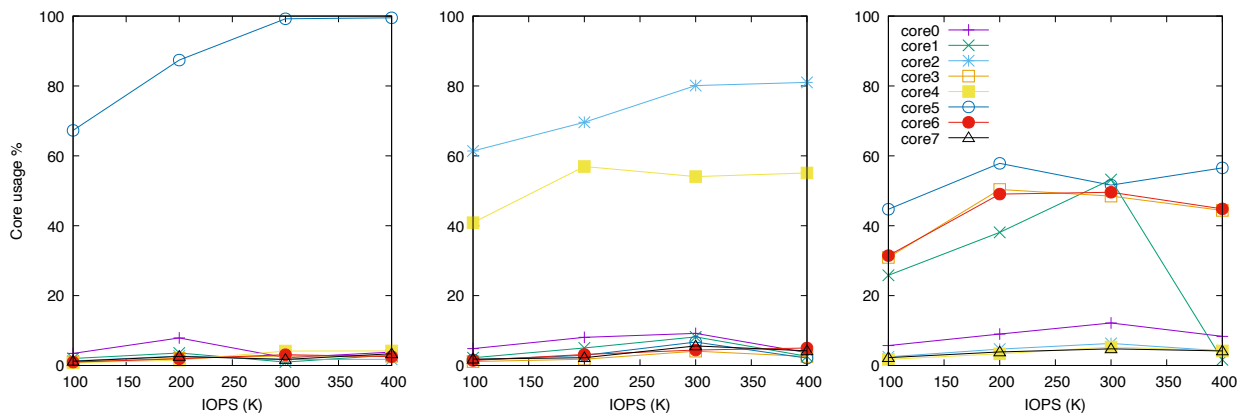
Figure 9-10. NVMe-oF BlueField-2 CPU Usage for Cores Processing Software Interrupts.

We find that the SmartNIC is unable to keep up with a CloudLab host in read IOPS. Our first thought was that using a single NVMe-oF queue saturated a core on the SmartNIC, and limited the number of read IOPS the system was capable of. Experiments with increased numbers of

queues showed that this assumption was correct and that core usage went down when the system was configured to use two queues (Figure 9-10). This increased the number of read IOPS the system could support, but increasing the number of queues beyond two had limited effect, and the system still had decreased performance compared to a CloudLab host. Future work may help determine another bottleneck.

Throughput measurements show that the SmartNIC can access the device with at least the same throughput, and better throughput in the case of read throughput, compared to accessing the device with another node. Latency measurements, comparing offloaded requests from a SmartNIC as well as a separate host, show that the SmartNIC has a latency advantage compared to other hosts (Figure 9-11), and operation latency is generally within 10 microseconds of local NVMe performance.
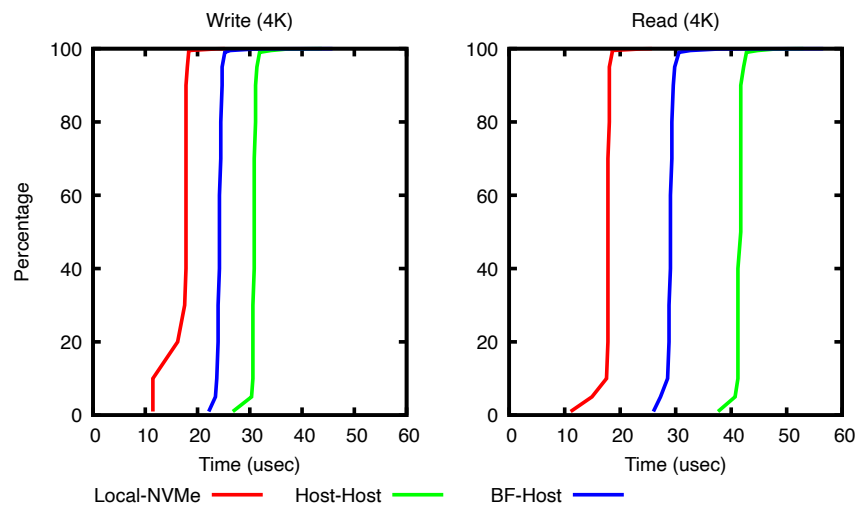
Figure 9-11. Operation Latency CDF – NVMe-oF Offload Host-to-Host and SmartNIC-to-Host

9.6.   Summary

We have shown that compute workloads can suffer performance penalties if there is an I/O-intensive workload running on a node-local filesystem. These performance penalties are disproportionate to the amount of CPU time used by the filesystem, which makes offload to low-power cores on SmartNICs an attractive solution.

The client-daemon architecture of BeeOND is suitable for offload. BeeOND daemons may run on SmartNICs, which can access host storage transparently using NVMe-oF. BeeOND clients may run without modification on hosts. In our experiments, accessing host storage from the SmartNIC with NVMe-oF proved to have excellent performance, matching or exceeding local NVMe performance in read/write throughput and write operations per second, and incurring a relatively small latency penalty. More investigation is needed to determine the bottleneck for small random read operations, but this workload is less important for traditional HPC applications. NVMe-oF NIC offload, available in modern ConnectX HCAs, reduces interrupt generation and CPU usage

117

on the host to idle levels. We expect, therefore, that offloading BeeOND filesystem daemons to SmartNICs will substantial lower CPU usage on host nodes, and maintain good filesystem performance. This approach retains POSIX-compliance, making the filesystem usable for workloads other than traditional checkpoint-restore, filling a gap in the current ecosystem of HPC filesystems.

10.   COSTS: PROCUREMENT AND POWER

In the previous chapters we have explored the technical aspects of integrating SmartNICs into modern HPC workflows and discussed ways to make the hardware more accessible to users. In this chapter we focus on a different aspect of SmartNICs that impedes widescale adoption in HPC: costs. SmartNICs are inevitably more expensive to purchase and consume more power to operate than traditional NICs. As such, each institution must determine whether SmartNICs can offload enough work to decrease the system's node count and be of value for their specific workloads. In this chapter we quantify the financial burden that SmartNICs add to purchasing and operating a system with SmartNICs. For this discussion we focus on procurement details and power measurements for Sandia's Glinda cluster.

10.1.   Procurement Costs

The first place where SmartNICs add to the cost of operating a computing platform is in the initial procurement. For HPC platforms we assume that architects will always select a high-bandwidth, low-latency communication fabric to maximize the rate at which host processors and storage systems can all exchange data. In this cost analysis it is important to remember that high-performance network infrastructure is not inexpensive. For example, core network switches may cost $500,000 or more. Plain HPC NICs for peak speeds are generally $1,000. Finally, high-speed network cables are much more expensive than most people realize, ranging from $200 for copper cables to over $1,000 for fiber links with high-speed transceivers. While system architects pay particular attention to driving the per-node network costs down in a procurement, these numbers help illustrate that spending more for a SmartNIC may be worthwhile if it reduces the number of nodes and network ports required in the overall platform.

10.1.1.   Impact of SmartNICs on Glinda Node Cost

Examining the procurement details of the 2020 Glinda system helps quantify the costs of purchasing an HPC system with SmartNICs. During the first stage in this procurement, Sandia's system architects gathered information about different system components and estimated the discounted price a system integrator would likely charge in a procurement of approximately 100 nodes. While the bidding process and fluctuating market prices make these estimates difficult, Sandia's planners found the winning bid to be very close in cost to their estimates. Table 10-1 lists the estimated costs for each component in a Glinda node when selecting either a plain InfiniBand NIC (e.g., a 100Gb/s ConnectX-6 InfiniBand card) or a BlueField-2 SmartNIC.

Table 10-1. Estimated Costs for a Glinda Compute Node

Component	Estimated Cost		Percent of Node Cost	
	InfiniBand	BlueField-2	InfiniBand	BlueField-2
Chassis and Motherboard	$2,300	$2,300	12%	12%
32-Core Zen3 Processor	$2,400	$2,400	13%	12%
512GB DDR4 DRAM	$2,400	$2,400	13%	12%
NVMe	$200	$200	1%	1%
25Gb/s Ethernet NIC	$300	$300	2%	2%
Ampere A100	$9,400	$9,400	51%	48%
InfiniBand NIC	$850	–	5%	0%
BlueField-2 VPI NIC	–	$2,000	0%	10%
100Gb/s Cable (Copper)	$120	$120	1%	1%
Single IB Switch Port	$370	$370	2%	2%
Total	$18,340	$19,490	100%	100%

1. Prices are based on 2020 discounted estimates
2. IB switch assumes 40x200Gb/s ports, with a $240 port-splitter cable

As this table indicates, GPUs are by far the most expensive component in the node architecture and dominate its overall cost. The main processor, memory, chassis, and SmartNIC represent the bulk of the cost for the remaining components in the node. While the BlueField-2 is 2.35x more expensive than the InfiniBand card, it only increases the node cost by 6%.

A caveat of this pricing is that the Glinda nodes were designed to supply data analytics users with a "thin-slice" architecture that includes one CPU, one GPU, and one network link. In contrast, HPC systems are typically designed with a "fat-node" architecture that features multiple CPUs, GPUs, and network links. Scaling the Glinda estimates to two CPUs and four GPUs reduces the BlueField-2 percentage of the node cost from 10% to 4%.

10.1.2. Network Costs for Additional Nodes

If the BlueField-2 SmartNICs processors *are* sufficient for implementing data management services, it is worthwhile to examine how much it would cost to build a comparable system that simply supplements the platform with additional compute nodes. For this analysis we ignore the cost of the new compute nodes and focus exclusively on the network costs. Additional nodes require additional network switch ports and cabling. The individual port cost listed at the bottom of Table 10-1 are calculated by dividing the cost of a switch by the number of nodes that can be connected to it. In this case, a $20,000 200Gb/s InfiniBand switch with 40 ports is equipped with $240 cable splitters to serve 80 100Gb/s nodes. As such, a new node costs $490 in additional network hardware ($250 for its switch port, $120 for a half a cable splitter, and $120 for a cable). However, these estimates ignore the cost of increasing upstream networking to ensure that sufficient bandwidth exists between switches to create a balanced network.

10.1.3. *Cost Discussion*

There is no denying that current generation SmartNICs are expensive. A \$2,000 NIC is more than most desktop systems and would be difficult to justify if standalone computing performance was the only point of comparison. However, in the context of building HPC platforms that seek to maximize the amount of computing power available in a fixed amount of space, a 10% increase in node cost may be acceptable. These gains may be more significant if extending the network fabric to support other compute nodes would be cost or space prohibitive.

10.2. **Estimating Glinda's Power Use**

The second place where SmartNICs add to the cost of hosting a computing platform is in their power use. SmartNICs supplement the communication ASIC with processor cores and require extra parts to be added to the NIC such as memory and nonvolatile storage. These components add to the overall power consumption of the card and require extra cooling from the data center. While NIC power use has traditionally been low enough that vendors do not typically report it, high-speed NICs such as the 100Gb/s Ethernet Intel E810 specify idle and max power rates of 14.9W and 16.6W for copper connections. In this section we report on power measurements for the Glinda platform and calculate how much of a financial burden SmartNICs add to Glinda's yearly operational expenses.

10.2.1. *Glinda Power Monitoring Capabilities*

Each Glinda compute node includes a baseboard management controller (BMC) that has access to a variety of sensors that are distributed throughout the chassis. An administrator can easily query the sensors of a node remotely through IPMI at a frequency of approximately one sample per second. There are several useful sensors available through IPMI:

- **`SYS_POWER`**: The `SYS_POWER` sensor provides the overall amount of power the entire node is currently consuming in watts. This value unfortunately is coarse grained and has a resolution of 25W.

- **`12V_GPUx`**: The Glinda node has four DC power connectors that are intended to be connected to GPUs in the node's four, larger PCIe bays. Each power connector splits into a small connector for a riser and a standard 8-pin auxiliary power connector for a GPU. The vendor has indicated that current sensor measurements cover both connections. Each current sensor has a resolution of 0.32A (i.e., 3.84W at 12V DC). `12V_GPU0` is connected to the A100 card. `12V_GPU2` from the front-middle bay has been routed to the BlueField-2.

- **Temperatures**: There are a variety of temperature sensors throughout the chassis. From a sample reading of an idle node, we observed cool air at the inlet (21°C) passing through the GPU (26°C), CPU (32°C), DIMMs (39°C), NVMe (39°C), power supply (31°C), and PCIe slots (59°C).

As a means of illuminating how much power is consumed by different components in the system, we collected power measurements from the BMC while different stress tests executed in parallel on the node. For this experiment we launched each test individually and waited approximately 30 seconds before starting the next test. The overall system power usage (i.e., SYS_POWER), GPU auxiliary power (i.e., $12V \times 12V\_GPU0$), and BlueField-2 auxiliary power (i.e., $12V \times 12V\_GPU2$) were captured during seven stages of activity, listed below.
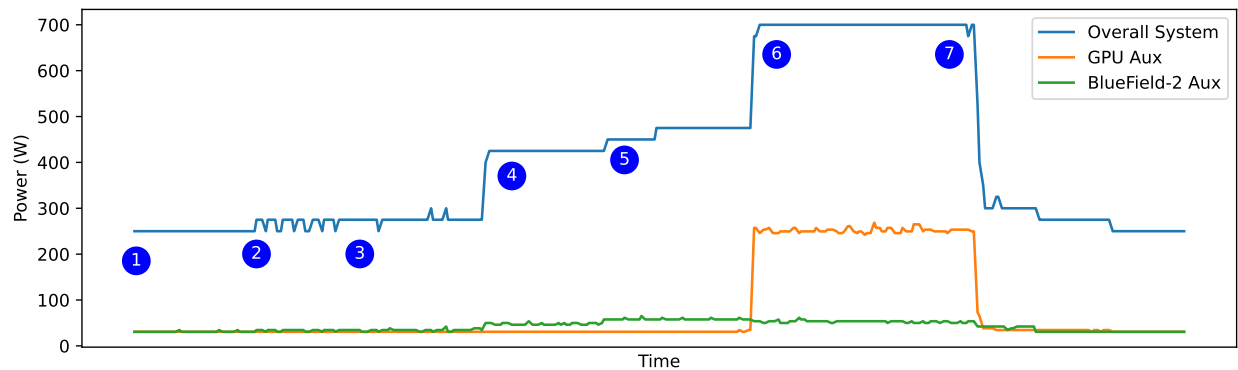
1. **Idle**: The host initially starts in an idle state with all hardware components booted.

2. **Host-to-Host Network Traffic**: Next, the `ib_send_bw` tool is started on the host to continuously push data to a neighboring node.

3. **Host-to-SmartNIC Network Traffic**: A second instance of `ib_send_bw` is then started to push data to the local SmartNIC CPUs.

4. **Host `stress-ng`**: A stress test is run on the host to maximize host CPU activity.

5. **SmartNIC `stress-ng`**: An additional stress test is then run on the SmartNIC to place load on its Arm cores.

6. **GPU Load Test**: The NVIDIA `dcgmi` tool is launched to maximize GPU activity.

7. **Shutdown**: Finally, all tests are stopped to allow the system to return to an idle state.

Figure 10-1. Power Measurements for a Glinda Node During Stress Tests

Power measurements during the stress test are presented in Figure 10-1. The Glinda node in this test consumed 250W while idle and 700W with all resources active. The largest jumps in power took place when the GPU test 6 (+225W) and host stress test 4 (+150W) activated. The

SmartNIC stress tests ⑤ (+50W) and InfiniBand operations ② ③ (+25W) had less impact on the overall power consumption of the node.

There are multiple sources of uncertainty that cloud an assessment of power use in a Glinda node. First, the 25W resolution of the `SYS_POWER` sensor makes it difficult to get an accurate measurement of power use in the system. As the InfiniBand transfer tests indicate, there are several important operations in the node that fall below a 25W cutoff. Second, it was unclear to us whether a `12V_GPUx` sensor monitors the PCIe card's auxiliary power connector, the riser's power connector, or both. Finally, we noticed that the BlueField-2's power use increased at times when the card was idle (e.g., ④). These measurements imply that the `12V_GPU2` measurements may include more than just the BlueField-2's power usage.

In order to gain more insight into the Glinda node's power characteristics, we examined the A100 and BlueField-2 cards individually.

10.2.3.   Ampere A100 Power Use

The Ampere A100 card is listed as having a maximum sustained power consumption of 250W. The IPMI sensor for the GPU's auxiliary power connector reported that an idle card used 30.72W and a fully-loaded card used up to 268W. The A100 card has additional, internal power and temperature sensors that can be queried through the CUDA libraries. The `nvidia-smi` tool reported the card used 250W while running the stress test. An inspection of the front riser card that holds the A100 reveals that it is a minimal circuit board that merges two PCIe data cables from the motherboard and one power connector into a standard PCIe slot. Given that current measurements for an unloaded front riser board were 0A and the power measurements of an active A100 match the A100's internal estimates, we conclude that the `12V_GPU0` sensor provides an accurate measurement of the A100 card's total power use.

10.2.4.   BlueField-2 Power Use

The power specifications for a P-Series BlueField-2 indicate that a 16GB card has a maximum power consumption of 63W. Our initial power measurements with the `12V_GPU2` sensor reported that the BlueField-2 consumed 30.72W when the node was idle, 42.24W when the SmartNIC alone was active, and 65.28W when the CPU, SmartNIC, and GPU were fully loaded. However, the 65.28W was also observed in instances when the host CPU was active and the BlueField-2 was idle. An inspection of the back-middle riser card revealed that it was more complex than the front riser: in addition to being able to host two PCIe cards, the riser connects to the motherboard through a dedicated slot. While the back riser uses the same 12V power connector as the front risers, we suspect that the system can move power from motherboard to riser and riser to motherboard as needed. As such, the `12V_GPU2` sensor is not a good indicator of BlueField-2 power use when the CPU is active.

To test this hypothesis we moved the BlueField-2 to the front-left bay and attached the `12V_GPU3` connector to it while leaving `12V_GPU2` connected to the back riser card. Power measurements for different workloads are presented in Table 10-2. As we suspected, the BlueField-2's idle power

Table 10-2. Power Use for Individual Slots in Different Scenarios

Slot	Idle	CPU Active	CPU + SmartNIC Active
Empty Front Slot	0W	0W	0W
SmartNIC in Back Slot	30.72W	65.28W	65.28W
SmartNIC in Front Slot	30.72W	30.72W	42.24W
Empty Back Slot	11.52W	26.88W	26.88W

consumption remained at 30.72W whether the host CPU was active or not. Placing load on the Arms increased power use to 42.24W. In contrast the empty, back-middle riser card jumped from 11.52W to 26.88W when the host CPU changed from idle to active. Given that an empty front slot did not consume power, we expect that the 42.24W measurement is a realistic estimate of the Arm's active power use.

Additional attempts to push the BlueField-2 power consumption closer to its 63W limit were not successful. We experimented with oversubscribing the cores, executing continuous RDMA network transfers, and launching additional tasks such as hashing random data, but none of the additions increased the power use beyond the value observed during `stress-ng`'s execution. The hardware accelerators (e.g., compression) were not explored in these tests and may contribute to the device's 63W limit.

10.2.5. Power Discussion

Power estimates can be converted to a yearly operational cost by multiplying the power estimate by the cost of power and the number of hours in a year. In the Sandia California data center the cost of power in 2023 is 8.62 cents per kilowatt hour (kWh), with forecasts warning that rates may raise to 10.61 cents per kWh in the next four year. The SmartNIC's idle power measurement gives us a basis for computing the worst case scenario: *What if a cluster is procured with SmartNICs but they are never used?* The cost of the wasted power in this case would be:

$$0.03072 kW \times 0.0862 \frac{\$}{kWh} \times 8760 \frac{h}{year} = \frac{\$23.20}{year} \times Nodes$$

The 126-node Glinda cluster in this case would waste $2,922 per year. It is important to note that this is an upper bound estimate, as it ignores the fact that a cluster without SmartNICs would still employ an InfiniBand NIC that consumes a small amount of power.

In contrast, the SmartNIC's active power estimate gives us an estimate of the best case scenario: *What if a cluster is procured with SmartNICs and they are always busy with useful work?* The concern in this scenario is that the power use could significantly increase the operating cost of the node. Fortunately, a 42.24W increase in power only results in an added cost of $31.90 per node to the yearly operating cost, or $4,019 for the cluster.

124

An important caveat to these estimates is that they are based on the power use of the SmartNIC alone and do not include the cost of cooling. Current generation SmartNICs use passive air cooling, which requires proper air handling from the data center. Air cooling is typically estimated to cost the same price as the electricity cost of the computing hardware. Therefore, a more accurate operating cost for SmartNICs would be to simply double the previous estimates.

10.3.   Summary

When considering whether SmartNICs are a worthwhile addition to a cluster, system architects must weigh the costs of purchasing and operating the hardware with the benefits they may provide. There are multiple insights to be learned from the Glinda system. First, the $2,000 BlueField-2 SmartNICs were expensive, costing 2.35x more than a plain InfiniBand NIC. While NICs at these price points are difficult to justify for thin-slice nodes, they may be more suitable for fat-nodes that include many CPUs or GPUs. Second, SmartNICs can be a cost-effective means of increasing the compute power of a platform when space is limited or network fabric expansion costs are considered. SmartNICs provide a convenient way to insert computing resources without requiring more space or network ports. Third, we estimate a BlueField-2 SmartNIC in Glinda consumes 30.72W to 42.24W depending on load. Although the maximum power use was less than we expected, we encourage vendors to focus on reducing idle power use as much as possible. Finally, we estimate that the SmartNICs in the 126-node Glinda cluster add roughly $6,000 to $8,000 to the annual operating cost of the system when the added power and cooling requirements are included. These costs are not out of the ordinary in a data center, but they do stress the importance of insuring that the hardware is leveraged by users given that cluster are expected to have a useful lifetime of eight to ten years.

11.   CONCLUSION AND FUTURE WORK

Our hypothesis for this project was that adding SmartNICs to a portion of the compute nodes in a platform will improve the efficiency of the overall system because they will provide a way to offload data management and storage services from the host while still preserving locality. The work summarized in this report confirms this statement to be true, with minor caveats relating to practical issues. Our characterization of current-generation hardware indicates that while SmartNIC processors are an order of magnitude slower than host processors, SmartNICs have sufficient resources for hosting data transformation, staging, and distribution tasks that would otherwise tax the host's resources or require additional compute nodes. Experiments with the BlueField-2's compression hardware illustrate that targeted hardware accelerators for SmartNICs can have a significant effect on performance and are immediately usable in data-intensive workloads. In terms of operational costs, our analysis of the Glinda procurement confirms that SmartNICs are less of a financial burden than simply adding other host nodes, once power and infrastructure costs are included.

We have explored five examples of ways that SmartNICs can be leveraged to achieve gains in different types of data management tasks: compression, distributed data reorganization, in-transit data queries, serialization optimization, and job-local storage. In the context of a full system, the importance of each of these examples is as follows. First, serialization optimizations reduce the amount of time an application spends ejecting data and allows serialization to take place at the same time as transfer to the SmartNIC. Second, on-card compression accelerators enable us to rapidly reduce data in the serialization process for situations where space is more important than read performance. Third, the particle-sifting example demonstrates how a collection of SmartNICs can work together to autonomously reorganize data in an asynchronous manner. Fourth, query engines give users an opportunity to peek at the data as it migrates through the system. Finally, hosting job-local storage with SmartNICs enables data to be accumulated an large amounts, without disturbing the simulation or being disturbed by other IO-intensive jobs.

The caveat for this work is that there are a number of practical issues that have made these elegant examples challenging to realize in production hardware. As such, there is much work to be done before there will be broad adoption of SmartNICs in the HPC community. The largest issue is the lack of a standard programming environment for SmartNICs that is both flexible and portable between vendors. While the use of standard Linux distributions for the BlueField-2 SmartNIC's OS provides an easy way to port existing libraries and services to the hardware, the use of closed-source, vendor-specific APIs and overly-restrictive EULAs dampens enthusiasm for SmartNICs. A significant amount of our work focused creating an environment that could insulate us from hardware specifics and provide basic primitives for orchestrating transfers and dispatching computations. While Mochi could easily be substituted for Faodel, we strongly feel that Arrow is the leading choice for processing structured, in-transit data.

11.1. Challenges and Opportunities

Looking forward, we see that there are several challenges and opportunities for future SmartNIC.

Decreasing Idle Power Use: While an idle BlueField-2 SmartNIC only consumes 30W of power, it is important to remember that an HPC deployment typically contains thousands to tens of thousands of nodes. As such, there are future opportunities to determine what the optimal distribution of SmartNICs would be for a platform to maximize use while minimizing power. Similarly, there are opportunities for vendors to improve power utilization by either lowering idle power use or developing additional hardware accelerators that implement common tasks in a more power-efficient manner than software.

Coordinating Network Transmissions: One hardship of developing in-transit services for current-generation SmartNICs is that the Arm processors lack a means of detecting when the host is actively using the network links. As such the Arm may inadvertently interfere with the host's communication and impede the performance of the simulation. The traditional means of avoiding this contention is to either implement quality of service metrics in the network fabric or designate explicit time periods where the host has exclusive access to the network. Future work may investigate whether InfiniBand's existing QoS metrics can mitigate this problem.

Heterogeneous Architectures Complexity: While we did not experience any compatibility issues when exchanging data between the x86_64 hosts and the Arm processors, there was a significant development burden associated with working in a heterogeneous environment. In addition to needing to build and maintain our software stack for two architectures, inconsistencies in the runtime environment hindered progress. Upcoming Arm-based HPC platforms such as Grace/Hopper are therefore attractive for future work.

Ecosystem of Intelligent Devices: Similar to the networking community, storage vendors are beginning to place embedded processors in their computational storage devices (CSDs) to allow users to implement queries and data processing tasks in remote devices. As more of these compute-aware products enter the market, there is an opportunity to converge on a common set of libraries to allow users to interact with their data no matter where it exists in a massive, distributed platform. Apache Arrow is well-suited for this work. A next step for our push-down/push-back work would be to extend the pipeline into a collection of SmartNICs and CSDs. We envision a system where a query plan would be split out to multiple SmartNICs, which in turn fan the work out to multiple CSDs.

BIBLIOGRAPHY

[1] Robert B Ross, George Amvrosiadis, Philip Carns, Charles D Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K Gutierrez, Robert Latham, et al. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology*, 35(1):121–144, 2020.

[2] Ron A. Oldfield, David E. Womble, and Curtis C. Ober. Efficient parallel I/O in seismic imaging. *International Journal of High Performance Computing Applications*, 12(3):333–344, Fall 1998.

[3] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. In *hpdc2010*, pages 25–36, Chicago, IL, June 2010.

[4] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight I/O for scientific applications. In *cluster2006*, Barcelona, Spain, September 2006.

[5] David Rogers, Kenneth Moreland, Ron Oldfield, and Nathan Fabian. Data co-processing for extreme scale analysis. Technical Report SAND2013-1122, Sandia National Laboratories, March 2013. Level II ASC Milestone 4745.

[6] Ron A. Oldfield, Kenneth Moreland, Nathan Fabian, and David H. Rogers. Evaluation of methods to integrate analysis into a large-scale shock shock physics code. In *ics*, pages 83–92, Munich, Germany, June, 2014 2014. ACM Press. SAND2014-4608 C.

[7] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Joudain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, Michael E. Papka, and Scott Klasky. Examples of in transit visualization. In *2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, PDAC '11, pages 1–6, Seattle, WA, November 2011.

[8] Ning Liu, Jason Cope, Phil Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2012.

[9] Ron A Oldfield, Gregory D Sjaardema, II Lofstead, F Gerald, and Todd Kordenbrock. Trilinos I/O support (Trios). *Scientific Programming*, 20(2):181–196, 2012.

[10] Matthew Tyler Bettencourt, Richard Michael Jack Kramer, Keith Cartwright, Edward Geoffrey Phillips, Curtis C. Ober, Roger P. Pawlowski, Matthew Scot Swan, Irina Kalashnikova Tezaur, Eric T. Phipps, Sidafa Conde, Eric C Cyr, Craig D. Ulmer, Todd Henry Kordenbrock, Scott Larson Nicoll Levy, Gary J. Templet, Jonathan J. Hu, Paul Lin, Christian Alexander Glusa, Christopher Siefert, and Micheal W. Glass. ASC ATDM level 2 milestone #6358: Assess status of next generation components and physics models in EMPIRE. Technical Report SAND2018-10100, Sandia National Laboratories, 9 2018.

[11] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R Butt. bespokv: application tailored scale-out key-value stores. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 14–29. IEEE, 2018.

[12] Craig Ulmer, Shyamali Mukherjee, Gary Templet, Scott Levy, Jay Lofstead, Patrick Widener, Todd Kordenbrock, and Margaret Lawson. Faodel: Data management for next-generation application workflows. In *Proceedings of the 9th Workshop on Scientific Cloud Computing*, 2018.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[14] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Jianshen Liu, Carlos Maltzahn, Matthew L Curry, and Craig Ulmer. Processing particle data flows with smartnics. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2022.

[17] Craig Ulmer, Jianshen Liu, Carlos Maltzahn, and Matthew L Curry. Extending composable data services into SmartNICs. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 953–959. IEEE, 2023.

[18] Jianshen Liu, Carlos Maltzahn, and Craig Ulmer. Opportunistic query execution on smartnics for analyzing in-transit data. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2023.

[19] Jianshen Liu. *Extending Composable Data Services to the Realm of Embedded Systems*. PhD thesis, University of California, Santa Cruz, 2023.

[20] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. Performance characteristics of the BlueField-2 SmartNIC. *arXiv:2105.06619*, 2021.

[21] Craig Ulmer, Jerry Friesen, and Joseph Kenny. Glinda: An HPDA cluster with Ampere A100 GPUs and BlueField-2 VPI SmartNICs. Technical report, Sandia National Laboratories, Livermore, CA (United States), 2023.

[22] John Shawger and Matthew Curry. Offloading node-local filesystems in high performance computing environments. Technical report, Sandia National Labs., Albuquerque, NM (United States), 2023.

[23] Stephen Bates. Computational Storage Real World Deployments. `https://www.snia.org/educational-library/` `fms-2020-computational-storage-track-computational-storage-real-world`, November 2020. publisher: Flash Memory Summit.

[24] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10. IEEE, 2016.

[25] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 681–693, 2020.

[26] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[27] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 106–122, 2021.

[28] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[29] Ming Liu, Tianyi Cui, Henry N. Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using iPipe. *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[30] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient smartNIC offload of a distributed file system with pipeline parallelism. In van Renesse and Zeldovich [117], pages 756–771.

[31] Whit Schonbein, Ryan E. Grant, Matthew G. F. Dosanjh, and Dorian Arnold. Inca: In-network compute assistance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Robert Ricci, Eric Eide, and the CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX login;*, 39(6), 2014.

[33] Colin Ian King. Stress-ng. "https://wiki.ubuntu.com/Kernel/Reference/stress-ng", 2023. [Online; accessed 1-October-2023].

[34] Linux kernel crypto api user space interface — the linux kernel documentation. "https://www.kernel.org/doc/html/v5.11/crypto/userspace-if.html", 2023. [Online; accessed 1-October-2023].

[35] Rfc: Crypto api user-interface. "https://lwn.net/Articles/410833/", 2010. [Online; accessed 1-October-2023].

[36] Sysvipc(7) - linux manual page. "https://man7.org/linux/man-pages/man7/sysvipc.7.html". [Online; accessed 1-October-2023].

[37] Futex(2) - linux manual page. "https://man7.org/linux/man-pages/man7/sysvipc.7.html". [Online; accessed 1-October-2023].

[38] Malloc(3) - linux manual page. "https://man7.org/linux/man-pages/man3/realloc.3.html". [Online; accessed 1-October-2023].

[39] Sendfile(2) - linux manual page. "https://man7.org/linux/man-pages/man2/sendfile.2.html". [Online; accessed 1-October-2023].

[40] Tee(2) - linux manual page. "https://man7.org/linux/man-pages/man2/tee.2.html". [Online; accessed 1-October-2023].

[41] Splice(2) - linux manual page.
"https://man7.org/linux/man-pages/man2/splice.2.html". [Online; accessed
1-October-2023].

[42] Mlock(2) - linux manual page.
"https://man7.org/linux/man-pages/man2/mlock.2.html". [Online; accessed
1-October-2023].

[43] Msync(2) - linux manual page.
"https://man7.org/linux/man-pages/man2/msync.2.html". [Online; accessed
1-October-2023].

[44] Swapon(2) - linux manual page.
"https://man7.org/linux/man-pages/man8/swapon.8.html". [Online; accessed
1-October-2023].

[45] Madvise(2) - linux manual page.
"https://man7.org/linux/man-pages/man2/madvise.2.html". [Online; accessed
1-October-2023].

[46] Ajay Tirumala. Iperf: The tcp/udp bandwidth measurement tool.
http://dast.nlanr.net/Projects/Iperf/, 1999.

[47] nuttcp. "https://www.nuttcp.net". [Online; accessed 1-October-2023].

[48] Rick Jones. Netperf. "https://hewlettpackard.github.io/netperf/". [Online;
accessed 1-October-2023].

[49] Robert Olsson. Pktgen the linux packet generator. In *Proceedings of the Linux Symposium,
Ottawa, Canada*, volume 2, pages 11–24, 2005.

[50] Daniel Turull, Peter Sjödin, and Robert Olsson. Pktgen: Measuring performance on high
speed networks. *Computer communications*, 82:39–48, 2016.

[51] iperf3 at 40gbps and above. "https://fasterdata.es.net/performance-testing/
network-troubleshooting-tools/iperf/multi-stream-iperf3/". [Online;
accessed 1-October-2023].

[52] Quynh H Dang. Secure hash standard. 2015.

[53] Craig Ulmer, Chris Wood, and Sudhakar Yalamanchili. Active sans: Hardware support for
integrating computation and communication. In *Proceedings of the Workshop on Novel
Uses of System Area Networks at HPCA*, 03 2002.

[54] Gavin Matthew Baker, Matthew Tyler Bettencourt, Steven W. Bova, Ken Franko, Marc
Gamell, Ryan Grant, Simon David Hammond, David S Hollman, Samuel Knight, Hemanth
Kolla, Paul Lin, Stephen Lecler Olivier, Gregory D. Sjaardema, Nicole Lemaster
Slattengren, Keita Teranishi, Jeremiah J Wilke, Janine Camille Bennett, Robert L. Clay,
Laxkimant Kale, Nikhil Jain, Eric Mikida, Alex Aiken, Michael Bauer, Wonchan Lee,
Elliott Slaughter, Sean Treichler, Martin Berzins, Todd Harman, Alan Humphreys, John
Schmidt, Dan Sunderland, Pat Mccormick, Samuel Gutierrez, Martin Shulz, Todd

Gamblin, and Peer-Timo Bremer. ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical Report SAND2015-8256PE, 9 2015.

[55] Kenneth Moreland, Christopher Sewell, William Usher, Li-ta Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, et al. Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE computer graphics and applications*, 36(3):48–58, 2016.

[56] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[57] Geoffrey Lentner. Shared memory high throughput computing with Apache Arrow. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, 2019.

[58] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. In *Proceedings of VLDB Endowment*, 2009.

[59] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[60] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. Citeseer, 2015.

[61] Acero: A C++ streaming execution engine. `https://arrow.apache.org/docs/cpp/streaming_execution.html`. [Online; accessed 2-February-2023].

[62] Gregory F. Pfister. An introduction to the InfiniBand architecture. *High Performance Mass Storage and Parallel I/O*, pages 617–632, 2001.

[63] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[64] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray XC series network. Technical Report WP-Aries01-1112, Cray Inc., 2012.

[65] Gary J. Templet, Matthew R. Glickman, Todd Kordenbrock, Scott Levy, Gerald Fredrick Lofstead, Jeff Mauldin, Thomas J. Otahal, Craig D. Ulmer, Patrick M. Widener, and Ron A. Oldfield. Data services for visualization and analysis ASC level II milestone (7186). Technical Report SAND2020-9451, Sandia National Laboratories, 2020.

[66] Jialin Liu, Quincey Koziol, Houjun Tang, François Tessier, Wahid Bhimji, Brandon Cook, Brian Austin, Suren Byna, Bhupender Thakur, Glenn Lockwood, Jack Deslippe, and Prabhat. Understanding the I/O performance gap between Cori KNL and Haswell. In *Proceedings of the 2017 Cray User Group Conference*, 2017.

[67] Deepak Vohra. Apache avro. In *Practical Hadoop Ecosystem*, pages 303–323. Springer, 2016.

[68] Balaswamy Vaddeman. Data formats. In *Beginning Apache Pig*, pages 201–208. Springer, 2016.

[69] Peter Deutsch. Deflate compressed data format specification version 1.3. Technical report, 1996.

[70] Thomas Boutell. Png (portable network graphics) specification version 1.0. Technical report, 1997.

[71] Jeffrey Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, Arthur van Hoff, and D Hellerstein. Delta encoding in http. Technical report, 2002.

[72] Scott Hollenbeck. Transport layer security protocol compression methods. Technical report, 2004.

[73] Tatu Ylonen and Chris Lonvick. The secure shell (ssh) transport layer protocol. Technical report, 2006.

[74] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. Supporting fine-grained network functions through intel dpdk. In *2014 Third European Workshop on Software Defined Networks*, pages 1–6. IEEE, 2014.

[75] Linux Foundation. Data plane development kit (DPDK), 2015.

[76] Jianshen Liu. Simplify accessing hardware compression/decompression accelerators, 7 2022.

[77] Sabrina Amrouche, Laurent Basara, Paolo Calafiura, Victor Estrade, Steven Farrell, Diogo R Ferreira, Liam Finnie, Nicole Finnie, Cécile Germain, Vladimir Vava Gligorov, et al. The tracking machine learning challenge: accuracy phase. In *The NeurIPS'18 Competition*, pages 231–264. Springer, 2020.

[78] Matthias Schäfer, Martin Strohmeier, Vincent Lenders, Ivan Martinovic, and Matthias Wilhelm. Bringing up opensky: A large-scale ads-b sensor network for research. In *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, pages 83–94. IEEE, 2014.

[79] National Oceanic and Atmospheric Administration. Vessel Traffic: Ais vessel tracks, 2009-2017.

[80] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[81] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. {POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, 2020.

[82] Zhongzhe Xiong. Computational Storage: Data Compression and Database Computing Pushdown, April 2021.

[83] Francis H. Harlow. The particle-in-cell method for numerical solution of problems in fluid dynamics. Technical Report LADC-5288, Los Alamos National Laboratory, 1962.

[84] Konstantin Matyash, Ralf Schneider, Francesco Taccogna, Akiyoshi Hatayama, Savino Longo, M. R. Capitelli, David Tskhakaya, and Franz Xaver Bronold. Particle in cell simulation of low temperature laboratory plasmas. *Contributions to Plasma Physics*, 47, 2007.

[85] Eugene Fourkal, Bilal Shahine, Meisong Ding, J. S. Li, Toshiki Tajima, and C M Charlie Ma. Particle in cell simulation of laser-accelerated proton beams for radiation therapy. *Medical physics*, 29 12:2788–98, 2002.

[86] Yann Collet. LZ4 Frame Format Description, December 2020.

[87] Yann Collet and Murray Kucherawy. Zstandard compression and the application/zstd media type. Technical report, 2018.

[88] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[89] DuckDB Labs. Duckdb substrait extension - blob generation. https://duckdb.org/docs/extensions/substrait.html#blob-generation.

[90] Jacques Nadeau. Substrait: Cross-Language Serialization for Relational Algebra.

[91] Lee Rhodes, Kevin Lang, Alexander Saydakov, Justin Thaler, Edo Liberty, and Jon Malkin. Apache DataSketches: A software library of stochastic streaming algorithms. https://datasketches.apache.org/.

[92] Karel Youssefi and Eugene Wong. Query processing in a relational database management system. In *Fifth International Conference on Very Large Data Bases, 1979.*, pages 409–410. IEEE Computer Society, 1979.

[93] Joshua D Drake and John C Worsley. *Practical PostgreSQL.* " O'Reilly Media, Inc.", 2002.

[94] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[95] Anirban Dasgupta, Kevin Lang, Lee Rhodes, and Justin Thaler. A framework for estimating stream expression cardinalities. *arXiv preprint arXiv:1510.01455*, 2015.

[96] Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*, pages 71–78. IEEE, 2016.

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

[98] Huihuo Zheng, Venkatram Vishwanath, Quincey Koziol, Houjun Tang, John Ravi, John Mainzer, and Suren Byna. HDF5 cache VOL: efficient and scalable parallel I/O through caching data on node-local storage. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*, pages 61–70. IEEE, 2022.

[99] Jonathan Hines. Stepping up to Summit. *Computing in science & engineering*, 20(2):78–82, 2018.

[100] Sudharshan S Vazhkudai, Bronis R De Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. The design, deployment, and evaluation of the CORAL pre-exascale systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 661–672. IEEE, 2018.

[101] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K Panda, Hari Subramoni, Jérôme Vienne, and Gene Cooperman. System-level scalable checkpoint-restart for petascale computing. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 932–941. IEEE, 2016.

[102] Hideyuki Akimoto, Takuya Okamoto, Takahiro Kagami, Ken Seki, Kenichirou Sakai, Hiroaki Imade, Makoto Shinohara, and Shinji Sumimoto. File system and power management enhanced for supercomputer Fugaku. *Fujitsu Technical Review*, (3):2020–03, 2020.

[103] Kurt B. Ferreira, Patrick G. Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, page 19. IEEE/ACM, 2008.

[104] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of HPC applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30, 2021.

[105] Michael J Brim, Adam T Moody, Seung-Hwan Lim, Ross Miller, Swen Boehm, Cameron Stanavige, Kathryn M Mohror, and Sarp Oral. UnifyFS: A user-level shared file system for unified access to distributed local storage. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 290–300. IEEE, 2023.

[106] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs-a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 319–324. IEEE, 2018.

[107] Jan Heichler. An introduction to BeeGFS. `https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf`, 2014. Accessed: 2023-10-17.

[108] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[109] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[110] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[111] Tzahi Oved. Nvme over fabrics offload. In *OpenFabrics Alliance Workshop*, 2019. Accessed: 2023-08-22.

[112] Ryo Nakamura, Yohei Kuga, and Kunio Akashi. How beneficial is peer-to-peer DMA? In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 25–32, 2020.

[113] Nvm express revision 1.2.1. `https://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_1_Gold_20160603.pdf`. Accessed: 2023-08-22.

[114] Antoine Petitet, R. Clint Whaley, Jack Dongarra, Andy Cleary, and Piotr Luszczek. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed Memory Computers. `https://www.netlib.org/benchmark/hpl/`. Accessed 2023-08-21.

[115] Hao Lu, Piyush Sao, Michael Matheson, Ramakrishnan Kannan, Feiyi Wang, and Thomas Potok. Optimizing communication in 2d grid-based mpi applications at exascale. In *Proceedings of the 30th European MPI Users' Group Meeting*, pages 1–11, 2023.

[116] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–9, 2017.

[117] Robbert van Renesse and Nickolai Zeldovich, editors. *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 2021.

APPENDIX A. Expressing Computations in Arrow and Kokkos

As discussed in Chapter 4, Apache Arrow and Kokkos enable users to express a computation in a form that a runtime can map to parallel resources. The following code listings provide examples of how a computational kernel is expressed in each library.

A.1. Maximum

The maximum kernel returns the particle with the largest velocity magnitude.

Listing A.1 Maximum in Arrow

```cpp
arrow::acero::Declaration op_arrow_max(const arrow::acero::Declaration& source) {

    namespace cp = arrow::compute;

    // Specify how to compute the squared magnitude ((x^2 + y^2) + z^2)
    auto e_mag =
            cp::call("add", {
                    cp::call("add", {
                            cp::call("multiply", {cp::field_ref("X"), cp::field_ref("X")}),
                            cp::call("multiply", {cp::field_ref("Y"), cp::field_ref("Y")}),
                    }),
                    cp::call("multiply", {cp::field_ref("Z"), cp::field_ref("Z")}),
            });


    // Create a projection named f to compute the magnitude
    // Use an aggregate to pick a single value from an input vector
    return arrow::acero::Declaration::Sequence(
            {source,
             {"project", arrow::acero::ProjectNodeOptions({std::move(e_mag)}, {"f"})},
             {"aggregate", arrow::acero::AggregateNodeOptions{ {cp::Aggregate{"max", "f", "max(f)"} }}   }
            }
            );

}
```

Listing A.2 Maximum in Kokkos

```cpp
double op_kokkos_max(kblock_t kblock) {

    double result=0.0;

    // note: no need to allocate a view since the reduction generates a scalar

    // Walk through table and find rows to keep
    Kokkos::parallel_reduce("reduce:", kblock.extent_int(0), KOKKOS_LAMBDA (const int i, double &lval) {

        double tmp=0.0;
        for(int col=0; col<3; col++) {
            tmp += (kblock(i,col) * kblock(i,col));
        }
        if(tmp > lval) {
            lval = tmp;
        }
    }, Kokkos::Max<double>(result));

    return result;
}
```

A.2. Normalize

The normalize kernel converts a vector into a normalized vector with a magnitude.

Listing A.3 Normalize in Arrow

```
arrow::acero::Declaration op_arrow_normalization(const arrow::acero::Declaration& source) {

    // Convert x,y,z into magnitude, x_norm, y_norm, z_norm
    namespace cp = arrow::compute;

    // specify how to compute sqrt((x^2 + y^2) + z^2)
    auto e_mag =
            cp::call("sqrt", {
                    cp::call("add", {
                            cp::call("add", {
                                    cp::call("multiply", {cp::field_ref("X"), cp::field_ref("X")}),
                                    cp::call("multiply", {cp::field_ref("Y"), cp::field_ref("Y")}),
                            }),
                            cp::call("multiply", {cp::field_ref("Z"), cp::field_ref("Z")}),
                    })
            });

    // Build a plan with two projects:
    // First project passes X,Y,Z, and generates the magnitude
    // Second project does the divisions and passes the magnitude
    return arrow::acero::Declaration::Sequence(
            {source,
             {"project", arrow::acero::ProjectNodeOptions({cp::field_ref("X"), cp::field_ref("Y"), cp::field_ref("Z"), e_mag },
                                                          {"X", "Y", "Z", "Mag" })},

             {"project", arrow::acero::ProjectNodeOptions({cp::call("divide", {cp::field_ref("X"), cp::field_ref("Mag")}),
                                                           cp::call("divide", {cp::field_ref("Y"), cp::field_ref("Mag")}),
                                                           cp::call("divide", {cp::field_ref("Z"), cp::field_ref("Mag")}),
                                                           cp::field_ref("Mag") },
                                                          {"XN", "YN", "ZN", "Mag"})
             }
            });
}
```

Listing A.4 Normalize in Kokkos

```
kmag_t op_kokkos_normalization(kxyz_t kxyz) {

    int original_num = kxyz.extent_int(0);

    kmag_t kout("Norm", original_num); //Allocate the output view in advance

    Kokkos::parallel_for("Mag", original_num, [=] (const int64_t i) {
        double mag = sqrt((kxyz(i,0) * kxyz(i,0)) + (kxyz(i,1) * kxyz(i,1)) + (kxyz(i,2) * kxyz(i,2)) );
        kout(i,0) = kxyz(i,0) / mag;
        kout(i,1) = kxyz(i,1) / mag;
        kout(i,2) = kxyz(i,2) / mag;
        kout(i,3) = mag;

    });
    return kout;
}
```

A.3. Bounding Box

The bounding box kernel returns a list of particles that are within a specified region.

Listing A.5 Bounding Box in Arrow

```
arrow::acero::Declaration op_arrow_boundingbox(const arrow::acero::Declaration& source, double cbrt_of_r) {

    // select * from source where x < x_lt and y < y_lt and z < z_lt
    auto filter_expression =
            arrow::compute::and_({arrow::compute::less(arrow::compute::field_ref("X"), arrow::compute::literal(cbrt_of_r)),
                                  arrow::compute::less(arrow::compute::field_ref("Y"), arrow::compute::literal(cbrt_of_r)),
                                  arrow::compute::less(arrow::compute::field_ref("Z"), arrow::compute::literal(cbrt_of_r))});

    return arrow::acero::Declaration::Sequence(
            {source,
             {"filter", arrow::acero::FilterNodeOptions{std::move(filter_expression)}, "f"}});
}
```

Listing A.6 Bounding Box in Kokkos

```
kblock_t op_kokkos_boundingbox(kblock_t kblock, double cbrt_of_r) {

    int original_num = kblock.extent_int(0);

    if(original_num==0){
        kblock_t kout("ThreshParticles:", 0);
        return kout;
    }

    Kokkos::View<bool *> khits("Hits", original_num); //Place to store hit/miss
    Kokkos::View<int *> koffset("Offset", original_num); //Place to store offset of where it will go

    // Walk through table to find rows to keep. Create a hit mask to simplify output
    Kokkos::parallel_scan("DS1:", original_num, KOKKOS_LAMBDA (const int i, int &lval, const bool final) {

        bool hit = (kblock(i,0) <= cbrt_of_r) && (kblock(i,1) <= cbrt_of_r) && (kblock(i,2) <= cbrt_of_r);

        koffset(i) = lval;
        if(hit) {
            lval++;
        }

    });

    //Figure out how many particles we'll keep and allocate an output view
    int found_count=koffset(original_num-1) + khits(original_num-1);
    kblock_t kout("BoundingParticles:", found_count);

    //Go through the list again. Each worker knows where it's part of the output starts
    Kokkos::parallel_for("DSCP2", found_count, KOKKOS_LAMBDA (const int& i) {
        if(khits(i)) {
            int row = koffset(i);
            for(int k=0; k<7; k++) {
                kout(row, k) = kblock(i, k);
            }
        }
    });

    return kout;
}
```

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
CA Technical Library	8551	cateclib@sandia.gov