UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EXTENDING COMPOSABLE DATA SERVICES TO THE REALM OF EMBEDDED SYSTEMS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND ENGINEERING

by

Jianshen Liu

June 2023

The Dissertation of Jianshen Liu
is approved:

Contents

List of Figures

List of Tables

Abstract

Extending Composable Data Services to the Realm of Embedded Systems

by

Jianshen Liu

The non-uniform improvement of computer hardware performance poses a significant challenge for contemporary data processing in managing the growing volume of data. General-purpose systems encounter obstacles such as design, power, and heat management that hinder their computing power improvement. As data processing becomes more expensive and the increasing performance demands from applications, academia and industry are evincing interest in offloading data services to embedded systems (i.e., system software that runs on peripherals such as storage or network devices) to improve data processing efficiency. Given the domain-specific nature of embedded systems, this approach opens up abundant research opportunities, particularly as more applications rely on big data analysis for insights.

Efficiently leveraging embedded systems for data services requires answering three critical questions concerning why, what, and how. The "why" question pertains to the potential benefits of offloading a data service to an embedded system. Answering this question requires developing a methodology that can accurately quantify the benefits by taking into account the embedded system's domain nature and the data service workload. The "what" question pertains to what data services to offload to an embedded system. Answering this question requires a comprehensive understanding of the intended system and function to identify potential matches for successful offloading. In this thesis, I focus specifically on composable data services, not only because they serve as fundamental building blocks in applications, but also because their composability allows for more convenient migration to diverse systems. The "how" question pertains to determining the strategies to use for offloading. Given that embedded systems are designed to operate within a constrained environment, effective offloading strategies are required to prevent suboptimal performance resulting from incapable or overloaded embedded systems.

This thesis makes contributions to addressing the challenges associated with each of these research questions. First, I develop a practical methodology focused on cost-benefit quantification and a mathematical model to evaluate the data availability benefit of offloading data services into storage devices. Second, I examine and evaluate composable data services in high-performance scientific workflows to identify potential functions suitable for offloading. Finally, I explore strategies aimed at reducing data processing overhead and scheduling workloads dynamically to improve performance efficiency for data services running on embedded systems.

This thesis is dedicated to my wife and daughter for their incredible support — I can't believe we made it! Also, a big shout-out to my parents and in-laws for all the support they've given us.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Carlos Maltzahn, for his unwavering guidance and strong support throughout my Ph.D. studies. His timely intervention revived my research every time I struggled, and his academic acumen helped me avoid unforeseen roadblocks. I also wish to thank Scott Brandt, Peter Alvaro, and Matthew Curry for their consistent and constructive feedback.

I would also like to express my gratitude to Craig Ulmer. His expertise constantly pushed me forward to achieve rewarding research results. I would like to extend my thanks to Philip Kufeldt as well. His expert mentoring helped me reach my first milestone, and I learned so much from his rigorous approach to research. I'm grateful to Jeff LeFevre for his meticulous advice and feedback. I sincerely thank Paul Stamwitz and Ike Nassi for their continuous critique of my preliminary experimental results. I'm grateful to Shel Finkelstein for broadening my perspective on the potential of my research.

Finally, I would like to thank all my senior friends in the research lab: Ivo Jimenez, Noah Watkins, and Michael Sevilla. Your words of encouragement helped me stay focused on my goal to graduate. I want to thank Aldrin Montana for his valuable feedback on my presentation. I cherish the time spent with Jayjeet Chakraborty, Holly Casaletto, Saheed Adepoju, Esmaeil Mirvakili, and Farid Zakaria. Your valuable suggestions and feedback have been greatly helpful.

This work was in part funded by National Science Foundation grant CNS-1764102 and under cooperative agreement OAC-1836650 (subcontractor of Princeton University), the US Department of Energy ASCR DE-NA0003525 (FWP 20-023266, subcontractor of Sandia National Labs), and the Center for Research in Open Source Software (cross.ucsc.edu).

Chapter 1

Introduction

The contemporary data processing landscape, characterized by large and complex data sets distributed across diverse domains, presents significant challenges to general-purpose systems. These challenges stem from constraints in semiconductor miniaturization [218], energy consumption [77], I/O bandwidth [226], scalability [3], and communication overheads [74]. Moreover, the trend towards resource disaggregation [176, 209, 104] within the computer industry exacerbates the gravity of these challenges. Despite these limitations, modern applications for machine learning, artificial intelligence, and high-performance computing continue to drive the ever-increasing demand for computing power to enable effective data analysis in the era of big data. This demand far surpasses the computing requirements of traditional applications for which general-purpose systems were originally designed.

This conundrum has propelled the development of embedded systems that couple computing with domain-specific resources, aimed at accelerating the performance of specific applications offloaded to the hardware. However, due to the non-general-purpose nature of embedded systems, comprehensive research is required on both software and hardware design to realize the full benefits of offloading. This involves investigating not only the suitable functions to offload and the software stacks that allow flexible offloading, data management, and communication protocols, but also coordinating with hosts and other embedded systems. Moreover, optimizing hardware resource compositions is paramount to ensure the cost-effectiveness of offloading to these systems.

This thesis aims to bridge the gap between hosts and embedded systems to enable the efficient execution of composable data services that commonly serve as fundamental building blocks in a wide range of applications. Composable data services, such as key-value stores, data queries, and redundancy functions, entail significant data movement overhead across multiple layers in general-purpose hosts. This is because data requires to travel from the source to host processors for function access, potentially consuming significant system I/O bandwidth and CPU cycles that could otherwise be available to applications. Offloading these services to embedded systems can enable data processing closer to the source, resulting in benefits such as reduced data service latency, host resource utilization, and energy consumption.

This thesis explores embedding data services in storage and network devices to reveal the advantages of various offloading schemes with domain-specific applications (e.g., key-value data access and in-transit data management). Embedded systems for storage differ from those for networks in that they manage the source of large volumes of persistent data, while the latter handle streaming data flows. While this difference results in distinct data inputs for offloaded functions, there are similarities in many aspects of managing the data and control planes, such as data processing frameworks and workload scheduling. Our research commences with constructing mechanisms that systematically quantify the offloading insights for functions of composable data services, followed by developing strategies that mitigate the limitations while exploiting the efficiency provided by embedded systems. Together, these enable more informed decision-making on leveraging embedded systems for composable data services.

1.1 Background

Advancements in computer technology are a constant pursuit. The exponential growth in transistor count started in the 1970s with microprocessors enabling personal computers [229], continued with embedded systems facilitating smartphones and mobile apps [116], and expanded to web 2.0 and cloud computing due to universal computing demand [84]. The rise of big data since the mid-2010s has stimulated the development of machine learning, artificial intelligence, and, later on, edge computing [214]. Computer hardware and applications share a symbiotic relationship, catalyzing each other to become more efficient and performant.

However, the advancement of technology is not uniform, especially when considering the hardware components of a computer (Figure 1.1). In the early days, processor performance increased at a faster rate than memory and storage performance. Yet, in the mid-2000s, this trend began to change with slower components accelerating their performance growth and the performance gap in relation to processors starting to narrow. For example, 10 Gbps network cards were available in 2004 [248], and only six years later, an improved version of ten times the performance, the 100 Gbps network cards, became available [249]. Notably, towards the close of the 2010s, performance improvements in processors and DRAM seemed to plateau, while the progress in network and storage components maintained its steady climb.

These changes in relative performance can have a significant impact on application and system software, which often incorporate fixed assumptions about the relative speed of hardware components. As these assumptions become out of date, they might require a complete rewrite of the software in order to effectively harness the performance improvements from the underlying hardware. This becomes especially important as new applications increasingly demand greater levels of performance to handle the challenges presented by big data. Unfortunately, general-purpose processors face significant obstacles in delivering more computing power per processor core [146], prompting industries and researchers to explore alternative means of meeting the ever-increasing power demands of modern applications.

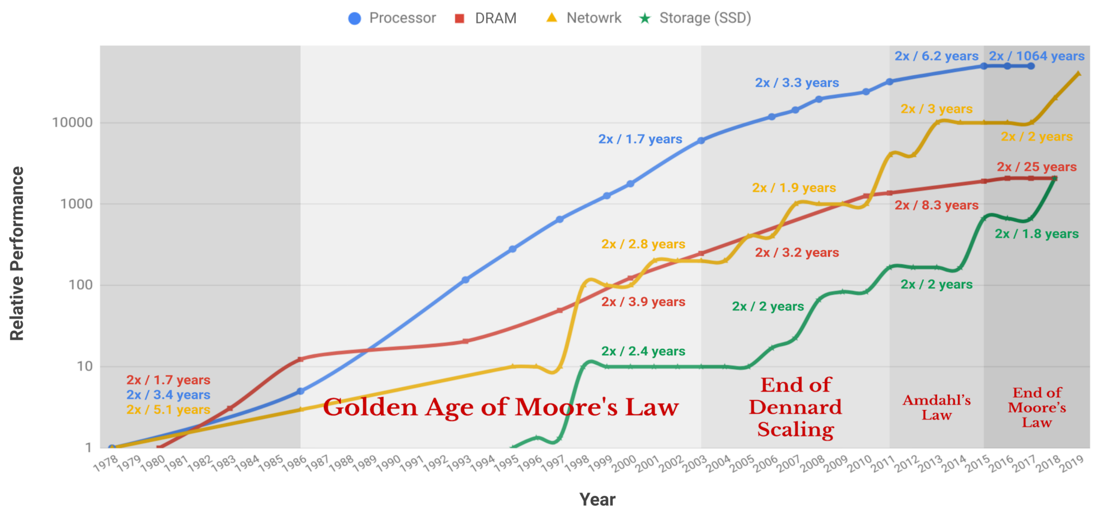


Figure 1.1: Performance growth in microprocessor, DRAM, network, and storage over 40 years[1]

[1] The processor performance data is sourced from the book "Computer Architecture: A Quantitative Approach" by John L. Hennessy and David A. Patterson, published in 2017 [107]. The data was obtained in relation to the VAX 11/780 using the SPEC integer benchmarks. The network and storage performance data is derived from Allen Samuels' 2016 presentation [216] titled "Consequences of Infinite Storage Bandwidth," while the memory performance data is obtained from the "List of Interface Bit Rates" on Wikipedia [252].

1.1.1 General-purpose Computing

The Power Wall

Prior to the 2000s, the semiconductor industry predominantly adhered to the trajectory of Moore's law, which facilitated a doubling of the number of transistors per computer chip every 18 months while simultaneously maintaining a relatively constant cost per chip area. During this era, computer system performance saw a marked escalation as clock rates increased, although such progress was largely constrained by the limited bandwidth of disk drives, which exhibited read/write performance that peaked at approximately 200 MB/s. As such, most applications operating on these systems preferred to transfer data from sluggish storage media to primary memory to accelerate processing. However, since the mid-2000s, the trajectory of microprocessor performance advancement has begun to curve owing to the breakdown of Dennard scaling [27]. Consequently, semiconductor manufacturers have found that regulating the power consumption of microprocessors has become an increasingly arduous endeavor compared to advancing the technology aimed at reducing transistor size. More specifically, the power consumption per silicon area has begun to surpass the power budget, which poses a significant challenge to counteract current leakage and prevent thermal runaway. This "power wall" has limited the increase in the practical clock rate to under 4 GHz to date [180] — same as server processors made a decade ago.

The breakdown of Dennard scaling played a significant role in driving the adoption of the multi-core architecture in microprocessors. With this design approach, the number of cores in a CPU socket rapidly grew from two to as many as 32 physical cores integrated into a single socket today. This shift in architecture meant that software developers could no longer rely solely on the performance enhancement of a single core, instead needing to bake different assumptions into programs about how they are separated into serial portions, and portions that can be parallelized with multiple cores. Nevertheless, several factors can impede parallelization efficiency, including the ratio of the parallelizable portion, workload data locality, lock contention, false sharing, power consumption, and inter-core communication. Amdahl's law [109] dictates that the maximum speedup that can be attained by parallelizing a computation is limited by the fraction of the computation that cannot be parallelized. Additionally, workloads with little data locality will eventually be bound by the memory channel bandwidth as the level of parallelization increases. Communication between cores also affects the gain from parallelization. Finally, power consumption challenges give rise to issues known as "dark silicon" [78, 105], which limits the number of cores that can be activated simultaneously.

Given these challenges, the semiconductor industry and researchers have been actively seeking solutions to some of these problems. One promising approach is the use of high-bandwidth memory (HBM) [120], which aims to increase the amount of low-latency memory accessible by processor cores. HBM uses a 3D-stacked memory architecture, enabling multiple layers of memory chips to be stacked on top of each other using through-silicon vias (TSVs) to enable communication between the layers. Another approach gaining traction is chiplet technology [266], which uses a modular design principle to integrate smaller chips with specific functions, such as processing cores, memory, or input/output interfaces, onto a larger substrate to form the final chip. These packaging technologies offer the potential for continued improvements in processor performance. However, concerns have been raised regarding carbon emissions during the manufacturing of these chips, which may exceed those from operational energy consumption [102].

Cost of Performance

Computing Power Efficiency: Data centers serve as the core component of the global digital infrastructure, enabling the storage and processing of the ever-increasing volume of digital data. However, the proliferation of data centers worldwide has raised concerns about their power consumption and energy efficiency. A previous study [208] indicated that in 2006, US data centers consumed almost 61 billion KWH, equivalent to 1.5% of the total energy consumption in the country. By 2011, this figure had surged to over 100 billion KWH. At a global scale, the energy consumption of data centers accounted for 1.1-1.5% of total global energy consumption in 2011. More recently, it has been reported that global data center instances increased by 550% between 2010 and 2018 [169]. Among the major contributors to energy consumption in data centers, running server tasks and operating refrigeration systems account for approximately 40% each. Importantly, servers consume a considerable amount of energy, even during idle periods. As electricity costs continue to increase, power bills have become a substantial expense for data centers [59]. Therefore, reducing energy consumption by servers and cooling systems is critical for the sustainable development of data centers.

Environmental Impacts: Carbon emissions resulting from computing are a major environmental impact that arises throughout the lifecycle of computer systems. As previously mentioned, operational energy consumption has experienced a significant rise, making it a substantial source of carbon emissions. In the era of big data, the prevalent use of machine learning [135] and deep neural network models [185] exacerbates this issue, given their substantial energy requirements. For example, a leading artificial intelligence company reported that the computational resources needed to train artificial intelligence models have been doubling every 3.4 months since 2012 [11]. Furthermore, since larger datasets and more parameters can generally improve model accuracy and capabilities, the current trend among technology companies to employ larger models has only intensified the demand for computing, leading to even greater carbon emissions. To mitigate the impact of machine learning and neural network training on the environment, researchers have developed estimators to predict the environmental impact of large-scale training and to help identify strategies to make the process more sustainable.

The manufacturing of computer hardware, particularly integrated circuits, also contributes significantly to carbon emissions. The modern semiconductor integrated circuits manufacturing process requires several hundred unique tools to perform hundreds of process steps to ensure controlled quality and final yield [20]. A recent study has shown that the fraction of life-cycle carbon emissions attributed to hardware manufacturing increased from 49% for the iPhone 3GS, released in 2008, to 86% for the iPhone 11 released one-decade later [102]. This highlights the critical role of computer hardware manufacturing in the carbon footprint of computing. Researchers are exploring greener manufacturing techniques to address this issue, such as using renewable energy sources [86] and designing more energy-efficient chips and productions.

1.1.2 Emerging Trends in Hardware Design Paradigms

Asymmetric Processors

The "power wall" and the limitations in scaling single-core performance have necessitated a paradigm shift in the approach to processor core design, which now involves trading off between functionality and performance. Traditionally, the processor design aimed to integrate as many transistors into each core while maintaining symmetric computing capability across all cores. However, as holistic improvement is limited by the power budget, sacrificing functionality for higher performance is becoming a necessary compromise. For instance, to enable higher parallelism, some cores may need to simplify their capabilities to reduce per-core area, allowing for more cores to be fitted on the same chip. Today, many modern processors utilize a hybrid architecture where a few cores are designed to be more powerful and power-intensive, while the others are less powerful but more efficient to enable more efficient parallelism. These less powerful cores can undertake tasks such as garbage collection, NUMA rebalancing, and security scanning while leaving off advanced functions [97] (e.g., the AVX-512 instruction set) that are only available on powerful cores.

Domain-specific Hardware

The trade-off between functionality and performance can extend beyond employing a hybrid architecture in processors to encompass the design of cores tailored specifically for only certain functions or workloads[2]. For instance, graphics processing units (GPUs) are equipped with cores intended for graphic processing workloads and scientific simulations. Owing to their specific focus, the design of each core on a GPU can be substantially simplified by excluding unnecessary functions. For example, while CPUs typically support double-precision arithmetic, most GPUs optimize mathematical calculations using single-precision and half-precision floating-point arithmetic [139, 168], which are generally adequate for graphics and scientific simulations. This simplification enables thousands of cores to be integrated onto a single chip to maximize parallelism for targeted workloads. Such a specialized design approach is also applicable to chips such as Google's tensor processing units (TPUs) [246], which aim to maximize the per-

[2]FPGA [134] is considered to trade flexibility for performance.

formance of matrix multiplications that are commonly used in machine learning and deep learning algorithms. More recently, data processing units (DPUs) [35], infrastructure processing units (IPUs) [34], and embedded storage devices [37] have also been developed to optimize a relatively small set of network- or storage-oriented operations by specifically allocating more semiconductor resources to these operations.

These specially designed microprocessors, along with peripheral devices incorporated to expose necessary functions, are commonly referred to as domain-specific embedded systems. Given the narrowed functionality of these systems, an important question arises regarding their desired placement within a general system to maximize their benefits. With the gradual reduction of the performance gap between storage devices and general-purpose processors, as well as the gap between network devices and general-purpose processors, one practical approach is to augment these devices to enable in-situ execution of higher-level domain-specific operations and leverage their improved performance with data locality. The recent emergence of embedded devices for computer systems, including programmable network interface cards (or SmartNICs) and computational storage devices [230], has highlighted this approach by enhancing one of these system devices.

The potential for energy savings and performance improvements from using embedded systems has been extensively researched in various application domains. Studies have demonstrated that the use of GPUs in scientific workloads and data analytics can result in power efficiency and computing performance that is several orders of magnitude higher than traditional CPU environments regardless of whether they are operating on a single core or multiple cores [112, 184]. In cloud computing, deploying SmartNICs for microservice-based applications or packet processing has been found to achieve significantly higher energy efficiency and lower overhead than running directly on host CPUs [162, 138]. Notably, a report [185] on energy consumption has claimed that data centers equipped with machine learning-oriented accelerators can be approximately 1.4 to 2x more energy efficient than typical data centers.

In the era of Moore's law, the development of domain-specific hardware has long been constrained by the limited lifetime due to the performance benefit being eclipsed by the next generation of processors that are typically only eighteen months away. This short period also poses a significant challenge in terms of achieving a return on investment for developing domain-specific hardware by amortizing sales. However, with the plateauing of single-core performance of general-purpose processors, the benefits of specialized hardware are not expected to diminish in the foreseeable future, and I anticipate an increase in vendors investing in the development of hardware tailored for specific application domains. As an exemplar, NVIDIA's BlueField SmartNIC has projected a roadmap that anticipates a 10x improvement every two years [35]. The upcoming BlueField-3 SmartNIC, with three times more transistors and double its predecessor's compute and network capability, is already available as engineering samples in the second quarter of 2023.

1.1.3 Data Services

Data services play an essential role in enabling efficient storage and retrieval of data by facilitating data resiliency, availability, validity, and curation, and are widely employed as fundamental building blocks across a diverse range of application domains. Web applications use data services to provide a semantically richer view of the underlying data and support advanced querying functionality [39].

In the context of high-performance computing (HPC) applications, data services serve as a critical interface for client applications to access and manipulate data while also enabling mechanisms for data storage, organization, and processing, such as key-value stores and message queues [210]. Storage and network systems, in particular, implement data services to offer a diverse range of characteristics that many applications can rely on, including provisioning, data protection, data availability, data performance, and data security [40].

Data Movement

The functionalities provided by data services require the movement of data across multiple system components. The process typically begins with ingesting relevant data from source devices into caches, followed by data processing and transmission of intermediate or final results to subsequent services or applications via appropriate channels. Depending on the scale of the data services, this process may involve frequent data movement across systems, leading to overhead observable by applications and requiring energy consumption by all relevant system components. Past research has demonstrated that a reduction in data movement of a key-value store can significantly improve lookup performance by order of magnitude [163]. Additionally, another study has found that the energy cost of moving data across the memory hierarchy is expected to be two orders of magnitude higher than the cost of performing a floating point operation with double precision [127]. To analyze the opportunities for reducing this data movement, I categorize data services based on the logical directions of the data movement they initiate.

The first type of data service moves data vertically, forming *north-south data movement* in the system hierarchy [132]. Data queries provided by databases are typical examples of this type of data service. The process typically starts with the movement of data blocks from storage media on a local storage device to the memory subsystem of the host. The host processor then deserializes the data to generate data tables stored in memory. In simplified terms, while executing the query, the processor reads the table data and writes the result back to memory for subsequent applications to consume. If data storage and processing are disaggregated across multiple systems, the amount of data movement on a system can increase significantly both internally and externally.

The second type of data service moves data horizontally between systems, constituting *east-west data movement* within the system hierarchy [132]. Typically, this type of service does not directly provide application-facing interfaces but focuses on functions enabling data reliability, availability, and scalability for scale-out systems. This type of service includes data recovery, data scrubbing [7], load balancing, failure discovery, and data tiering.

Streamlining Data Transfer

The advent of domain-specific hardware presents an opportunity to mitigate data movement overhead by partially or completely offloading data services to the hardware to leverage its specialized data processing capabilities. For data services that initiate north-south data movement, offloading these services can encapsulate the data movement in proximity to the source while minimizing the propagation of bandwidth and data processing power requirements to the other end of the traffic. A case in point is offloading SQL data services to SmartNICs or computational storage devices, allowing data to be filtered at the source and applications to receive the data, possibly without requiring additional resources for further processing [162]. For data services that transfer data between devices, offloading these transfers localizes the data movement overhead to lower levels of the memory hierarchy. For instance, data redundancy for distributed storage systems can be offloaded to SmartNICs, enabling host applications to rely on data availability assumptions at the onset when the data is written to the network, without concern for potential impacts from host failures [128].

1.1.4 Summary

Since the inception of computer technology, system architecture has experienced a myriad of evolutionary shifts to adapt to the diversity of applications. Notwithstanding the challenges inherent in general-purpose computing, such as the power wall and excessive data movement, the performance requirements of modern applications continue to rise. Consequently, there is an emergent need for more efficient solutions to balance the system-wide resource utilization for function execution. The sheer performance chasing of functions has thus been superseded by the pursuit of function efficiency. Extensive research has been conducted on storage- and network-oriented embedded systems to explore how existing functions or applications can enhance performance while reducing energy consumption. Additionally, researchers have recognized the potential benefits of leveraging specialized embedded devices that collectively provide data service endpoints constructed based on higher-level APIs as a more cost-effective approach to computing in the future.

1.2 Contributions

The scope of the thesis is to enable efficient function offloading for composable data services tailored toward exploiting the benefits of embedded systems. The thesis makes the following four contributions:

The first contribution is a novel methodology for quantifying the benefits of offloading data access functions to storage devices. This methodology encompasses a set of metrics designed to evaluate the efficiency of storage media performance used by a given data access function in terms of cost, power, and space. Unlike the speedup metrics commonly used in previous research, these metrics intend to guide the optimization of resource composition in embedded storage devices rather than merely disclosing changes in workload performance from offloading a particular data service function. Based on these metrics, a prototype is developed that implements the benefits quantification methodology and automates the process for normalizing the cost of performance with a specific focus on evaluating the benefits of offloading the key-value data access function. Through the use of this prototype, I demonstrate the methodology's effectiveness by exploring multiple distinct offloading landscapes that illustrate the quantitative impact of offloading configurations on the resulting benefits. A version of this work appears in HPC-IODC 2019 [155].

The second contribution is a mathematical model that explores the impact of embedding storage systems into storage devices on data availability. Data availability is a crucial metric for running data storage services, particularly in edge data centers where the tax of maintaining data availability is especially high, and offloading these services typically leads to increased data fragments being managed by individual storage devices. This model features parameters that capture the storage and compute capabilities of embedded storage nodes relative to those of general-purpose servers and incorporates the effects of diverse data replication schemes. The evaluation of this model mathematically illustrated that increasing the number of independent failure domains that a failover mechanism spans can significantly enhance the data availability of the system. As embedded storage nodes are generally more cost- and space-efficient than general-purpose servers, an edge data center can deploy a higher number of these independent nodes than servers for constructing the basic layer of a storage system, thus significantly improving data availability. The proposed model can serve as an instrumental tool for system architects, aiding them in pinpointing the optimal balance between system performance, cost, and data availability. This balance, in turn, contributes to the improvement of performance efficiency for storage systems in edge data centers. A version of this work appears in HotEdge 2020 [156].

The third contribution is a library that accelerates the performance of data serialization, motivated by the insight that compression poses a significant bottleneck to serialization efficiency. Data serialization is crucial in many data services, such as event streaming [87] and distributed data stores [243]. The library achieves acceleration by offloading the compression function to specialized hardware accelerators. Furthermore, the library exposes a generalized API that can be readily integrated into many serialization protocols and data management services, such as Apache Arrow IPC format [157] and Ceph BlueStore transparent compression [5]. Internally, the library leverages the parallelism and zero-copy capabilities inherent in the underlying hardware to maximize performance. As a concrete example, I demonstrate that it could accelerate Apache Arrow table serialization performance to an extent equivalent to utilizing an entire modern CPU socket. This performance enhancement is achieved while maintaining a similar compression ratio on the data output and occupying substantially fewer host CPU resources. This work highlights the necessity of adopting offloading strategies that can effectively harness the specific advantages of embedded systems to optimize the overall performance of data services. The work was presented at the

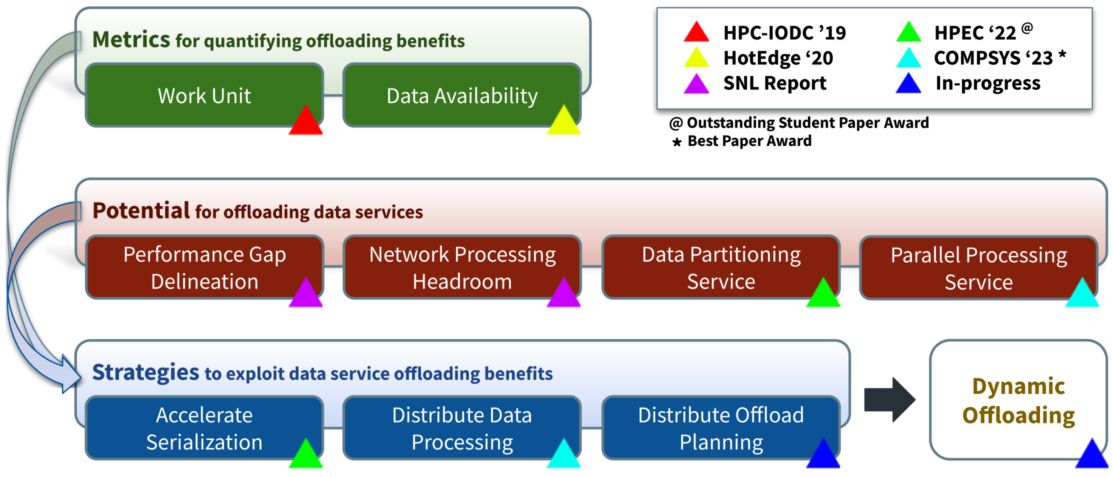
HPEC 2022 conference [157].

The final contribution is a mechanism for estimating the optimal placement of executing data query workloads on in-transit data, accompanied by a policy prototype that encapsulates this mechanism into a decision engine, designed to operate on SmartNICs. Data query services are crucial in many data analytic applications (e.g., data warehouses [101], business intelligence platforms [46, 201], and network data management [88]) for providing curated data flow to service consumers. While embedded systems can potentially improve the efficiency of query workloads, their utilization poses a significant challenge in scheduling workloads among systems of distinct architectures to avoid workload performance degradation caused by local system overload. As a core component of the decision engine, I implement a cardinality estimator to capture the job size of a query as an operations vector and employ machine learning-based predictive models to assess the time consumption of both execution and communication necessary for estimating the optimal placement of a workload. The proposed mechanism presents a generalized scheme to enable dynamic offloading in the context of embedded systems, with considerations and strategies tailored to resource-constrained environments to facilitate more efficient execution of data query services. This work is being submitted for publication.

In addition to academic publications, the contributions presented in this thesis and the corresponding prototypes have garnered considerable attention from the community. First, all of these contributions were funded either by the Center for Research in Open Source Software or Sandia National Laboratories. Second, the performance characterization of the BlueField-2 SmartNIC, which was a part of the research, was featured in the Next Platform and Data Centre Dynamics magazines as soon as I posted a draft of the paper on arXiv [159]. This work was also presented as a technical report authored by Sandia National Laboratories in 2021 [160]. Finally, the paper discussing the implementation of the library for improving the performance of data serialization utilizing hardware accelerations won the "Outstanding Student Paper" award at the 26th IEEE High-Performance Extreme Computing Conference. Moreover, the paper that described a data re-organization pipeline running on distributed SmartNICs was awarded the "Best Paper" award at the 2nd Workshop on Composable Systems.

1.3 Outline

An outline of the thesis is shown in Figure 1.2.



HPC-IODC '19
HotEdge '20
SNL Report
HPEC '22 @
COMPSYS '23 *
In-progress
@ Outstanding Student Paper Award
* Best Paper Award
Metrics for quantifying offloading benefits
Work Unit
Data Availability
Potential for offloading data services
Performance Gap Delineation
Network Processing Headroom
Data Partitioning Service
Parallel Processing Service
Strategies to exploit data service offloading benefits
Accelerate Serialization
Distribute Data Processing
Distribute Offload Planning
Dynamic Offloading

Figure 1.2: An outline of this thesis

Chapter 2 examines related work on the success and failure of some of the most important research efforts aimed at enabling embedded systems for specific use cases.

Chapter 3 begins by introducing a methodology for quantifying the cost-benefit of offloading data access functions to embedded storage devices. This methodology is based on a set of Media-based Work Units (MBWU) normalized metrics and is specifically designed to measure the efficiency in terms of cost, power, and space for a given data access workload. The second part of this chapter focuses on a perhaps surprising metric, namely data availability, for evaluating the benefit of offloading storage systems to embedded storage devices in the context of edge data centers. This metric is particularly important for the edge environment. Unlike traditional data centers, edge deployments operate on a smaller scale, making them more vulnerable to catastrophic failures when consisting of only a few failure domains with monolithic systems. Assuming edge deployments have more embedded storage devices than hosts, offloading storage systems functionality onto devices increases the number of failure domains and hence improves data availability. The evaluation of the data availability benefit is conducted through a mathematical model that takes into account the impacts of storage and computing aggregation within a storage node.

Chapter 4 explores the potential of offloading data services to SmartNICs. It begins with a thorough discussion of the methodology employed to create a performance envelope for a SmartNIC, which reveals potential functions that can benefit from offloading to the hardware. Subsequently, the chapter discusses the evaluation of the computing capacity of a network function that can be accommodated on a specific card under different network stacks and hardware configurations. Moving forward, this chapter explores data services provided by a collective of SmartNICs treated as network processing elements with use cases from high-performance computing workflows. Specifically, these use cases involve the partitioning of particle data flows and multi-threaded data processing.

Chapter 5 presents the strategies identified for offloading data services onto embedded systems. First, I introduce “Bitar,” a library designed to accelerate data serialization performance by utilizing hardware accelerators. Next, I discuss an embedded processing pipeline that distributes data objects across a cluster of SmartNICs and employs them to transform the object layout to suit the form required by subsequent applications. A version of this work was presented at COMPSYS 2023 [233]. Finally, the chapter examines a critical strategy: a mechanism for estimating the best possible workload placement, implemented as a decision engine. This mechanism enables the dynamic offloading of data query services to SmartNICs to strive toward optimized performance. This work builds upon previous research efforts in query optimization from the database and data science communities and adapts them to the unique context of embedded systems.

Chapter 6 concludes the key findings of the thesis and charts the course for ongoing and future work on top of the contributions made based on my understanding of the role of embedded systems in meeting the increasing demands of modern applications.

Chapter 2

Related Work

2.1 Specialized Data Processing Hardware

2.1.1 Channel I/O

Channel I/O [221, 8] is a concept that emerged in the 1950s to offload I/O functions to specialized hardware utilized in mainframes. Channel I/O allows mainframes to handle IO-intensive workloads in parallel, providing a significant advantage over other types of computers. Systems equipped with channel I/O initialize communication with channel hardware by executing a `start subchannel` instruction with an operand pointing to the starting address of the loaded channel program in memory. The channel hardware can then process I/Os from or to a set of controllers or devices without intervention from host CPUs. This process continues until a "success" or "failure" event occurs, which generates an I/O interrupt to the host. A "success" event indicates that the channel program has been completed successfully, with the intended data records being processed. In contrast, a "failure" event suggests that the channel program was terminated by an exception, requiring a future decision to proceed. The exception can be triggered by the channel program, subsystem, or any events the underlying storage controllers or devices raise.

Channel programs comprise channel command words (CCWs) that are decoded and executed by processors in channel hardware. Although these processors are usually RISC processors [186] that are less powerful than the host CPUs, they are designed to work in parallel, making them more efficient in I/O processing. Channel I/O can be considered a sophisticated form of direct memory access (DMA) [253] for microcomputers, except that it can fetch and execute I/O instructions without host CPU intervention.

The execution of a channel program is determined by the task for which it is designed. To filter raw data from storage before further execution, a channel program may use conditional branches. This dynamic execution capability allows host CPU cycles to be reserved for more complex operations from user programs, which would otherwise compete with data filtering functions even with the presence of DMA. The following code snippet provides an example of a channel program used in the IBM 2835 Storage Control and 2305 Fixed Head Storage Module [113]:

```
Seek
Search Key Equal
TIC *-8
Write Data
```

Listing 1: A channel program example

Each CCW in the code snippet provided consists of an eight-byte command with fields arranged in the following order: command code, data address, flags, and count. The `seek` command's data address specifies the location on the disk to seek, allowing the program to identify the cylinder and the track on the disk to start with. On-disk data records are organized in a count key data format [251].

With the `search key equal` command, the key field of the next record on the track is compared with the key at the location specified in the current command's data address. If the comparison fails, the program pointer is updated using the `TIC` command to return to the previous search command and continue the loop until the key comparison is satisfied. When the key comparison is successful, the storage control generates a status modifier bit to end the loop by moving the program pointer forward by 16 bytes to execute the write command, which writes the result data to the current record.

2.1.2 Specialized Hardware for Database Systems

In the 1970s, Lin et al. [150] devised a hardware system that implemented a content-addressing mechanism on head-per-track rotating storage devices to optimize tuple selection workloads in relational database systems of that era. Tuples represent the form of data stored on such devices, and each tuple is characterized by a size-fixed physical data layout comprising a predetermined number of tracks on disks. The collection of tracks spanning a tuple is known as a "band." The content-addressing mechanism comprises multiple search modules integrated into a storage device. Each search module is responsible for a contiguous sequence of bands, with the optimal number of bands allocated to each module being determined by balancing the cost of a module against the desired performance level of the workload.

This design was motivated by the low utilization of channels for transferring data between memory and head-per-track rotating storage devices. This design was potentially beneficial to specialized database systems, which were costly at the time and could thus accommodate dedicated designs tailored to specific data layouts. However, it is far from applicable to contemporary cost-efficient general-purpose storage servers, given that the data layouts on which they operate are typically workload-dependent.

Ozkarahan et al. [183] from the University of Toronto proposed an associative processor architecture for database management systems named “RAP”. The RAP infrastructure can communicate with general-purpose computers outside the infrastructure through programming language calls or I/O statements. Users are thus able to write programs with various query languages on general-purpose computers and send the query requests to the RAP infrastructure. Within the RAP, a set of cells is managed by function controllers. Each cell is an independent data processing microcomputer that works on its own track of data. The cell comprises an information search and manipulation unit and a register buffer of size 128 bytes, which is used to cache data during transfers between data on tracks and the logical unit. Data on a track is organized into back-to-back tuples, with each tuple representing a row corresponding to an entity in a relational data table. As the track of data moves through the cell, data can be filtered or manipulated by the query program running on the cell.

The RAP architecture represents a significant advancement over previous approaches that rely on search modules within head-per-track rotating storage devices. This is because RAP supports a data format that allows for variable data lengths and can perform a wider range of database management operations beyond mere search and filtering. Furthermore, RAP’s use of inexpensive large-capacity circulating memory devices makes it more cost-effective than the previous approach that employs customized storage devices.

Although RAP demonstrated superior performance over conventional database systems [182], DeWitt later identified in 1979 that RAP’s performance was still limited by its single-instruction multiple-data stream architecture [60]. As the data processing capability in cells is hardwired to data tracks, the utilization of cells is highly dependent on the relations being referenced in a query. The natural idea for improvement was to break the hard association with tracks and dynamically align the resource allocation for data processing according to the available resources and the query requirements at runtime. This change enabled the concurrent execution of queries from different users based on their priorities. The system created with this idea was named "DIRECT."

In this system, user queries are initially compiled into sequences of relational algebra operations by a host processor. These algebra operations are then evaluated in terms of complexity by a back-end controller and dynamically assigned to query processors based on their complexity. Unlike RAP and previous database system implementations, DIRECT slices a relation into equal-sized pages instead of tracks that contain a variable number of tuples. When a processor is assigned to a query, it operates only on the page to which it is linked at that time. Once an operation (e.g., scanning) on a page is completed, the query processor can request the next page that the query refers to but has not yet been processed by the back-end controller. When multiple users issue queries on the same relation, different query processors can share page caches from registers, thereby reducing memory consumption and supporting inter-query concurrency. Another key differentiating factor from other implementations with fixed computing resource assignments is that DIRECT preserves the view of data by copying the results that satisfy the search criterion to an associated memory buffer. This additional step of copying data eliminates the need to lock the relation while it is being evaluated by multiple query processors.

These early efforts focused on improving the performance of database query operations by increasing the level of parallelism for scanning and processing data from slow media. Storage devices have historically been a system bottleneck in workload performance improvement. However, beyond the efforts seen in database systems, few opportunities exist to generalize the pattern of developing specialized hardware for other types of workloads. This is primarily due to two reasons. First, relational database systems handle one of the most important workloads that deserve investment in designing and developing specialized hardware. Second, these systems abstract data into a standardized layout, thereby facilitating the discovery of solutions for standardized problems.

2.2 Active Storage Devices

2.2.1 Active Disks

Garth Gibson and Erik Riedel are pioneers in exploring the advantages of making storage devices the first-class citizens of storage systems. One of their early works aimed to demonstrate that distributed file systems constructed with network-attached disks capable of maintaining storage metadata (used to map client requests to disk sectors) offloaded from traditional file managers are more cost-efficient as they reduce the amount of server work per byte access [90].

To this end, the authors compared two different file system architectures, both of which utilized network-attached disks. The first architecture, "Network SCSI," sought to preserve the SCSI protocol [257] as much as possible, and clients could directly receive data from network-attached disks. However, since these disks could only handle SCSI commands while clients sent file requests using POSIX interfaces [255], a file manager was required to sit between clients and disks as the metadata server and guard data access permissions. This file manager was usually heavily loaded due to the work required to translate read and write requests, manipulate the namespace, and access control. As the file system of this architecture scaled up, the performance of the file manager became increasingly critical in ensuring sufficient bandwidth in serving the critical I/O path. Developing robust and efficient metadata servers for a file system has therefore been an important part of the development history of distributed storage systems. Modern solutions for managing metadata include utilizing a cluster of metadata servers and partitioning or distributing namespace to them using subtree partitioning [148] or pure hashing [55].

Another solution to alleviate the load of metadata servers in distributed storage systems is to adopt the flexibility of object storage. Accordingly, Garth's second architecture for comparison involved a system in which clients could directly communicate with network-attached disks using an object protocol. Disks used in this system were required to serve on object interfaces and maintain sufficient metadata information to translate object requests to locations of sectors. While the need for a file manager persisted, its role was significantly reduced to only mapping file names to objects and managing security aspects of requests, thereby removing it from the critical I/O path between clients and the smart disks. This narrowed scope of responsibility allowed the server hosting the file manager to be leaner and more cost-effective. This is a marked difference from the "Network SCSI" system, where the file manager had to translate every single data access for all clients. Although this idea was still in its nascent stages at the time, and no prototypes had been implemented, it serves as an important example of attempting to improve the cost-efficiency of a file system by offloading a higher-level data access function to storage devices.

"Active Disks," proposed in 1998, represented an important milestone that discussed the motivations and suggested possible workloads that could be offloaded to hard drives [203, 202, 204, 2]. The authors put forward three statements to support the idea of active disks:

- Performance of large database systems was limited by the bandwidth of interconnects of storage devices.
- Technology trends showed that the feature size of silicon process technology has continued to decrease, enabling the packaging of additional microcontrollers into a hard drive while maintaining its form factor.
- A growing number of applications rely on the sequential scanning of large amounts of data, such as multimedia and data mining applications. These applications are potential candidates for running on hard drives.

Although the aforementioned statements remain relevant today, the crux of the issue lies in the fact that having the space for packaging extra microcontrollers into a hard drive does not automatically guarantee that the drive can possess the requisite power to process the applications offloaded from hosts. Additionally, even if technically feasible, embedding a powerful microcontroller into a hard drive does not necessarily indicate that running a data-intensive function on the drive is more cost-effective than the traditional host-based approach.

The concept of active disks was initially founded on the premise that application candidates could efficiently run on hard drives, a notion that far exceeded the technological level at that time. Offloading functions to storage devices leads to programming and management overheads in hardware and software, which could offset or potentially outweigh the benefits of offloading application candidates.

Multiple research groups conducted the research work on active disks. One group, in particular, focused on making the concept more practical by implementing a stream-based programming model for task scheduling between hosts and active disks [2]. They employed an aggressive offloading approach to distribute most of the processing of an application to disks, using the host solely for coordination purposes. To ensure the security and integrity of data access, their programming model imposes strict restrictions on disk-side programs, which cannot allocate or free up memory, nor can they initiate I/O operations. The only operations that disk-side programs can perform are to scan the data streams provided by host-side programs and operate on the data with a small buffer. Both disk-side and host-side programs must be written for an application beforehand and compiled and manually moved to the corresponding locations before execution.

Keeton et al. [126] from UC Berkeley proposed a concept similar to active disks called “IDISKs.” However, it differs from active disks in that it allows inter-disk communication via a high-bandwidth dedicated network. In addition to the statements in support of active disks, the authors of IDISKs contended that the cost of system administration and cluster packaging for database systems was significant. Today, managing a data center is significantly less expensive than it was twenty years ago, owing to improvements in the technologies of automating data center management. Large data centers can now be maintained without interruption with the support of a small group of operations staff [115]. Additionally, the space efficiency of cluster packaging has improved notably since 1998 due to the availability of servers in multiple compact form factors.

The authors of IDISKs also anticipated that drives would continue to increase in processing power and memory capacity. However, the history of storage device development shows that vendors have mainly focused on reducing the cost per gigabyte. The increase of internal computing resources on a drive has been primarily to meet media management requirements. This is because, at the time, the benefits of offloading functions to hard drives were outweighed by the costs of hardware augmentation, infrastructure changes, and software modifications. Nevertheless, IDISKs proposed the first architecture in which storage devices were interconnected through a network. The authors also raised several relevant questions within the context of computational storage [17], such as how the IDISKs architecture will scale as the system size grows.

Unfortunately, despite the growing body of research in this area throughout the 1990s to 2000s, the computer industry exhibited little response to materialize augmented hard drives for executing offloaded functionalities from hosts. I posit that several reasons have contributed to this outcome:

- The cost-effectiveness of augmented hard drives was insufficient due to semiconductor technology limitations.
- The performance improvements in host computing resources competed with the potential benefits of offloading functions to storage devices.
- The POSIX data access interface [255] limited the capabilities of offloaded functions to operations involving only the scanning of byte streams.
- Finally, the lack of systematic research on benefits quantification, offloading mechanisms, and strategies has hindered the development of practical solutions within the industry.

2.2.2 Active Solid-state Drives

While both solid-state drives (SSDs) and hard drives offer the same block interface from the outside, they differ internally in many aspects. One important difference is in computing power. SSDs manage data using a subsystem called the flash translation layer (FTL), which maps logical block addresses to physical block addresses. This mapping is complex due to the fact that data on flash chips can only be erased at the block level, while data read can be done at the page level. Since a page is much smaller than a block, efficient data mapping requires intelligent and timely decision-making within SSDs. The FTL is also responsible for other complicated functions, such as garbage collection, wear-leveling, error correction code (ECC), and bad block management [123, 141]. Running all these functions requires an SSD to be equipped with more computing resources than those available in a hard drive.

Early research coined the term “active flash” to refer to SSDs that can perform operations beyond normal I/O requests, borrowing from the concept of “active disks.” While the motivation behind active disks was to decrease the server workload per byte access, active flash aimed to build energy-efficient domain-specific clusters using wimpy nodes, each consisting of a flash device integrated with a low-power processor and a small amount of DRAM. One such implementation is “FAWN [13],” which built a distributed key-value store consuming only one-tenth of the power consumption of a traditional server-based implementation without sacrificing performance requirements. Nodes in a FAWN cluster are either front-end nodes responsible for dispatching key-value requests or back-end nodes on which specific key-value operations are processed. The implementation offloads the key-value data access function to individual flash devices and organizes them in a way that exposes a single key-value namespace to clients.

In addition to power efficiency, the Gordon project [41] leveraged the bandwidth and latency provided by SSDs. A Gordon node is a compact PCB [256] with several hundred gigabytes of flash memory and a high-performance Atom processor. The authors of Gordon created a cost model to explore the design space of a Gordon cluster using storage devices of varying performance profiles. The results demonstrated that the performance gain from specialized hard drives was modest, while the gain from specialized SSDs was significant regarding both power savings and performance. These findings illuminated a reality observed in previous extensive research on active disks — despite their promising outcomes, only a handful were actually translated into products by the storage industry.

The difference in performance gain between specialized hard drives and SSDs lies in the overheads of the data path [212]. Specifically, the internal bandwidth of an SSD with sixteen channels and a single bank can easily exceed 6.4 GB/s, while the typical four lanes of PCIe Gen3 connection to an SSD can only provide 4 GB/s of duplex interconnection bandwidth. Additionally, an SSD can deliver read and write latencies below 10 µs, while the host-side I/O stack latency is over 20 µs, implying that the latter dominates the end-to-end latency. Extensive research is motivated by reducing these overheads and implementing in-storage computing to improve performance for various functions, including string matching, statistical calculations, data merging, and database scans [129]. However, due to the computing resource constraints on SSDs, many implementations involve FPGAs [118, 130, 142, 245], which incur costs from moving data between the data source and the computing elements built within FPGAs.

Today's technology has made significant advancements in the performance of storage interconnects, providing up to 4 GB/s per lane via PCIe Gen5. Additionally, the Linux community has been optimizing the data path [66] for high-bandwidth and low-latency storage devices such as NVMe SSD [244] and 3D-Xpoint [263]. These achievements may seem to eliminate the motivation for offloading functions to SSDs. However, with modern big-data workloads requiring data processing on hundreds or even thousands of nodes, the trends of serverless computing and disaggregating storage have increased the complexity of network involvement and, more importantly, the distance between processes and data. Mahapatra et al. [166] found that while these trends individually provide significant benefits, they collectively pose challenges. To address these challenges, they proposed a system architecture that can effectively utilize domain-specialized SSDs for serverless computing in storage-disaggregated data centers.

2.3 Programmable Network Interface Cards

The research efforts focused on enabling the programmability of network interface cards can be traced back to the mid-to-late 1990s. Singh et al. [220] in 1994 observed that network protocol processing was the bottleneck for supercomputers. They proposed a programmable network interface unit (NIU) for a specialized appliance called “Pixel Planes 5,” a custom, message-based multicomputer optimized for interactive graphics applications. Their goal was to address network issues, including sustaining high-bandwidth data transfer with minimal latency and supporting protocol research with sufficient programmability to process a wide range of upper-layer network protocols. The processing capability of the NIU was required by the use case that the Pixel Planes CPUs were exclusively dedicated to graphic processing and not anything else. Thus, all protocol processing had to be handled by the NIU. To meet these requirements, the NIU partitioned tasks between hardware and software. All data movement was performed by hardware to meet throughput requirements, while a microprocessor directed and initiated the data movement and performed protocol processing to ensure programmability.

SPINE [82] proposed an alternative approach to optimizing network performance for workstations rather than focusing on specialized applicants. This approach was motivated by the observation that improvements in I/O bus performance lagged behind the improvements in processor speed and I/O devices. To address this issue, SPINE provided an extensible runtime environment that enabled applications to compute directly on the network interface. Specifically, SPINE offered developers three key properties for constructing solutions tailored to specific applications: runtime adaptation and extensibility, performance, safety, and fault isolation. The first property allowed applications to define SPINE extensions at any privilege level and load them onto an intelligent I/O device. These extensions could directly transfer data using device-to-device DMA and communicate via peer-to-peer message queues to improve performance. Furthermore, these extensions were built with a type-safe language, enforced linking, and defensive interface design to ensure that the execution of any extension would not compromise the safety of other extensions running on the same NIC, firmware, or host operating system. The authors showcased an IP packet forwarding extension that ran on an embedded processor with a speed of 33 Mhz. Despite this low processing power, the extension achieved comparable throughput to a host-based IP forwarding system with a 200 Mhz CPU.

In 2000, Crowley et al. [56] identified several workloads suitable for programmable network interfaces, including packet classification or filtering, IP packet forwarding, data transcoding, and duplicate data suppression. While some of these workloads require limited processing of protocol headers, others necessitate substantial processing capacity to achieve the network link rate. The authors observed that the processing of one packet is usually independent of any other packet, allowing for packet-level parallelism and significant performance gains. To determine the necessary processor and memory features to support application-specific packet processing at the network link rate, the authors conducted experiments with different cache parameters and levels of parallelism for two types of workloads: one that processes a portion of a packet and another that performs computation over the entire packet. The results showed that fine-grain thread-level parallelism, supported by simultaneous multithreaded architectures, can exploit packet-level parallelism, aiding system architects in designing embedded processors to efficiently handle network packet-oriented workloads.

In recent years, the demand in modern applications for more advanced network functions, such as virtualization, security features, and cost-effective approaches to access data, has driven the emergence of programmable network interface cards, or SmartNICs, by various vendors. With the availability of this hardware, recent research has shifted its focus to designing architectures that enable different functions to offload to SmartNICs. One such architecture is iPipe [161], an actor-based framework that includes a runtime system for managing actors' execution on both hosts and SmartNICs and an offloading engine that decides which actors should be offloaded to the card based on their computation intensity. FairNIC [95], on the other hand, is a system designed to provide performance isolation between tenants using commodity SmartNICs. The system was developed with the observation that shared SmartNICs between different tenants can lead to unpredictable performance degradation, considering that leading data center operators have deployed these cards at scale to support network virtualization and application-specific tasks. E3 [162] is another microservice execution platform that accelerates microservice-based applications in the data center by offloading to SmartNICs. To achieve this, E3 uses the equal-cost multi-path load balancer as a key technique, which allows the system to maximize performance and minimize energy consumption. Additionally, E3 takes into account the layout of the cluster topology in the placement of microservices. An important feature of E3 is its ability to monitor the incoming/outgoing network throughput and packet queue depth with the traffic manager inside a SmartNIC to identify cases when the card is overloaded.

2.4 Peer-to-Peer Systems

Offloading data services that initiate east-west data movement require domain-specific hardware capable of peer-to-peer (P2P) communication. Peer-to-peer systems [207] consider nodes in the system as equal peers and decentralize most of the work to participating nodes that collectively provide specific services through the network. Due to the decentralized nature of such systems, they usually exhibit a high level of self-organization and strong resilience to faults and attacks. Early P2P systems like Freenet [50] and Gnutella [205] primarily arranged peers in unstructured P2P networks, where a query from a node was propagated through peers to locate the requested data. However, this protocol was not particularly efficient in system resources and network utilization, especially when searching for rare data. In response to these inefficiencies, P2P systems evolved to incorporate structured P2P networks, typically characterized by a topology maintained by a distributed hash table (DHT). This shift significantly improved the overall efficiency of these systems. Different P2P systems have different DHT implementations. Chord [222] is one of the popular structured P2P systems, with a DHT implementation with node searching/routing complexity of $O(\log(N))$, where N is the number of nodes in the system. Nodes in a Chord system are distributed in a ring structure, where each position represents the ID of a node. To facilitate request routing to other nodes, each node keeps two types of pointers: a pointer to the immediate successor and pointers in its finger table. A pointer consists of an IP address and port of a node. Routing with only the pointer to the immediate successor is not efficient, especially for a system with millions of nodes. To reduce the number of hops in routing a request, a node can look for a pointer in its finger table and forward the request to the known node that is closest to the node with the desired data. Pastry [211] has a different DHT implementation in which each node maintains pointers to its successor and predecessor nodes and pointers in a prefix-matching-based routing table. The routing table maintains a list of pointers whose nodes share the same ID prefixes with the current node, with different pointers associating matching prefixes of varying lengths. To select a node for a prefix, Pastry chooses a node with the shortest round-trip time since there may be multiple nodes that share a particular ID prefix with the current node. Therefore, among all the nodes in a routing table, the node associated with a short prefix is likely to be closer to the current node than the node associated with a long prefix.

2.5 Collectively Acting Specialized Devices

2.5.1 Network-connected Smart SSDs

CORFU [19, 247] is an early example that recognizes the potential of network-connected Smart SSDs, although it was not explicitly associated with the concept of function offloading. CORFU's objective was to enhance the utilization of SSDs by allowing multiple clients to concurrently access a cluster of SSDs while preserving performance efficiency and data consistency. It proposed delegating the metadata management of a storage system to SSDs and clients as an alternative to the centralized metadata server that would limit access bandwidth to an SSD cluster. CORFU treats a cluster of SSDs as a distributed shared log, and a sequencer acts as a centralized unit for bookkeeping the tail of the log. Clients maintain their views of metadata for mapping log offsets to the locations of data, and when a client needs to access the log, it queries the sequencer for the current tail position. The sequencer increases the internal counter by one to note the advance of the tail and returns the value to the client. Clients with the allocated position can calculate the corresponding flash device with their self-managed metadata and send requests directly to that device. It is worth noting that the sequencer is not a single point of failure, and clients without access to it can still look for the tail position of the log by contention. The sequencer only serves to optimize performance, and accessing it is not in the I/O path. Clients can freely write/read to/from the targeted flash device once they have reserved the corresponding log offset.

The CORFU system utilizes epoch numbers to account for configuration changes arising from device replacements or failures. Specifically, each request is tagged with an epoch number, allowing clients to issue a seal command to a subset of flash devices, signaling that the mapping to these devices is subject to modification. The sealed devices subsequently reject any requests with an epoch number equal to or lower than the sealed epoch. Meanwhile, the client who initiated the seal command prepares a map containing all the required fixes and writes the updated mapping to an auxiliary that durably stores the version history of all mappings. As a result, other clients who are rejected by flash devices can synchronize their mappings with the auxiliary, ensuring consistency across the system.

The distinctive attribute of flash devices in CORFU lies in their ability to autonomously manage the mapping of offsets to locations and respond to client requests. This feature enables all flash devices to collaboratively form a distributed shared log, resulting in high consistency and read/write performance that any single storage device cannot achieve.

BlueDBM [121, 122] represents another example of the potential of connected smart storage devices. This work is distinguished by two contributions. First, each host has an FPGA-based smart flash array connected via a PCIe bus, enabling the host to invoke accelerators implemented on the FPGA for data processing. Second, BlueDBM employs a high-performance inter-controller network that directly connects FPGA-based smart flash arrays without traversing the high-latency network stack in host software. This network unifies the address space of flash arrays residing in different hosts, allowing any host to view the entire storage space of the system while delegating address translation to the flash controller. Significantly, the host issuing the data request is oblivious to the address translation, efficiently bypassing the network stack and addressing overhead in software. BlueDBM offers three interfaces via a software library to user applications: a file-based interface, a block-based interface, and an accelerator interface. Notably, the FPGA-based flash array on a host does not directly expose a file-based interface. A user application seeking file access must first query the physical address and offset of the file by sending a request to the kernel file system, which knows the logical-to-physical address mapping because the FTL of the flash array is implemented in the device driver similar to that suggested by open-channel SSDs. Finally, the user application can send streams of physical addresses to the FPGA and leverage the accelerators to enhance data access.

2.5.2 Eusocial Storage Devices

Ants exhibit eusocial behavior [194] in which a colony is divided into groups with distinct responsibilities such as foraging, brood care, patrolling, and nest maintenance [92]. Group decision-making is crucial for achieving job efficiency, including searching for food, assigning ants to different tasks, and responding to external influences. Despite individual ants lacking the potential to perform these jobs, a large group demonstrates emergent intelligence to manage complex tasks without centralized management or high-level communication, relying solely on biologically programmed cycles [93]. Chemical signals are used for interaction among ants, which allow the group to accomplish tasks when combined with probabilities of receiving different signals. For example, ants mark paths to food with a chemical scent, and others follow the heaviest scent. The number of ants assigned to a task is also maintained using this mechanism [91]. Ants can remember the chance of encountering different types of ants and adjust their roles accordingly. Despite its simplicity, this mechanism enables simple participants to function effectively in complex ways.

The concept of leveraging the collective behavior of relatively simple entities to achieve high-level goals can also be applied to computer systems. For instance, in 2006, Brewer et al. [31] explored the potential of using collections of spinning hard drives to achieve global objectives related to input/output operations per second (IOPS) and storage capacity. Large data centers, such as those managed by Google, commonly utilize different generations and types of spinning hard drives. Although drives of later generations may offer higher storage density, their random seek performance may be degraded, leading to lower IOPS performance. As a result, hard drives in large data centers tend to have diverse performance characteristics. However, the workloads or applications running on these drives may have specific performance requirements defined in terms of service level objectives (SLOs) that not every drive involved can achieve. To address this issue, Brewer proposed leveraging the collective performance of hard drives to improve resource utilization and cost efficiency. The key to making this approach work is a carefully designed admission control system running on hosts capable of monitoring and adapting to changes in workload and the real-time status of various drives. Furthermore, the authors emphasized that the admission control system should only be responsible for the policy of input/output operations while leaving detailed operations, such as media management and error correction, to the hardware.

Kufeldt et al. [132] proposed the concept of eusocial storage devices in 2018, arguing that previous efforts to push back data management into storage devices, such as mapping/placement, scrubbing, redundancy, recovery, and accessibility, have failed due to the need for additional computing and memory, which increased per-GiB costs. The emergence of the smartphone market over the past decade has driven the cost of embedded processors below that of server processors. Additionally, more advanced and denser flash media have made it easier to hide the cost of computing resources within the per-GiB cost of flash devices. Eusocial storage devices embed computational capabilities and are self-managed by high-level APIs that abstract data placement, location, movement, availability, and recovery from the I/O path, and provide data management services through autonomous and collective behavior between devices. Eusocial storage devices can also be grouped into castes for scaling on a class-of-service basis. To offload data management services to these devices, the services that work on data traffic between applications and devices, the services that move data between different servers, and different computational requirements need to be mapped to different in-storage computing castes. Once these devices expose object-level service APIs through a crossbar network, client applications can interact with storage based on updated storage cluster configurations without considering the underlying system architecture. Meanwhile, system infrastructure maintenance can enjoy the flexibility of optimizing the disaggregated architecture of eusocial storage devices as long as the requested application-facing quality of service (QoS) is not violated. The paper highlights the need to change storage devices and how they can be federated to offer service APIs beyond the traditional block-based approach, as the performance scales differently in different hardware resources.

2.6 Workload Orchestration

As the programmability of embedded systems increases, the effectiveness of offloading certain functions to hardware may not always be optimal due to the real-time availability of resources on the hardware, particularly when embedded systems are shared among multiple workloads or application clients. Resolving this issue requires orchestrating the placements of workloads using strategies that can balance performance optimization targets and resource availability among placement alternatives. E3 [162] utilizes a reactive approach by migrating offloaded microservices only when overload is detected on the SmartNIC serving the microservice. Similarly, iPipe [161] employs a reactive approach but moves workloads among finer-grain execution environments. When the workload tail latency surpasses a threshold on a SmartNIC, the queued-up workload is migrated from the first-come-first-serve cores to cores reserved for scheduling tasks using the deficit round-robin algorithm. Additionally, when the mean request latency of workloads running on the first-come-first-serve cores on a SmartNIC exceeds a threshold, indicating the SmartNIC is overloaded, workloads on the waiting list are migrated to the host side for execution. In contrast, Clara [196] employs a predictive approach to determine the optimal number of cores on a SmartNIC for running a given network function program and traffic workload. This approach is achieved using machine learning techniques inspired by TVM [47], which trains cost models despite the vendor-specificity of accelerators (e.g., GPUs and TPUs), and infers effective optimizations for a given tensor program. By separating the "algorithm" (i.e., program logic) from its "schedule" (i.e., strategies of execution), TVM-like approaches search through the schedule space to identify effective optimizations. Similarly, λ-IO [264] utilizes predictive models to determine the placement of workload execution. To determine the parameters of the cost model, λ-IO profiles partial requests periodically. Specifically, for the first k requests in a period, λ-IO submits them to both the host and embedded system runtimes. Each runtime measures the values of profiling variables of a request during execution. After k requests are completed, it calculates the average of each variable and uses it as the values of parameters in the model.

2.7 Scope

Despite the extensive research that has been conducted on offloading functions to embedded devices, designing systems that achieve beneficial offloading for data services remains challenging. Specifically, the lack of meaningful metrics to quantify offloading benefits still hinders the development of effective solutions, especially for architects designing these embedded devices. Additionally, given the performance constraints inherent in embedded devices, strategies for device-side offload optimization driven by the varying availability of hardware resources and streaming data have yet to be explored. Although device-side offload optimization is particular to the devices themselves, similar to the standard query optimization employed in database systems, optimization strategies can potentially leverage derived information from outside of the devices. This thesis aims to bridge these gaps by developing methodologies to quantify the benefits of offloading and exploring optimization strategies for dynamic offloading of composable data services.

Chapter 3

Offloading Metrics

Embedded systems are specialized devices constructed with a customized resource composition to optimize their functionality for particular domains while adhering to specific limitations such as cost-effectiveness, power consumption, and space constraints. From the perspective of embedded system vendors, the purpose-built nature of these systems enables them to allocate resources judiciously, thus ensuring that each device can fulfill its intended purpose effectively. For instance, in the case of embedded storage devices, the device's primary focus may be on the storage media by directing the resource allocation towards exposing the storage media's optimal performance. Similarly, in the case of embedded network devices, the focus may be on maximizing network bandwidth, necessitating allocating resources toward achieving the highest possible bandwidth utilization.

One of the essential questions in research concerning function offloading to embedded systems is how to evaluate the benefits. Given that the evaluation necessitates consideration of both the offloading function and the target offloading system, two primary types of benefits evaluation exist. The first type involves varying the function to be offloaded while keeping the system constant. This approach enables application users to assess the performance of different offloading functions on a particular system. In contrast, the second type entails varying the system while keeping the function constant, which enables system architects to optimize the resource composition of a system to improve the cost-effectiveness of a given offloaded function.

In this chapter, we explore metrics that can effectively measure the benefits of offloading a given function to embedded storage devices. I present the concept of *Media-based Work Unit* (MBWU) and introduce MBWU-based efficiency metrics. These metrics provide a comprehensive and practical framework for normalizing and evaluating the cost-effectiveness of offloading a data access function to a specific storage device, as outlined in Section 3.1. Moreover, we observe that enabling storage devices to take on offloaded storage system functions increases the number of failure domains in the overall system and, therefore, can enhance data availability. Section 3.2 delivers a quantitative analysis of this data availability benefit, including a mathematical model that predicts data availability for different host/device ratios.

3.1 MBWU: Data Access Function Efficiency

Data access functions in storage systems refer to functions that facilitate communication with the persistent layer for reading and writing data. This includes functions such as get/put operations in key-value stores, read/write operations in filesystems, and select/project operations in database management systems. These functions are typically highly IO-intensive, thus providing an opportunity for optimization through offloading them to embedded storage devices, thereby directly harnessing the optimal media performance within the device.

However, the cost-optimal placement of a data access function is affected by two primary factors: the characteristics of the data access workload and the storage media. For example, latency-sensitive workloads running on slower media might favor execution by host CPUs to utilize the more substantial DRAM availability for cache, potentially leading to an adverse benefit from offloading. Conversely, throughput-sensitive workloads operating on faster media might benefit from offloading execution to storage devices, as this could reduce the bandwidth costs associated with transferring large volumes of data to host processors.

On the other hand, despite the recent progress made in function offloading to embedded storage devices, existing research has faced criticism regarding the efficiency of individually designed embedded devices. Specifically, it raises the question of whether the cost-benefit of offloading a specific function could be improved by optimizing the resource composition of the embedded device. Most current evaluation methods primarily rely on external characteristics like workload performance and system cost, using metrics such as MB/s, Kops/s, and $/ops. Unfortunately, these metrics fail to sufficiently capture the efficiency of the domain-specific nature of these devices, specifically the cost to exploit the storage performance of the devices for a specific workload. This cost could include aspects such as total cost of ownership, power consumption, and space usage during workload execution. By evaluating cost based on storage performance rather than workload performance, system architects can gain valuable insights into the efficiency of a system's storage resource utilization, allowing them to drive a balance between the system's resource allocation and the requirements of the offloaded function.

Reflecting on these considerations, to quantify the benefits of offloading a data access function from general-purpose systems to embedded devices — a process involving systems with significantly different architectures and design trade-offs — necessitates maintaining a consistent data access workload and storage media across these systems as a reference point for comparison. Moreover, it is crucial to account for the device's storage performance under the workload associated with the function. These requirements ought to be encapsulated into new metrics to standardize the process of quantifying the benefits of offloading data access functions to storage devices.

3.1.1 Metrics for Efficiency Evaluation

Normalization

To meet the previously outlined requirements for quantifying the benefits of offloading data access functions, we propose a novel, throughput-oriented metric for normalization, termed the **media-based work unit, or MBWU**, to encapsulate the storage performance of specific storage media under a given data access workload. In particular, one MBWU denotes the peak workload throughput performance, expressed in data access function calls per second, which can be achieved on a particular storage media while minimizing caching effects. To normalize the performance of a workload running on multiple storage media, we divide the performance value by the throughput value represented by an MBWU. This conversion translates the workload performance into a quantity (MBWUs) reflecting the amount of storage performance that the aggregated resources of the system running the workload can feasibly deliver. This metric exhibits the following three characteristics:

- **workload-dependent:** The MBWU serves as a workload reference since it captures the throughput performance for a specific workload. However, it is crucial to understand that an MBWU established for one workload is not interchangeable or applicable for normalizing the performance of a different workload.

- **storage media-dependent:** The MBWU also serves as a storage media reference, setting it apart from general performance evaluation metrics. Consequently, it is worth noting that a measured MBWU cannot be used to normalize the workload performance evaluated on a different storage media.

- **system-neutral:** The MBWU's dependence solely on a data access workload and storage media makes it a system-neutral metric. This property is particularly beneficial when examining the performance of a workload on a system that falls below the MBWU. Such a scenario suggests underutilization of the storage performance of the media on that system, indicating potential improvements in cost-effectiveness through system resource composition optimization. To maintain isolation from the impacts associated with specific systems when measuring the MBWU, minimizing the effects of caching is also essential. While caches can enhance a workload's performance, these improvements depend on factors beyond the storage media, causing the measured workload's performance to be system-dependent.

Optimizing the resource composition of a system has long been a challenging task for system architects. However, utilizing the MBWU metric makes it feasible to evaluate the utilization of a system's storage media capable performance for a given data access workload and quantify the degree to which storage resources are imbalanced in relation to other hardware components. This information can serve as a valuable reference in guiding the optimization of the system's overall resource composition.

To evaluate the utilization of a system's storage media performance for a given data access workload, we can calculate the ratio of achieved MBWUs on the system to the amount of storage media consumed by the workload. While some systems may have sufficient capacity to house multiple storage media, various constraints on the system may limit the workload's ability to fully utilize the aggregate performance of these media, resulting in suboptimal performance. Embedded storage devices, in particular, may emphasize maximizing the utilization of storage media performance while operating under various constraints to strike a balance between cost and performance.

MBWU-based Efficiency Metrics

Embedded storage devices encounter significant constraints, including power consumption, cost, and physical space limitations, that impede the achievement of optimal storage media performance for a given data access workload. By associating MBWU with investments in a storage system, it is possible to evaluate the cost necessary to enable the full performance of the storage media on the system. Specifically, we can proportion the investments, such as power, cost, and space, by dividing them by the normalized workload performance measured in MBWU. This approach allows for quantifying the cost-effectiveness associated with achieving a specific level of storage media performance while accounting for the various cost factors. In this regard, we propose three MBWU-based metrics for cost-effectiveness evaluation:

- **$/MBWU** for cost-efficiency: This calculates the total cost of ownership of the system for running the data access workload divided by the number of MBWUs achieved.
- **kWh/MBWU** for energy-efficiency: This calculates the system's energy consumption for running the data access workload divided by the number of MBWUs achieved. It is important to align the energy consumption measurement with the time frame the MBWU is measured. For example, if

the MBWU represents the workload performance per second, the energy consumption should also be measured as a per-second consumption.

- $\boldsymbol{m^3}$**/MBWU** for space-efficiency: This calculates the volume of the system divided by the number of MBWUs achieved. Different systems may have various space limitations, and for an embedded storage device, the space consumption may include the device's dimensions and the size of accessories required to support the device, such as substrate and interconnects.

3.1.2 Cost-benefit Quantification for Key-value Offloading

The ability to evaluate the cost-effectiveness of a data access function executed on a storage system provides important guidelines for system architects to optimize system resources. More importantly, it also enables quantifying cost benefits associated with offloading the function to systems with varying configurations but consistent storage media. As an illustrative case study, we conducted experiments to quantify the cost-benefit of offloading the key-value function, chosen due to the substantial data access overhead that this function incurs.

Overhead of Data Access Functions

We have selected the widely adopted RocksDB [38] to provide the key-value function for our experiment. RocksDB is an open-source, high-performance, embedded key-value store developed by Facebook, which is designed to offer efficient and scalable storage for various applications. It is built on top of LevelDB [89] and optimized for modern hardware architectures, providing high write and read throughput, low latency, and space-efficient storage. RocksDB employs *compaction* as a mechanism for managing the storage and performance of the database over time (Figure 3.1). This process involves merging multiple smaller sorted string table files into a larger, more efficient file when their size reaches a certain threshold. The compaction process operates asynchronously behind the scenes of user applications. However, due to frequent reads and writes during compaction, the amount of data accessed from the underlying storage device can be multiple times greater than the data accessed by the user application that submits the key-value requests, leading to data access amplification.

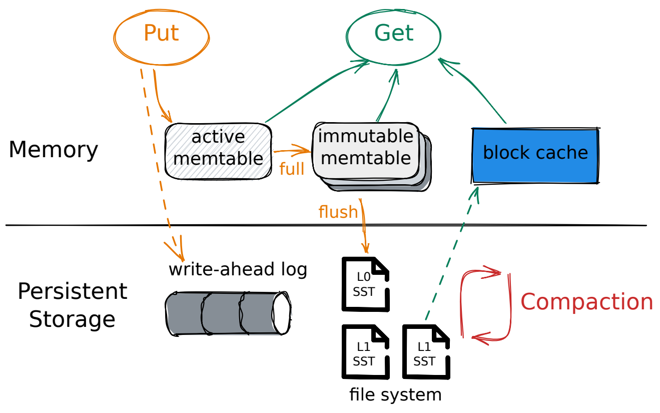


Figure 3.1: The basic architecture of RocksDB. The compaction process causes read/write amplification, consuming additional CPU, memory, and storage interconnect resources.

For the purpose of evaluation, we utilized the Yahoo! Cloud Serving Benchmark (YCSB) [54], a framework designed for assessing the performance of various distributed and cloud-based data storage systems, to generate the workload. Figure 3.2 displays the data access amplification under workload type A, with an equal ratio of read and update operations. We collected data on the YCSB observed throughput in operations per second, the RocksDB measured throughput in MB/s, and the underlying storage device throughput under the current workload in MB/s. Regarding the RocksDB throughput, we observed no significant improvement despite increasing the number of threads. However, we can note a considerable increase of over 20% in the device's raw throughput from 1 to

2 threads. Moreover, the discrepancy between the user application's observed throughput and the raw storage device's measured throughput was substantial, with the former being six times greater than the latter. The significant data access amplification not only impedes the user application's data access performance but also utilizes the host CPU, memory, and storage interconnect resources to manage the additional operations. By offloading the key-value data access functions to embedded storage devices, we can hide these additional operations and overhead within the storage devices, leaving more performance for user applications to utilize.

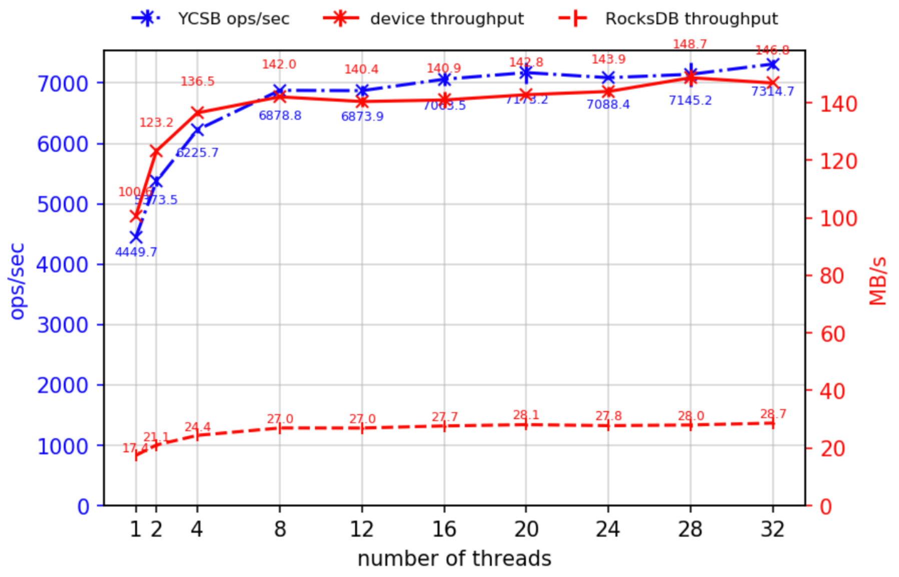


Figure 3.2: Data access amplification of the key-value function

Automate Quantification

The evaluation of MBWU-based cost-effectiveness involves a two-step process. The initial step requires measuring the MBWU of the key-value data access workload on a specific storage media. Upon the determination of MBWU, the subsequent step is to evaluate the performance capability of a system utilizing the same storage media under the identical workload, expressed in terms of the number of MBWUs. Once the cost-effectiveness values of the systems under comparison are measured, we can quantify the cost-benefit of offloading the key-value data access function from one system to another, with the latter system assumed to be an embedded storage device.

To demonstrate the applicability of our proposed quantification method across a variety of system configurations, we have designed an automated program to streamline the evaluation process. This program adheres to a client-server architecture, with the server assigned to execute the key-value function, and the client responsible for generating the workload. These components can be deployed either on a single machine or on separate machines, thereby accommodating configurations where workloads need to be delivered over the network. The YCSB is employed as the workload generator in our implementation.

To ensure the reproducibility of the MBWU measurements, the automation initiates a pre-conditioning procedure at the outset to initialize all NAND-based storage media involved in the evaluation. This procedure follows an industry-standard method laid out by the Storage Networking Industry Association (SNIA) [225]. Upon completing the pre-conditioning, the program moves forward to launch RocksDB daemons, corresponding to the number of storage devices in use. Each daemon then enters a state of readiness to accept connections from YCSB.

The RocksDB daemon is implemented using Java RMI technology [250], which exposes the interfaces of a RocksDB object such as *open()*, *close()*, *get()*, *put()*, and *delete()* over the network by binding them to an RMI registry, as depicted in Figure 3.3. Given the extensive use of ARM-based processors in embedded systems, we have successfully extended the compatibility of the RocksDB RMI program to support both x86 and aarch64 systems.

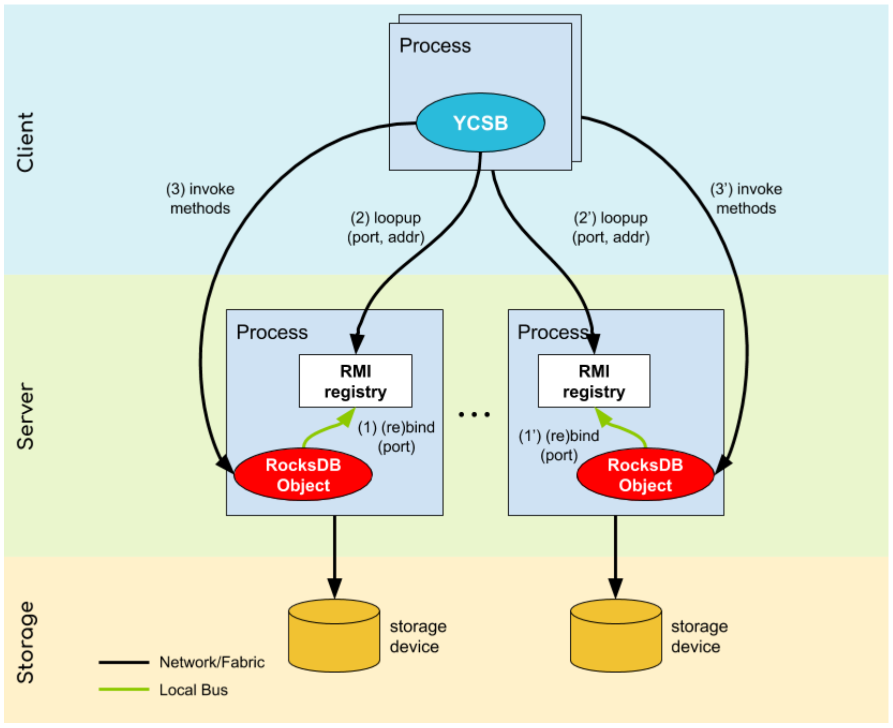
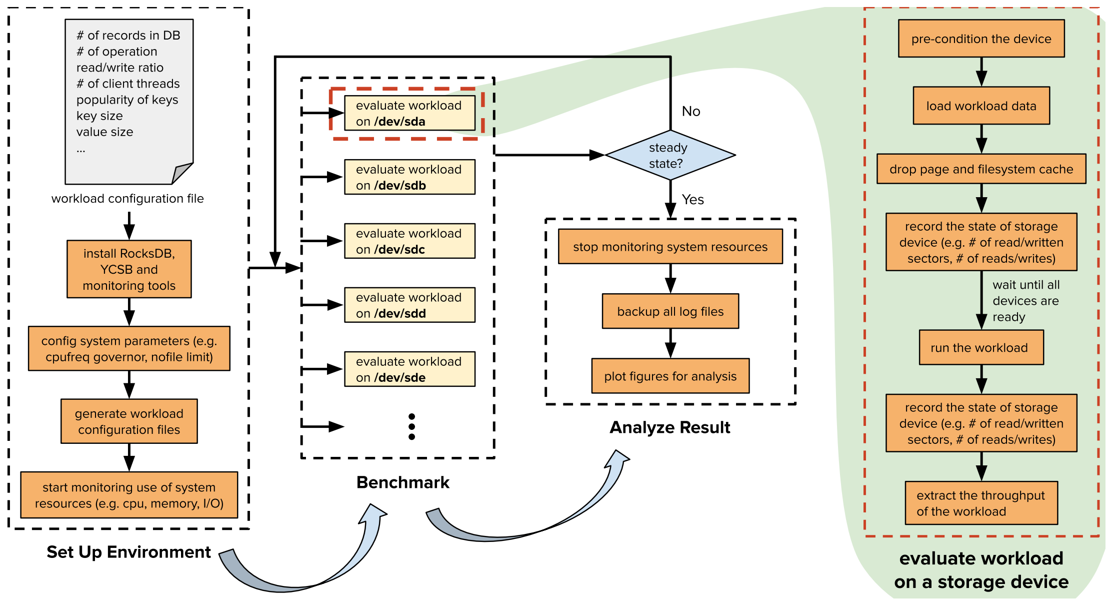


Figure 3.3: Call graph of the RocksDB RMI server

On the client side, a YCSB process is started to perform a URL lookup for the corresponding RocksDB daemon. Upon identification of the relevant daemon, the YCSB process proceeds to request the creation of a remote RocksDB instance through an *open()* remote procedure call (RPC). This RPC provides the RocksDB object with the path to a RocksDB options file that defines all the essential parameters required to properly maintain the internal LSM tree [69] and all key-value data management policies.

During the investigation, we found that keeping a consistent RocksDB options file across different systems can prevent issues caused by the "system-specific" configurations generated by RocksDB. Such system-specific configurations could lead to semantic alterations of the offloading function, potentially causing discrepancies in the function's evaluation across different systems.

Upon successfully creating a RocksDB instance, it is required to perform initial data loading before running the test workload. This process is carried out by sending key-value write requests to the RocksDB daemon through RPCs. To ensure a consistent shape of the resulting LSM trees (i.e., the same number of levels) on systems with different performance capabilities (e.g., a host and an embedded storage device), we applied jitters between load requests to ensure the system has adequate computing resources to complete data compaction. An overview of the evaluation process of our prototype is presented in Figure 3.4.



# of records in DB
# of operation
read/write ratio
# of client threads
popularity of keys
key size
value size
...
workload configuration file
install RocksDB, YCSB and monitoring tools
config system parameters (e.g. cpufreq governor, nofile limit)
generate workload configuration files
start monitoring use of system resources (e.g. cpu, memory, I/O)
Set Up Environment
evaluate workload on /dev/sda
evaluate workload on /dev/sdb
evaluate workload on /dev/sdc
evaluate workload on /dev/sdd
evaluate workload on /dev/sde
Benchmark
No
steady state?
Yes
stop monitoring system resources
backup all log files
plot figures for analysis
Analyze Result
pre-condition the device
load workload data
drop page and filesystem cache
record the state of storage device (e.g. # of read/written sectors, # of reads/writes)
wait until all devices are ready
run the workload
record the state of storage device (e.g. # of read/written sectors, # of reads/writes)
extract the throughput of the workload
evaluate workload on a storage device

Figure 3.4: A high-level view of the evaluation process

Throughout the automated evaluation process, the program continuously monitors the system's power, computing, and network resource utilization. The collected information is important in the two-step evaluation process: first, to ensure that the measured MBWU has maximized the storage media performance on the key-value workload, and second, to calculate the efficiency of the offloading function running on the system with the absolute cost measured.

3.1.3 Offloading Landscapes

In practice, the cost-benefit quantification for offloading the key-value data access function can be an arduous task due to the complexity of the evaluation process and the potential variations in the offloading landscapes. Our automation program provides a systematic approach to evaluate the cost benefits across different offloading landscapes efficiently and enables researchers to gain valuable insights into how different landscapes may impact the benefits.

Infrastructure

To create multiple offloading landscapes for testing, we used the infrastructure described in Figure 3.5 for the following experiments. The host platform served as a baseline for comparison, with RocksDB running locally over the storage devices connected directly or through a network. The host machine boasts 24 vCPUs and a total of 64 GB DRAM.

For the embedded storage platform, we used a single-board computer (SBC) called ROCKPro64 [192] and combined accessories to connect it to a Toshiba HG6 SSD. The SBC is equipped with a Rockchip RK3399 hexacore processor with 4 GB DRAM and is capable of running an Ubuntu 20.04 system, on which we run a RocksDB. The SSD is connected to the SBC via an adapter on a USB 3.0 port.

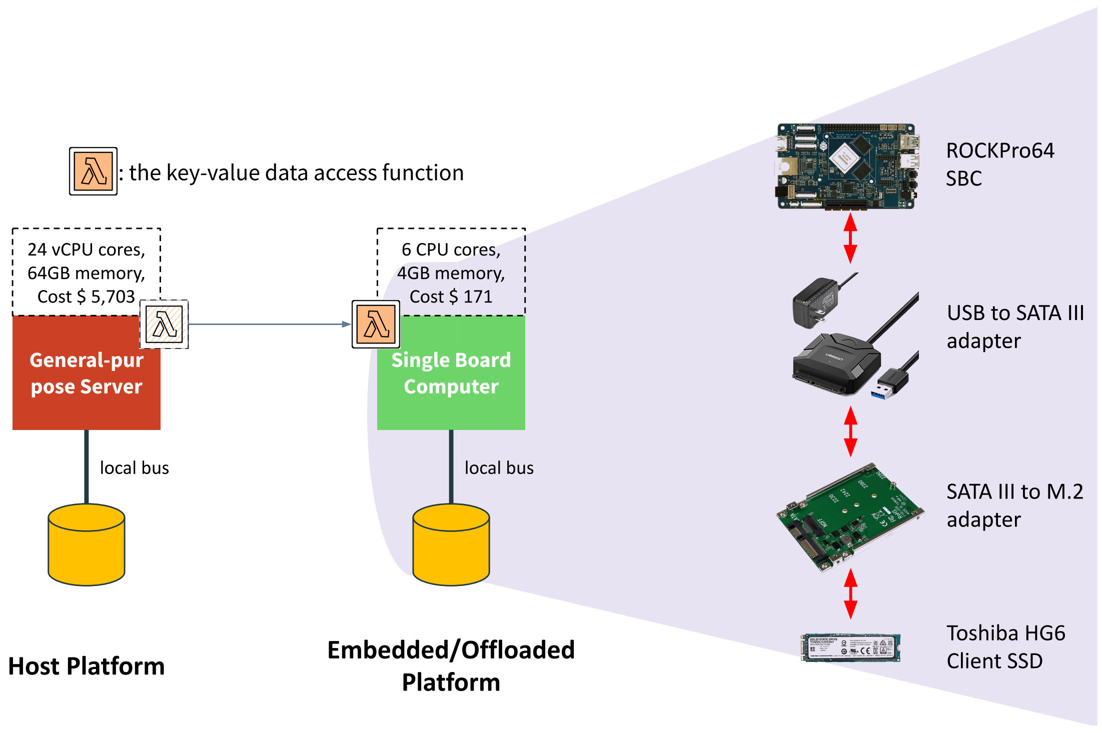


Figure 3.5: Test infrastructure for constructing offloading landscapes

Offloading Landscapes

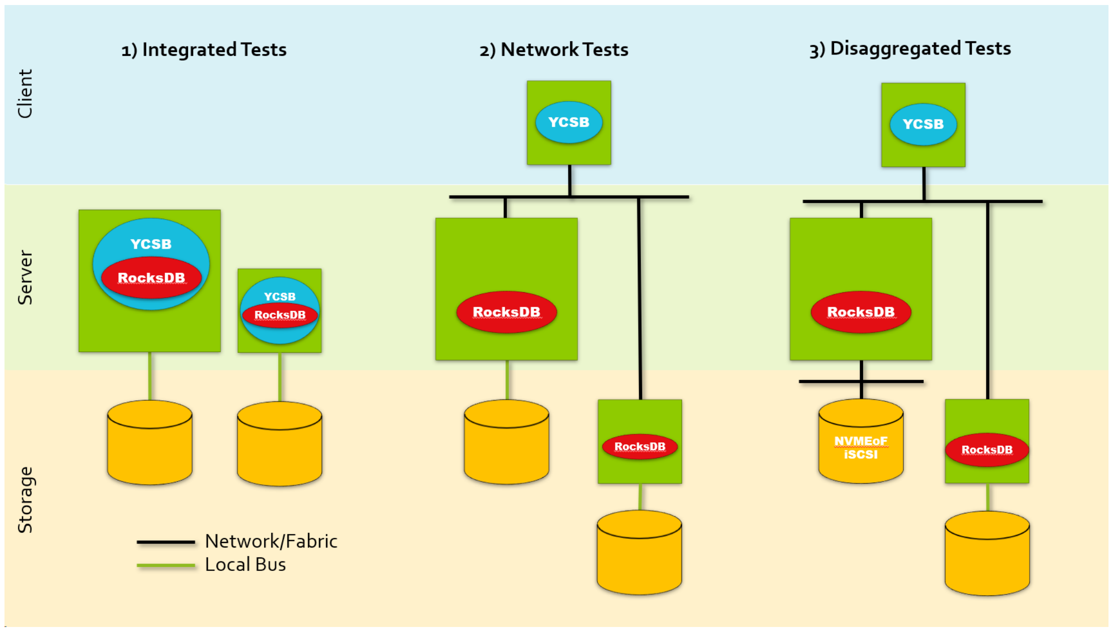


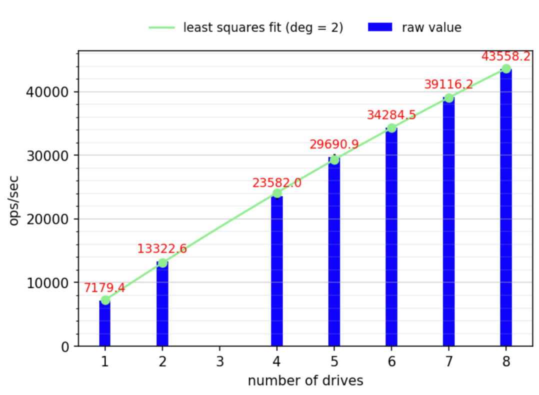
Figure 3.6: Three offloading landscapes for testing

For the purpose of evaluation, we constructed three different offloading landscapes to assess the cost-benefit of offloading the key-value data access function from a host platform to an embedded platform (refer to Figure 3.6). The MBWU for the key-value workload on the Toshiba SSD is measured to be 7314.7 ops/sec.

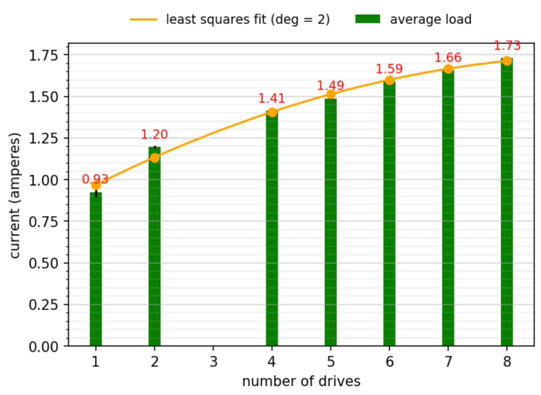
Integrated tests: The first offloading landscape involves evaluating integrated performance by running both RockDB and YCSB locally on the two platforms. The MBWUs of the host platform were measured using eight storage devices, as this system can support up to eight direct-attached storage devices, and the workload performance was observed to scale up with the increased number of devices. However, it is important to note that if the system were to support more storage devices, the host platform could potentially achieve a greater number of MBWUs. This would suggest that the system's storage resources are currently under-provisioned for the specific key-value workload or that the computing resources are over-provisioned, resulting in low cost-effectiveness for running the function to be offloaded.

Overall, the host platform achieved a total of 5.95 MBWUs, as indicated in Figure 3.7, with certain data points omitted to reduce the evaluation time. The embedded platform, on the other hand, was only connected to a single storage device and could only achieve a throughput of 0.5 MBWU (Figure 3.8).

Using the obtained MBWUs for both platforms, we can compute cost-effectiveness for each platform, employing the three MBWU-based efficiency metrics presented in Section 3.1.1. Our experimental results show that, when subjected to the key-value workload, the embedded platform reduces the \$/MBWU by 64% when compared to the host platform. Additionally, the embedded platform demonstrates a reduction of 39.6% in kWh/MBWU for energy consumption. These considerable cost and energy savings substantiate the effectiveness of offloading the key-value



(a) Aggregated Throughput



(b) Power Consumption

Figure 3.7: Integrated tests: workload performance as a function of the number of storage devices on the host platform

data access function to embedded platforms equipped with cost-effective hardware.

Network tests: The second offloading landscape differs from the first by incorporating a network component to separate the workload generation from the test platforms. This change could introduce varying degrees of additional overhead on different platforms. Although the host platform, with its abundant computing resources, may effectively handle the network overhead due to an unbalanced resource allocation, the embedded platform could face a different outcome. In the initial experiment, the embedded platform demonstrated insufficient computing resources to maximize storage media performance. Thus, the added network overhead in this scenario may further compromise its cost-efficiency in managing the offloaded key-value function.

The results of the network tests are presented in Figures 1 and 2. The host platform achieved 5.2 MBWUs, showing a 13% reduction in comparison with the integrated test results. Meanwhile, the embedded platform achieved only 0.37 MBWUs, representing a 26% decrease in performance relative to the previous test. Regarding the cost benefits of offloading the key-value function in this offloading

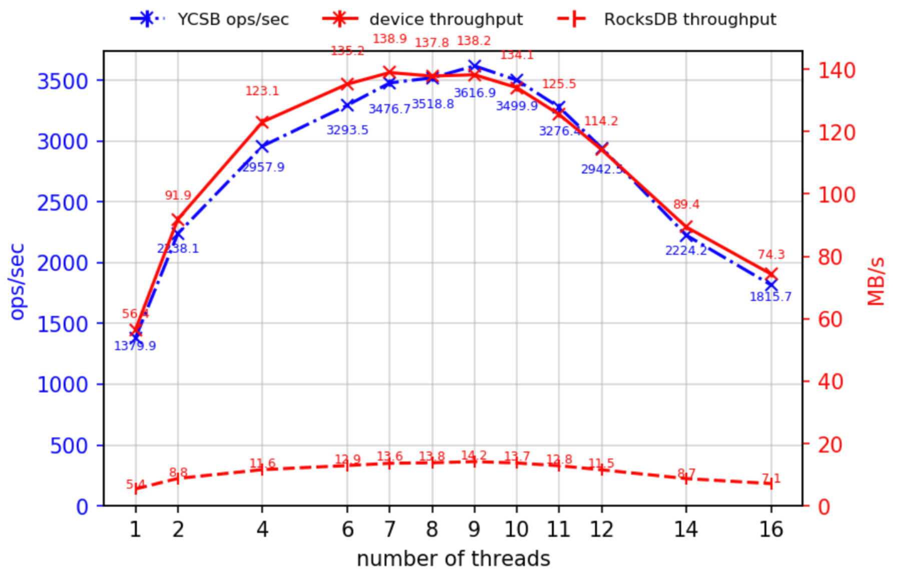
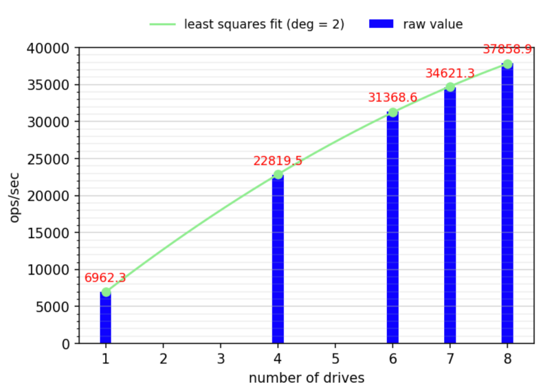


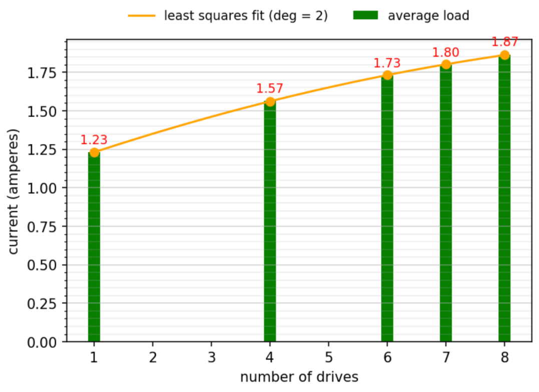
Figure 3.8: Integrated tests: workload performance as a function of the number of threads on the embedded platform

landscape, the embedded platform achieved a cost-efficiency reduction of 57.86% in $/MBWU compared to the host platform. In addition, the embedded platform showed a 45.9% reduction in kWh/MBWU, highlighting a significant decrease in energy consumption.

It is worth noting that the energy efficiency benefits of offloading to the embedded platform were further augmented in this landscape. This result can be attributed to the host platform utilizing the remaining computing resources to handle the network overhead, resulting in increased energy consumption. In contrast, the embedded platform had already allocated all system resources for the workload processing task, and therefore, adding the extra network overhead did not exert significant additional energy consumption. This observation highlights the distinct energy efficiency advantage of offloading to the embedded systems. In particular, it underscores the importance of resource composition when considering offloading data services to optimize energy efficiency in storage systems.



(a) Aggregated Throughput



(b) Power Consumption

Figure 3.9: Network tests: workload performance as a function of the number of storage devices on the host platform

Disaggregated tests: This offloading landscape separates the storage component from the host platform, simulating a common cluster deployment scheme by disaggregating the storage. Given the significant amplification of the key-value I/O at the storage, we added a 10 Gbps network adapter to the host platform to transmit the amplified data I/O through the storage network via iSCSI. However, the addition of this network adapter impacts the cost-effectiveness of the platform in three ways. First, it slightly raises the total cost of the platform by 2%. Second, it adds an average of 18 watts to the platform. Third, it occupies a PCIe slot initially assigned for storage devices, thereby reducing the number of capable storage devices on this platform to four. Finally, communicating with remote storage over the network competes for system resources needed for processing key-value workloads.

Disaggregating storage in this test exacerbates the storage resource imbalance on our host platform. However, it is worth noting that system resource imbalance is common in the clouds [99] and distributed storage systems [165], since building scalable resource management to optimize resource utilization is a challenging task. By quantifying the cost benefits from the disaggregated offloading landscape,

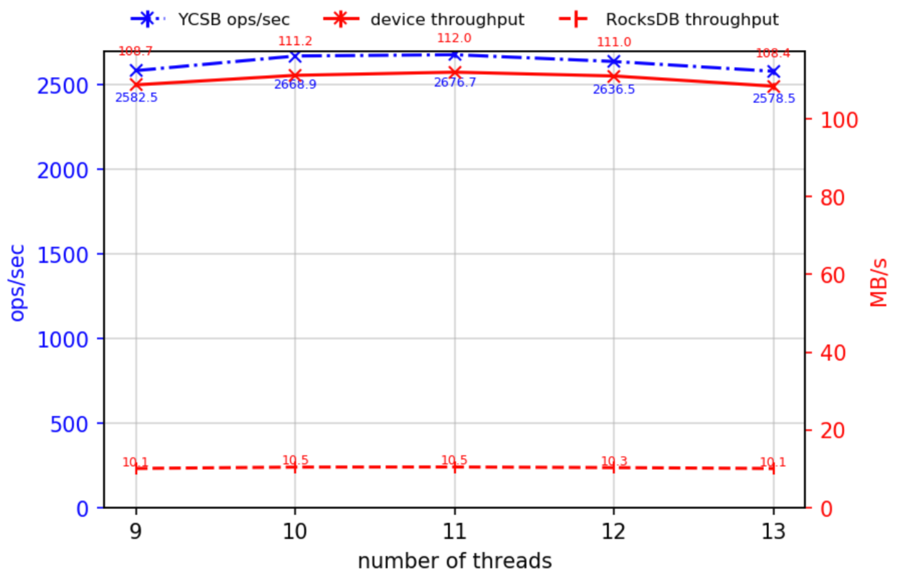
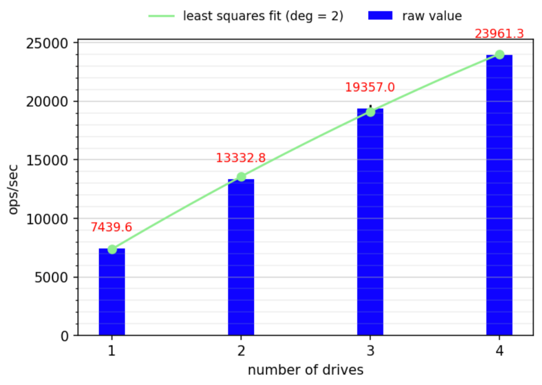


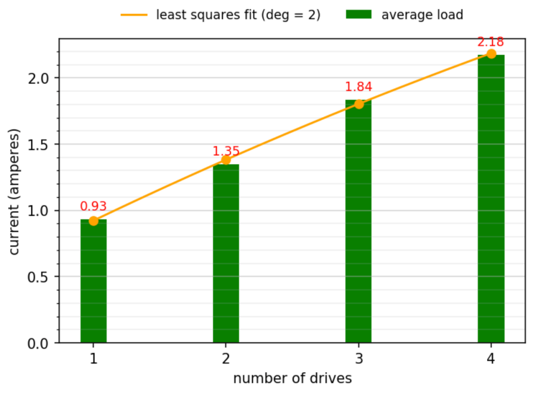
Figure 3.10: Network tests: workload performance as a function of the number of threads on the embedded platform

we can gain insights into the potential savings that can be achieved by offloading the key-value function from a disaggregated storage system.

The performance results of the host platform under the disaggregated configuration are presented in Figure 3.11. The measured number of MBWUs for the host is 3.28, while the number of MBWUs for the embedded platform remains unchanged as the configuration was the same as in the network tests. Combining these numbers, we observed that the embedded platform achieved substantial cost savings — 73.4% in \$/MBWU and 70.7% in kWh/MBWU. These outcomes align with our expectations based on the network test results. Specifically, the decreased system resource utilization on the host platform, resulting from fewer storage devices employed in this configuration, led to a lower number of MBWUs. Moreover, the added storage network overhead caused the platform to consume more energy.



(a) Aggregated Throughput



(b) Power Consumption

Figure 3.11: Disaggregated tests: workload performance as a function of the number of storage devices on the host platform

Discussion

MBWU is a metric based on the storage media performance under a specific data access workload and is primarily considered a throughput-based metric. However, it is important to note that the metric is not exclusively applicable to throughput-oriented workloads. Data access functions sensitive to latency can also harness this metric to evaluate the cost-effectiveness of potential offloading. To transition the throughput metric to a latency metric, we can leverage the widely observed correlation between throughput and average latency in queuing systems [195, 22, 85]. Specifically, reducing the throughput can lower the average request latency. Consequently, instead of measuring the maximum throughput under full storage media performance, we can measure an MBWU that complies with a specific latency requirement and use it to normalize the workload performance on a system.

MBWU and MBWU-based efficiency metrics offer a means of evaluating the efficiency with which a data access workload utilizes storage media performance. These metrics fill an existing gap and present a new toolkit to system architects striving to evaluate and optimize the resource composition of embedded storage systems for better cost-effectiveness.

3.1.4 Summary

Existing research on function offloading has been primarily focused on evaluating the benefits of offloading from an application user's perspective. In this section, we propose a set of metrics designed to help system architects evaluate and quantify the cost-effectiveness and benefits of offloading a given data access function to embedded devices with specific storage media. To facilitate the evaluation process, we have developed an automation program capable of quantifying the offloading of a key-value data access function in various offloading landscapes, thereby enabling more convenient and insightful analysis.

3.2 Data Availability

The benefits of offloading data services to embedded storage are manifold and demand the use of performance- and non-performance-oriented metrics for a comprehensive evaluation. A crucial factor that system architects must consider is the impact on data availability when offloading data services since this process may require data to be fragmented into multiple parts and processed by different embedded systems. This section will focus specifically on quantifying the benefits of data availability in the context of edge storage, as it represents a general problem for small storage systems that have limited failure domains.

Edge storage has emerged as a key driver for expanding the global datasphere [198] with a forecast suggesting that 75% of data will be generated and processed outside the cloud [237]. With the trend of increasing storage at the edge to handle the growing demand for efficient data services, maintaining geographically distributed edge sites presents a significant challenge. Failures in edge infrastructures require maintenance personnel to travel to remote sites to rectify the issue, with the cost of these “truck rolls” estimated to exceed one thousand dollars per event [259]. Unlike central data centers, edge data centers often face environmental constraints such as limited space and power, network instability, and temperature [137, 18]. These factors make the cost of provisioning and operating redundant resources at the edge comparable to the cost of truck rolls.

3.2.1 Cost-effective Failure Domains

Embedded storage nodes can encapsulate computing resources and storage media in a compact form factor, making them a potentially attractive option for building edge storage systems. In this section, we will elaborate on the reasoning behind the benefits of utilizing embedded storage nodes for this purpose as follows:

Failure domains: Just as diversifying one’s investments can mitigate risk, a failover mechanism that spans multiple independent failure domains can enhance the availability of data stored in a system. A storage node represents a failure domain because the failure of critical components, such as the CPU and DRAM, can result in the inaccessibility of all the data hosted by that node. Thus, a reliable failover mechanism should store redundant data on independent storage nodes. For example, a failover mechanism using data replication should store replicas of a data item on storage devices of different servers. Furthermore, the complexity of a failure domain impacts its reliability, as a less complex failure domain (i.e., with fewer disks attached to a node) is typically more reliable.

However, edge data centers often face environmental restrictions that can limit the number of failure domains in a storage system with monolithic storage nodes like general-purpose storage servers. Utilizing embedded storage nodes presents a promising solution with several key advantages. First, their compact form factor allows the deployment of more nodes within a given spatial constraint. Second, their simpler system design makes them more affordable, enabling the deployment of a greater number of nodes within a specified budget constraint. Finally, constructing storage systems using embedded storage nodes embraces a scale-out approach, ensuring that fewer storage devices coexist within a single failure domain. It has been demonstrated in distributed database systems that scaling out can effectively improve data availability [68]. This approach can be similarly applied to edge storage systems, employing embedded storage nodes to provide more optimally sized failure domains.

Cost efficiency: Embedded storage nodes exhibit superior power and space efficiency compared to general-purpose servers. The breakdown of Dennard scaling [27], which started around 2005, signifies that improving the computing performance of processors demands an increase in the power supply to the circuits. The subsequent rise in power consumption generates more heat, which in turn necessitates more space for effective heat dissipation. Embedded storage nodes are designed with moderate computing power and a relatively simple system design, contributing to their improved power and space efficiency. According to our evaluation using the MBWU-based efficiency metrics, offloading the key-value data access function from a host platform to an embedded platform can yield a 45.9% increase in power efficiency and a 79.7% increase in space efficiency.

As a means of evaluating the benefits of data availability offered by embedded storage nodes, we have constructed a mathematical model to compare a storage system built with general-purpose servers to one that is built with embedded storage nodes, as depicted in Figure 3.12.

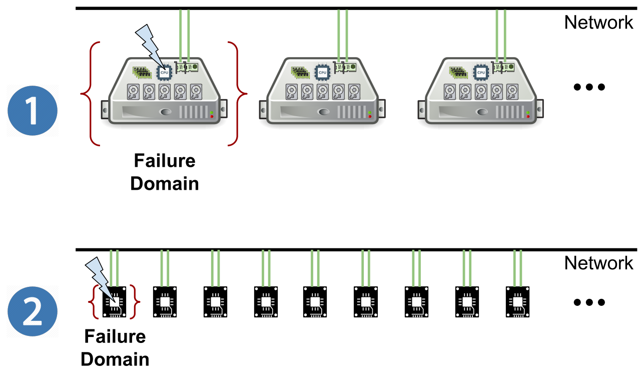


Figure 3.12: Storage systems with different building blocks: the first system uses general-purpose servers while the second system uses embedded storage nodes

3.2.2 Model Assumptions

Given the significant architectural differences between these two building blocks, directly comparing the two types of systems without making certain assumptions is challenging. Therefore, we carefully established model parameters based on system configuration assumptions while ensuring that the generality of the results was not impacted. Our assumptions regarding the configurations of the two systems for comparison and the reasoning are outlined as follows:

Storage Node: We assume all nodes in the storage system built with general-purpose servers share the same configuration. This includes an identical number and type of CPU cores, the same amount and type of DRAM, and the same number and model of block storage devices. Similarly, we assume that all embedded devices are of the same model for storage systems constructed with embedded storage nodes. These assumptions allow us to maintain a consistent failure rate for components of the same type within the same type of building blocks. For example, all CPUs within the servers would have the same failure rate.

External Redundancy: We assume that both the network and power have the same level of redundancy for all nodes. This allows us to omit external dependencies and focus solely on the data availability provided by the system building blocks.

Data Redundancy: We assume both systems use 3-way replication for data redundancy. While there are other data redundancy techniques, such as erasure coding, we focused on data replication in this study and left the analysis of other redundancy techniques for future work. We could increase the replication factor from 3 to 4 or even higher. However, as studied in [49], increasing the factor to 4 does not significantly improve the probability of data loss for the scale of nodes typically deployed at the edge. Additionally, given the space restrictions of the edge, employing a higher replication level would require more nodes within the limited space available for edge deployment.

Replication Scheme: We assume that both systems utilize the copyset scheme [49] for data replication, which offers several advantages over the random replication scheme used in some production storage systems. With random replication, where replicas of a data chunk are stored on k nodes, the replicas can be distributed across any combination of k nodes when the system has a sufficient number of chunks. This can lead to the formation of joint failure domains, where any combination of k nodes becomes a potential point of failure, posing a significant risk of data loss. In contrast, the copyset replication scheme limits the number of joint failure domains (known as "copysets") that share replicas, reducing the probability of data loss when any combination of k nodes fails. This makes the copyset scheme a more robust and reliable solution for data replication in systems with large amounts of data and a small replication factor.

Failure Correlation: We also assume that the failures of different servers and storage devices are independent. This allows us to model the probabilities of hardware failures using the Poisson distribution [254]. The Poisson distribution is commonly used to model the occurrence of rare events, such as hardware failures in a storage system. Additionally, we assume that a general-purpose server can host multiple block storage devices while an embedded storage node is limited to a single storage device.

We list the symbols used in our mathematical model in Table 3.1 with the following parameter assumptions:

- $R_m = R_m^{'}$ and $R_d = R_d^{'}$. We make the assumption that the failure rates of a general-purpose server and an embedded storage node, excluding storage components, are equivalent over a specific period of time. Despite the greater complexity of the system design of a general-purpose server in comparison to that of an embedded storage node, which may suggest a higher possibility of failure, we employ this assumption to derive a conservative outcome in our comparison. Even in cases where the failure rate of the general-purpose server, denoted as R_m, is five times greater than that of the embedded storage node, denoted as $R_m^{'}$, our model shows that, when $n = 4$, the possibility of data loss in an embedded storage node-based system remains lower than in a general-purpose server-based system. We also assume equivalent failure rates for storage devices employed in both system building blocks.

- $R_d = f \cdot R_m$, where $f > 0$. This ratio defines the relationship between the failure rate of a storage device and that of the computing resources in a storage server. For spinning hard drives (HDDs), the value of f may exceed 2, whereas, for solid-state drives (SSDs), f may be less than 1. We simply call f **the ratio of failure rates**.

- $m^{'} = c \cdot m$, where $c >= 1$. An embedded storage node may have less process-

Table 3.1: List of Model Parameters

Name	**Description**
m	the number of servers in the storage system
$m^{'}$	the number of embedded storage nodes in the storage system
n	the number of storage devices in a server
R_m	the failure rate of a server excluding the storage components
R_d	the failure rate of a block storage device in a server
$R_m^{'}$	the failure rate of an embedded storage node excluding the storage component
$R_d^{'}$	the failure rate of the storage device in an embedded storage node
w	the scatter width of the copyset replication

* For the purpose of aiding retention, we utilize "m" to signify "machine" and "d" to indicate "device" in the following notations: R_m, R_d, $R_m^{'}$, $R_d^{'}$.

ing power than a general-purpose server. In such cases, it may be necessary to use multiple embedded storage nodes to achieve a level of performance comparable to that of a server. We call c **the ratio of computing performance**.

- $n >= 2$. We assume that each server will host multiple block storage devices, with a minimum requirement of two storage devices per server. We call n **the ratio of storage performance**.
- $m >= 3$. As we use 3-way replication for data redundancy, it is necessary to have a minimum of three failure domains, each corresponding to a server, to provide fault tolerance.

By defining these ratios, we can evaluate the effects of changes in the relative probabilities of failure between the two storage systems, and consequently evaluate the impact of these changes on the probability of data loss.

3.2.3 Mathematical Model

The scatter width parameter specifies the number of nodes to which the data on a given node can be replicated. Based on the assumptions regarding the configurations of the two types of storage systems, the total number of copysets in the general-purpose server-based storage system and the embedded storage node-based storage system are given by $l_{gp} = \frac{wm}{6}$ and $l_{es} = \frac{wm'}{6}$, respectively (see Section 3.2 in [49]). For simplicity, we assume that data will be replicated to storage devices with the same index. For instance, if $1, 4, 7$ is a copyset, the data on disk 1 of node 1 will be replicated to disk 1 of node 4 and disk 1 of node 7. Note that a different device mapping for replication could be employed, but it should not impact the results obtained from our model.

For storage systems constructed with general-purpose servers, the occurrence of data loss can stem from one of three situations. First, multiple server failures may lead to data loss if at least three of these servers fall within the same copyset. Second, data loss can result from multiple storage device failures if at least three of these devices are hosted by servers in the same copyset, and all three devices share the same device index. Finally, a combination of three or more failures can result in data loss, whereby a particular combination of simultaneous failures leads to the loss of data.

Since server failures are independent, we can express the probability of failures involving exactly k servers by applying the probability mass function of the Poisson distribution:

$$P(\text{failures of } k \text{ servers}) = \frac{{R_m}^k e^{-R_m}}{k!} \tag{3.1}$$

Similarly, the probability of failures involving exactly j storage devices is:

$$P(\text{failures of } j \text{ storage devices}) = \frac{{R_d}^j e^{-R_d}}{j!} \tag{3.2}$$

We can then express the possibilities of the aforementioned situations that cause data loss as follows:

(i) $$P_m(k) = P(\text{failures of } k \text{ servers}) \times \frac{N_m(k)}{\binom{m}{k}} \tag{3.3}$$

(ii) $$P_d(j) = P(\text{failures of } j \text{ storage devices}) \times \frac{N_d(j)}{\binom{mn}{j}} \tag{3.4}$$

(iii) $$\begin{aligned} P_{m,d}(k,j) = {} & P(\text{failures of } k \text{ servers}) \\ & \times P(\text{failures of } j \text{ storage devices}) \\ & \times \frac{N_{m,d}(k,j)}{\binom{m}{k} \times \binom{mn}{j}} \end{aligned} \tag{3.5}$$

Equation 3.3 defines $N_m(k)$ as the number of k-combinations of servers, requiring in any combination at least three servers to fall within the same copyset. Equation 3.4 defines $N_d(j)$ as the number of j-combinations of block storage devices, requiring each combination to contain at least three devices sharing the same device index whose hosts are in the same copyset. Lastly, equation 3.5 defines $N_{m,d}(k,j)$ as the number of combinations that contain failures of k servers and j storage devices, which requires at least three failures in each combination to be associated with a copyset. Specifically, for each combination, there must be a copyset that contains either one failed server with the other two servers in the copyset hosting two failed storage devices that share the same device index, or two failed servers with the remaining server in the copyset hosting the failed storage device.

By adding up the possibilities of these cases, we can get the possibility of data loss for the storage system constructed with general-purpose servers:

$$\begin{aligned} P_{gp} = & \sum_{k=3}^{m} P_m(k) + \sum_{j=3}^{mn} P_d(j) \\ & + \sum_{k=2}^{m} \sum_{j=1}^{mn} P_{m,d}(k,j) + \sum_{j=2}^{mn} P_{m,d}(1,j) \end{aligned} \tag{3.6}$$

Similarly, the possibility of data loss for the storage system constructed with embedded storage nodes is:

$$\begin{aligned} P_{es} = & \sum_{k=3}^{m'} P_m^{'}(k) + \sum_{j=3}^{m'} P_d^{'}(j) \\ & + \sum_{k=2}^{m'} \sum_{j=1}^{m'} P_{m,d}^{'}(k,j) + \sum_{j=2}^{m'} P_{m,d}^{'}(1,j) \end{aligned} \tag{3.7}$$

where

$$P_m^{'}(k) = \frac{{R_m^{'}}^k e^{-R_m^{'}}}{k!} \times \frac{N_m^{'}(k)}{\binom{m'}{k}} \tag{3.8}$$

$$P_d^{'}(j) = \frac{{R_d^{'}}^j e^{-R_d^{'}}}{j!} \times \frac{N_d^{'}(j)}{\binom{m'}{j}} \tag{3.9}$$

$$P_{m,d}^{'}(k,j) = \frac{{R_m^{'}}^k e^{-R_m^{'}}}{k!} \times \frac{{R_d^{'}}^j e^{-R_d^{'}}}{j!} \times \frac{N_{m,d}^{'}(k,j)}{\binom{m'}{k} \times \binom{m'}{j}} \tag{3.10}$$

Finally, to compare the possibility of data loss between the two types of storage systems, we can evaluate the ratio between P_{gp} and P_{es}:

$$\text{Relative Benefit} = \frac{P_{gp}}{P_{es}} \tag{3.11}$$

3.2.4 Data Availability Evaluation

The proposed model utilizes expressions such as $N_m(k)$ and $N_d(j)$ to represent the number of combinations that could potentially lead to data loss. However, the effectiveness of these expressions is subject to the values of m and w, and it may not always be possible to identify an optimal scheme that yields non-overlapping copysets covering all servers in the storage system. Therefore, it becomes necessary to compare the two systems based on predetermined, fixed values of k and j. As an illustration, consider the scenario where $k + j \leq 3$, represents cases where exactly three components failed. The relative probability of data loss between the two systems is:

$$\text{Relative Benefit} = \frac{P_m(3) + P_d(3) + P_{m,d}(1,2) + P_{m,d}(2,1)}{P'_m(3) + P'_d(3) + P'_{m,d}(1,2) + P'_{m,d}(2,1)} \tag{3.12}$$

In this scenario, we have $N_m(3) = l_{gp}$, $N_d(3) = nl_{gp}$, $N_{m,d}(1,2) = N_{m,d}(2,1) = 3nl_{gp}$ and $N'_m(3) = N'_d(3) = l_{es}$, $N'_{m,d}(1,2) = N'_{m,d}(2,1) = 3l_{es}$.

Referring to equation 3.12, we can generate plots by fixing f and w to reasonable values and examine the correlation between the relative benefit, the number of servers m, the ratio of computing performance c, and the ratio of storage performance n. To this end, we present a series of figures from 3.13 to 3.16. We set $w = 4$ since it offers comparable performance in terms of recovery time to random replication on small clusters [49]. To evaluate the impact of m and n, we conservatively set $c = n$, indicating that the total number of block storage devices in the server-based system equals the number of nodes in the embedded storage node-based storage system. Additionally, to illustrate the impact of m and c, we set $n = 12$ to emulate a moderate-sized edge server.

Figure 3.13 presents an evaluation of the impact of variations in the number of servers and the ratio of storage performance on the relative benefit. We set $f = 2$ as the ratio of failure rates for HDDs, according to [240]. The figure demonstrates that despite the total number of storage devices in the server-based system being equal to the total number of nodes in the embedded storage node-based system, the latter has a lower probability of data loss. This is primarily due to the embedded storage node-based system having more independent failure domains over which the copyset replication scheme can span. For instance, when each server hosts four storage devices, the relative benefit can be as high as 7.1. In Figure 3.14, when $c = 12$, the total number of storage devices in the server-based system equals the number of embedded storage nodes. However, we observe a significant relative benefit even when $c < 12$. This figure supports our hypothesis that less complex failure domains tend to be more reliable.

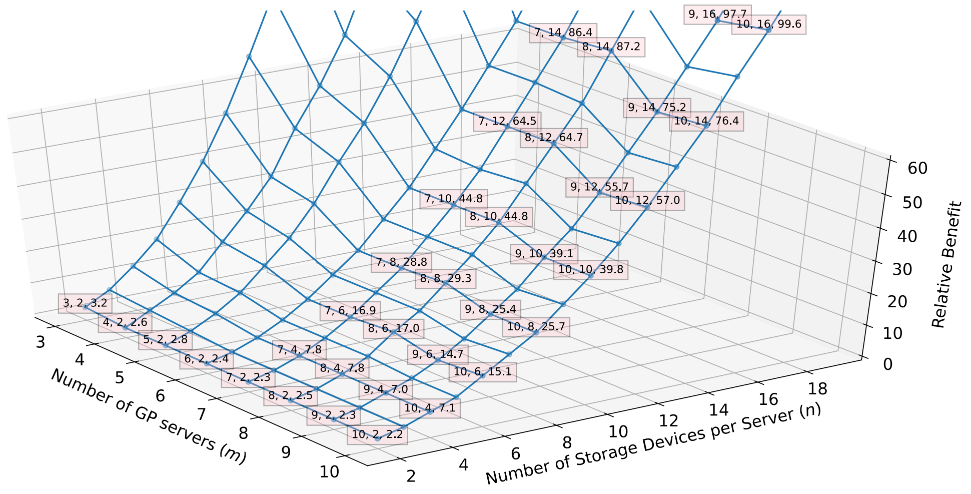


Figure 3.13: The impact of m and n on the relative benefit with HDDs

Figures 3.15 and 3.16 use $f = 0.06$ to emulate the low failure rate of SSDs, based on a previous study [261] that indicates only 5.6% of all hardware failure events are due to SSDs. Under this configuration, the primary risk of hardware failures stems from computing resources such as CPU and DRAM, resulting in steeper growth in the curves shown in these figures compared to those with HDDs.

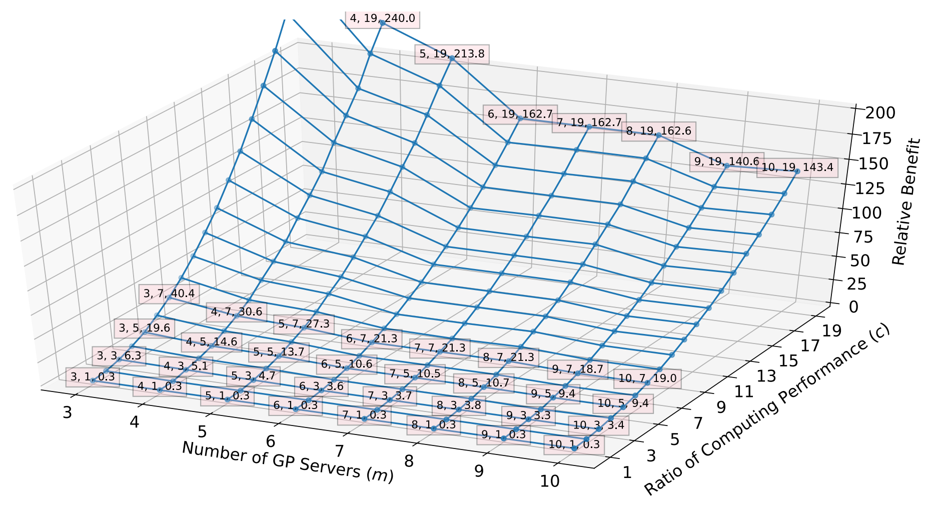


Figure 3.14: The impact of m and c on the relative benefit with HDDs

For example, in the scenario where a server hosts four SSDs, the relative benefit reaches 20.7. The result suggests that storage systems employing SSDs may be better suited to a scale-out architecture that employs less complex building blocks, such as embedded storage nodes, to achieve more cost-effective data availability.

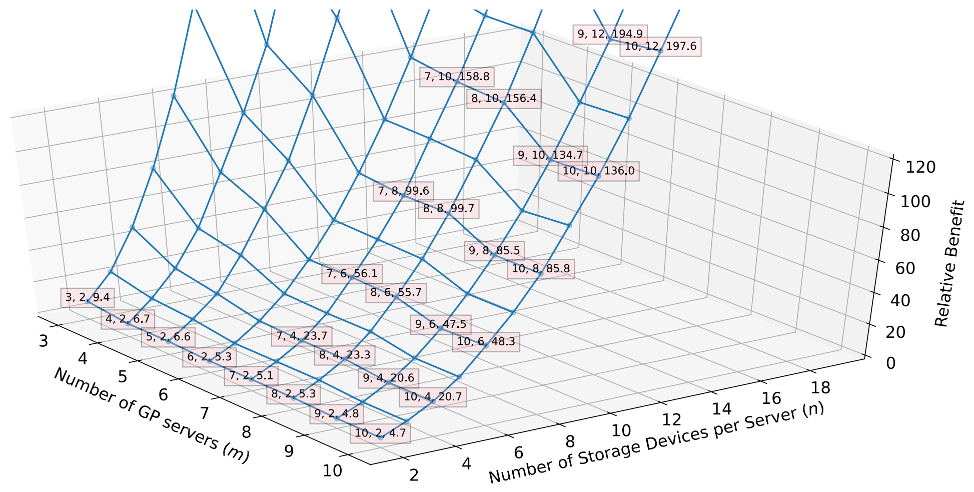


Figure 3.15: The impact of m and n on the relative benefit with SSDs

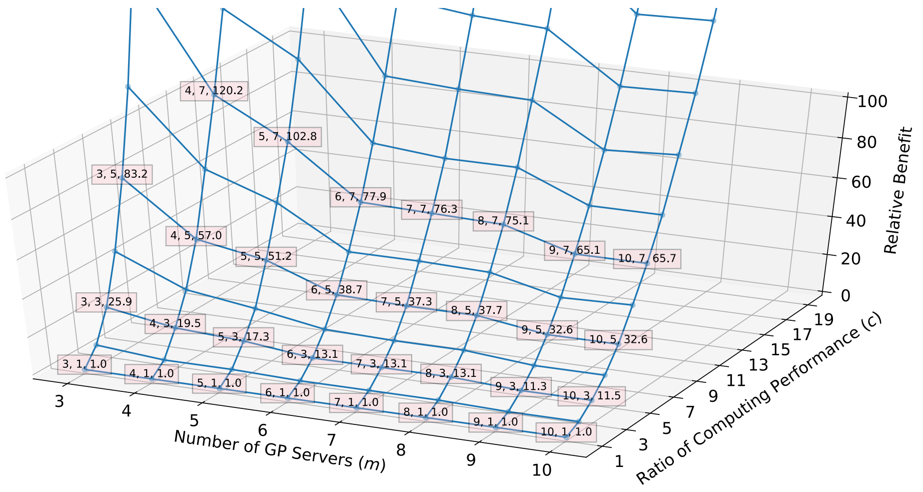


Figure 3.16: The impact of m and c on the relative benefit with SSDs

3.2.5 Observations and Insights

Based on the evaluation above, we present our key observations that may provide valuable guidance to storage system architects seeking to optimize the data availability of their systems while minimizing costs.

Insight 1: The lower the failure rate of storage devices used in servers, the higher the *Relative Benefit* of embedded storage.

Figure 3.17 depicts a frame obtained from the figures 3.13 and 3.15 at $m = 10$. This figure compares a server-based system with ten servers (10 failure domains), each hosting n storage devices, to an embedded storage node-based system with $10n$ nodes ($10n$ failure domains). Given that SSDs have a lower failure rate than HDDs, the increasing gap in relative benefits between these two types of devices indicates that aggregating storage devices with a lower failure rate in a server can result in a higher relative risk of data loss. In addition, this figure presents that:

Insight 2: The more storage devices the server aggregates (higher storage aggregation), the higher the *Relative Benefit* of embedded storage.

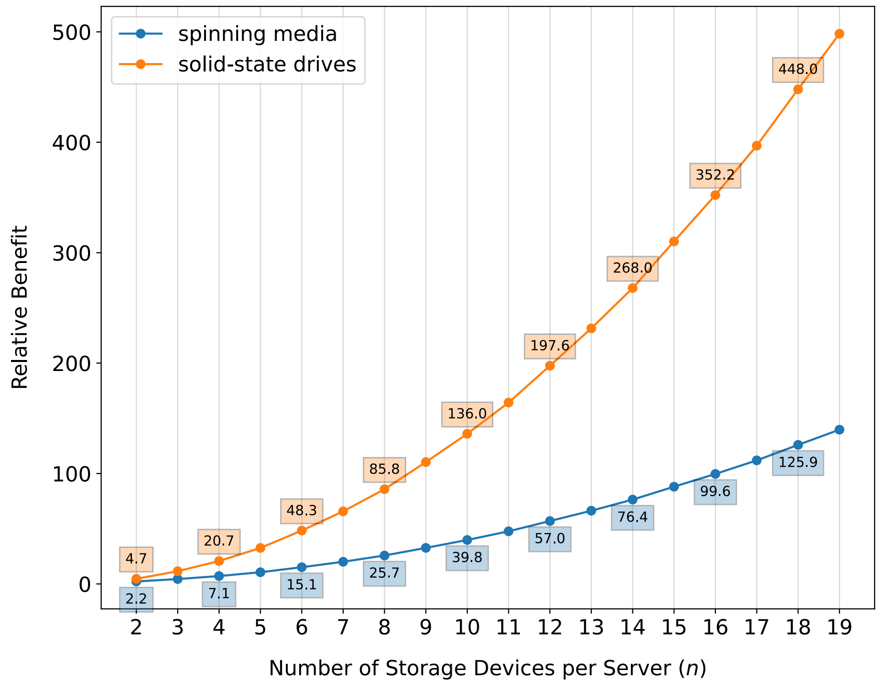


Figure 3.17: The impact of n on the relative benefit ($m = 10$)

Insight 3: The more computing resources the server aggregates (higher compute aggregation), the higher the *Relative Benefit* of embedded storage.

Figure 3.18 depicts a frame obtained from the figures 3.14 and 3.16 at $m = 10$. This figure compares a server-based system with ten servers (10 failure domains), each hosting 12 storage devices, to an embedded storage node-based system with $10c$ nodes ($10c$ failure domains). The trends observed in the two curves depicted in the figure reinforce our hypothesis that the greater the number of independent failure domains the failure mechanism can encompass, the lower the risk of data loss. Furthermore, our results indicate that to augment the computing capacity of a storage system, scaling out with embedded storage nodes may offer an additional benefit in terms of data availability, as opposed to scaling up each storage server.

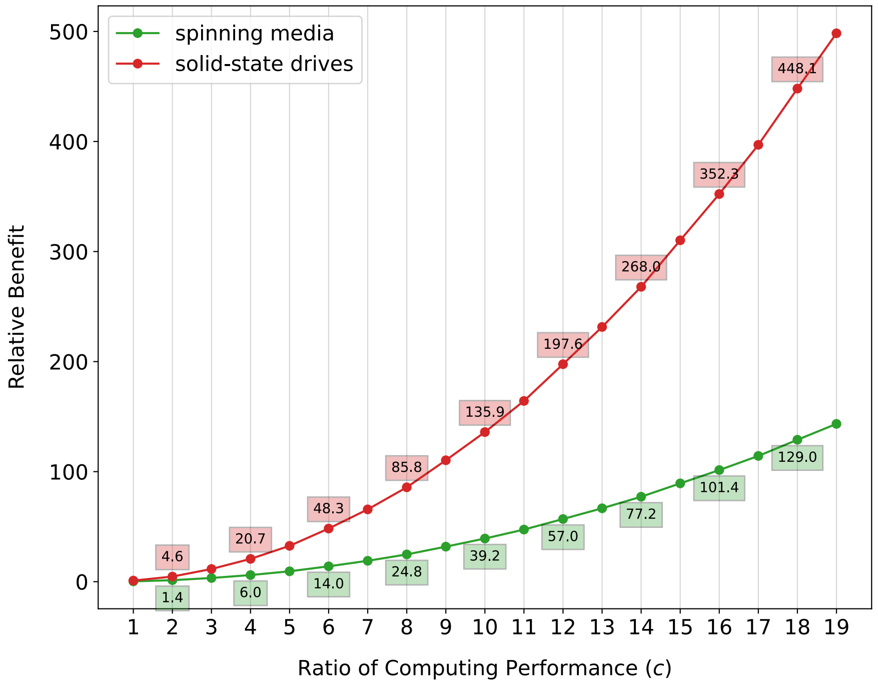


Figure 3.18: The impact of c on the relative benefit $(m = 10)$

Insight 4: Scaling out the resources of a storage system offers a non-linear improvement in the *Relative Benefit*.

That is, storage systems with higher resource aggregation can benefit more from scaling out the system.

Discussion

About the model: **i)** It is better to be able to calculate the possibility of data loss without fixing the value of k and j. To achieve this, a general formula is required to compute the values of $N_m(k)$, $N'_m(k)$, $N_d(j)$, and $N'_d(j)$ for any values of m and w, even in cases where optimal schemes are not present. However, solutions to these expressions remain elusive. In future work, we may employ stochastic simulations to estimate the probability of generating a subset of size k to cover any predefined equal-sized subsets to solve this problem. **ii)** It is pertinent to acknowledge that the assumption of independent failures of different servers or nodes may not always hold. The need for data redistribution, when failures occur, can lead to excessive traffic, potentially reducing the lifetime of involved storage devices. Additionally, the repair rate may be incorporated in our study in the future to facilitate the simulation of uncorrelated failures using the Markov model [96, 33].

Performance of embedded storage nodes: Embedded storage nodes are generally less powerful than general-purpose servers, primarily attributed to their compact form factor, which limits the space available for packaging computing resources. However, scale-out storage systems augmented with embedded storage nodes can offer higher aggregate bandwidth, making them especially attractive to bandwidth-sensitive functions such as in content delivery services [236]. For latency-sensitive workloads, domain-specific accelerators can be utilized on embedded storage nodes to optimize applications by tailoring functions that benefit most from running within these nodes. Our proposal for using embedded storage nodes does not aim to replace general-purpose servers in storage systems; instead, it serves as a complement to enable more cost-effective execution of profitable functions.

External complexity: The adoption of embedded storage nodes in storage systems may necessitate additional network connection ports for communication, which may raise concerns regarding data loss due to the increased network complexity. However, it is important to note that traditional storage devices also require connections with ports inside servers, such as SAS/SATA ports. Consequently, the storage device connection complexity remains relatively unchanged when using embedded storage nodes. Furthermore, no conclusive evidence indicates a significant difference in failure rates between storage device connectors and network ports.

System design: Resizing the failure domains of storage systems requires a careful balance between hardware cost and performance. A reduction in the size of a failure domain may lead to an increased cost per gigabyte, as each server will host a smaller number of storage devices, contrary to the conventional notion that the cost of computing resources can be amortized by increasing the number of storage devices within a server. However, servers with finer-grained resources may prove to be more cost-effective than aggregating resources to increase their power. This model can play an important role in system design, aiding system architects in determining the optimal balance between the size of a failure domain, hardware cost, and performance.

3.2.6 Summary

This study provides additional insights into the benefits of employing embedded storage nodes to improve data availability and potentially reduce operational costs for small storage systems. These systems are typically characterized by a restricted number of failure domains, a scenario frequently encountered in edge infrastructures. Embedded storage nodes present more optimally sized failure domains, thereby facilitating the deployment of a greater number of nodes to attain these improvements. Our evaluation demonstrates that embedded storage node-based systems offer a significantly lower data loss risk than general-purpose server-based systems (7 to 20 times lower). These findings emphasize the potential of systems built on embedded storage nodes to offer a cost-effective and reliable solution for sites operating with small storage systems.

3.3 Conclusion

This chapter presents several metrics that can assist system architects in achieving a cost-optimized balance when designing embedded systems. The MBWU and MBWU-based efficiency metrics quantify the performance utilization of storage media and associated costs of achieving the performance under a given data access workload. Additionally, we present a mathematical model to analyze the trade-offs between multiple resource ratios in a system that can impact system performance and the cost of realizing the performance in terms of data availability. These metrics and evaluating methodologies can serve as valuable tools in embedded system design, providing a comprehensive understanding of the benefits of utilizing embedded storage hardware for data service functions and workloads.

Chapter 4

Offloading Potential

Embedded devices have the potential to significantly enhance system performance by leveraging their unique capabilities. For instance, computational storage devices can utilize high-bandwidth and low-power internal data links to process data in place, while programmable network interface cards, or SmartNICs, can improve system performance by taking over networking and data processing tasks from the host CPU. However, given the diverse requirements of different applications, it is essential to have a comprehensive and quantified understanding of the performance of these embedded devices, particularly under application-relevant workloads. This understanding can help avoid discrepancies in performance expectations and facilitate the creation of performance envelopes for customizing functions of applications that can benefit from offloading.

In the high-performance computing (HPC) and high-performance data analytics (HPDA) communities, there has long been a vision of improving application workflows through the use of programmable processing elements embedded in the network fabric [133]. Vendors' recent introduction of SmartNICs has piqued their interest in exploring potential roles these devices may play in the data paths of simulation and data processing applications. To address this gap, this chapter presents an in-depth performance characterization of a representative SmartNIC, specifically the NVIDIA Bluefield-2 SmartNIC. This examination spans four different dimensions, each with its unique scope ranging from general to application-specific aspects. Section 4.2 discusses the performance of general micro-operations on the SmartNIC by using techniques similar to MBWU for normalization, except that in this case the performance of a Raspberry Pi serves as the reference. Section 4.3 explores the network processing capacity of an offloaded data processing function. In Section 4.4, the focus shifts to the performance of data partitioning, a critical function that enables SmartNICs to distribute and manage in-transit data. Finally, Section 4.5 examines parallel data processing, evaluating the device's performance when processing data streams with multiple threads.

4.1 Prototyping Platform

In recent years, multiple hardware vendors have introduced network devices that feature user-programmable CPUs or FPGAs. These resources enable developers to "push down" application-specific functionality to remote hardware to help customize queries and reduce the amount of data returned. Multiple network companies have created powerful SmartNICs that can inspect and process network data as it moves between the host and the network. Current generation SmartNICs feature multiple processor cores, sizable amounts of volatile and non-volatile memory, and direct access to high-speed communication networks. As such, SmartNICs present a new opportunity for optimizing application workflows in HPC platforms.

One promising setup to embed SmartNICs in the network fabric for optimizing application workflows is to co-locate them with HPC compute nodes. Figure 4.1 illustrates possible interactions between the host and a local BlueField-2 SmartNIC with virtual protocol interconnect (VPI) or InfiniBand support. This SmartNIC has eight A72 ARM processor cores running at 2.75 GHz, 16 GB of DRAM, and 60 GB of eMMC storage. The connection with the host is via PCIe Gen 4.0 x 16 lanes. For network capability, it includes two 100 Gb/s network ports that can interact with either InfiniBand or Ethernet. The default software stack for the card boots an Ubuntu 20.04 Linux installation from the eMMC storage. This OS operates independently from the host and is visible through drivers that provide network and console access to the card. The embedded processors can be configured to either (1) intercept traffic between the host and the network (i.e., embedded function mode) or (2) serve as a separate host that shares access to the network ports (i.e., separated host mode). Special-purpose hardware accelerators are available for offloading encryption, compression, and regular expression operations.

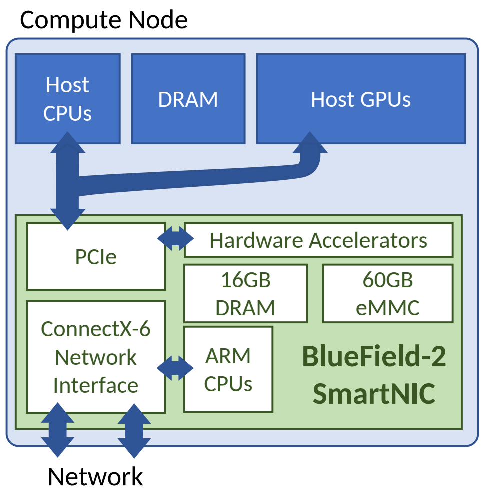


Figure 4.1: HPC Compute node with a BlueField-2 SmartNIC

4.2 General Micro-operations

One of the main questions when considering offloading functions to embedded devices is: which functions should be offloaded to achieve the maximum benefit? Ideally, answering this question requires evaluating the performance of every possible function that an embedded system is capable of to identify those that specific applications can leverage.

4.2.1 Benchmark Considerations

Accurately capturing the performance matrix of embedded devices is difficult due to their unique architectures and the limited resources they possess. Direct evaluation using applications or integrated tests often fails to identify the strengths and weaknesses of device operations, as the overhead from weak operations can overshadow the few strong operations. To address this issue, microbenchmarks are well-suited for evaluating specific operations of embedded devices since each focuses on particular aspects of their system or hardware.

For this evaluation, we selected the `stress-ng` tool [131], which comprises a comprehensive set of stressor functions designed to test and cover a broad spectrum of low-level system operations. For example, the msync stressor is used to test the msync(2) system call, while the CPU stressor evaluates the CPU's floating-point, integer, and bit manipulation operations individually. With 250 stressors that cover a wide range of resource and operation domains, including disk IO, network IO, DRAM, filesystem, system calls, CPU operations, and CPU cache, `stress-ng` is a robust and reliable tool for microbenchmarking. Inside the `stress-ng` tool, these test domains are classified into "classes." Our evaluation collected performance results from multiple systems and analyzed the differences both at the individual stressor and stressor class levels, providing a detailed insight into an embedded device's strengths and weaknesses.

4.2.2 Normalization

One challenge in comparing the performance results from various stressors lies in the inconsistent performance units they use. While each stressor reports the performance result in terms of the execution rate (i.e., "bogo-ops-per-second", read as "bogus operations per second") for the specific test operation it conducts, varied overheads associated with the test operations of different stressors make a direct comparison of these performance figures infeasible. Consequently, it becomes difficult to answer the specific question of which operations can perform better on the embedded device. Listing 2 shows the example results from two stressors executed on the same machine.

```
- stressor: branch
  bogo-ops: 14511254875
  bogo-ops-per-second-usr-sys-time: 36305366.212159
  bogo-ops-per-second-real-time: 1451159699.206867
  wall-clock-time: 9.999764
  user-time: 399.700000
  system-time: 0.000000

- stressor: memrate
  bogo-ops: 283
  bogo-ops-per-second-usr-sys-time: 0.700877
  bogo-ops-per-second-real-time: 27.980803
  wall-clock-time: 10.114077
  user-time: 388.940000
  system-time: 14.840000
```

Listing 2: Stress-ng example results

To address this issue, we have normalized the stressor performance results relative to those obtained from a Raspberry Pi 4B (RPi4B) — specifically the 4 GB

DRAM model. This approach is similar to the introduction of the MBWU for normalizing the performance of data access workloads, albeit with a slight modification; in this context, the unit of measurement is a **Raspberry Pi-based work unit, or RPWU**, as the "WU" is normalized to a RPi4B. It should be noted that while the RPi4B is not a high-performance embedded system, its widespread adoption makes it an ideal reference system for the comparison of our results by other researchers interested in the data.

4.2.3 Performance Characterization

To evaluate the relative advantages of different operations running on the BlueField-2, we executed `stress-ng` on the BlueField-2 SmartNIC and a variety of host systems available at CloudLab [200], as listed in Table 4.1. Among these 12 host systems, 11 are equipped with Intel x86 processors; the remaining system, the m400, is an ARMv8-based system. It should be noted that while it may not be appropriate to compare the performance of the BlueField-2 card's embedded processor to that of general-purpose servers' processors, the host systems used in this study are relatively outdated. For instance, the d710 operates on a on a 12-year-old processor.

We sequentially executed all available stressors on each test system. Every stressor ran for a fixed period of 60 seconds, with one instance being launched on each online CPU core. To ensure the accuracy and reliability of our results, we repeated this execution process five times on each system and computed the average performance for each stressor. Certain stressors were not executed due to various reasons, such as requiring root privileges or specific hardware features that were not available on the test system. For instance, the rdrand stressor was not executed on the BlueField-2 SmartNIC and the m400 system, because the

ARM CPU does not support the *rdrand* instruction. Figure 4.2 displays the final normalized performance numbers of stressors from all test systems.

Table 4.1: Micro-operations Test Systems

ID	System	CPU (Release Date)	Cores	DRAM	Disk	NIC
1	c220g1	Intel E5-2630 v3 @ 2.4 GHz (Q3'14)	2 x 8	128GB (DDR4-1866)	2 x SAS HDD, 1 x SATA SSD	2 x 10Gb, 1 x 1Gb
2	c220g2	Intel E5-2660 v3 @ 2.6 GHz (Q3'14)	2 x 10	160GB (DDR4-2133)	2 x SAS HDD, 1 x SATA SSD	2 x 10Gb, 1 x 1Gb
3	c220g5	Intel Xeon Silver 4114 @ 2.2 GHz (Q3'17)	2 x 10	192GB (DDR4-2666)	1 x SAS HDD, 1 x SATA SSD	2 x 10Gb, 1 x 1Gb
4	c6220	Xeon E5-2650 v2 @ 2.6 GHz (Q3'13)	2 x 8	64GB (DDR3-1866)	2 x SATA HDD	2 x 10Gb, 4 x 1Gb
5	c8220	Intel E5-2660 v2 @ 2.2 GHz (Q3'13)	2 x 10	256GB (DDR3-1600)	2 x SATA HDD	2 x 10Gb, 1 x 40Gb IB
6	d430	Intel E5-2630 v3 @ 2.4 GHz (Q3'14)	2 x 8	64GB (DDR4-2133)	2 x SATA HDD, 1 x SATA SSD	2 x 10Gb, 2 x 1Gb
7	d710	Intel Xeon E5530 @ 2.4 GHz (Q1'09)	1 x 4	12GB (DDR3-1066)	2 x SATA HDD	4 x 1Gb
8	dss7500	Intel E5-2620 v3 @ 2.4 GHz (Q3'14)	2 x 6	128GB (DDR4-2133)	45 x SATA HDD, 2 x SATA SSD	2 x 10Gb
9	m400	ARMv8 Atlas/A57 (64-bit) @ 2.4 GHz	1 x 8	64GB (DDR3-1600)	1 x M.2 SSD	2 x 10Gb
10	m510	Intel Xeon D-1548 @ 2.0 GHz (Q4'15)	1 x 8	64GB (DDR4-2133)	1 x NVMe SSD	2 x 10Gb
11	r320	Xeon E5-2450 @ 2.1 GHz (Q2'12)	1 x 8	16GB (DDR3-1600)	4 x SATA HDD	2 x 1Gb
12	xl170	Intel E5-2640 v4 @ 2.4 GHz (Q1'16)	1 x 10	64GB (DDR4-2400)	1 x SATA SSD	4 x 25Gb
13	MBF2H516A-CENO_Ax	ARMv8 A72 (64-bit) @ 2.5 GHz	1 x 8	16GB (DDR4-1600)	eMMC flash memory	2 x 100 Gb/s or 1 x 200Gb/s

* All systems except the MBF2H516A-CENO_Ax (BlueField-2 SmartNIC) ran Ubuntu 20.04 (kernel 5.4.0-51-generic).

* Tests were all conducted on the ext3 filesystem.

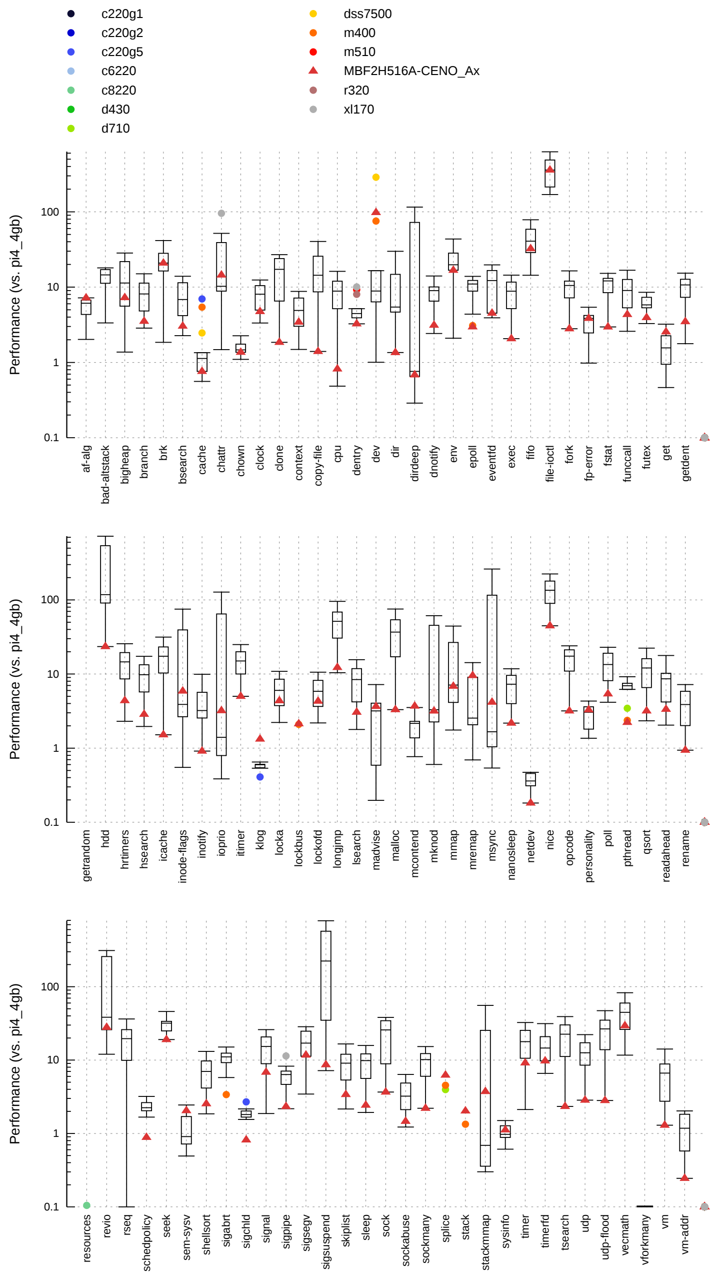


Figure 4.2: Box plotting the relative performance of a set of stress-ng stressors ran on 12 general-purpose systems and the BlueField-2 SmartNIC (MBF2H516A-CENO_Ax). Each stressor was run for a duration of 60 seconds. The performance figures have been normalized with respect to the Raspberry Pi 4B (4 GB model). Triangle data points denote the performance of the BlueField-2. Other data points are plotted only if they are outliers, falling outside the range of the corresponding whisker. A full version of this plot can be found at [152].

Individual Results Analysis

As anticipated, the performance of most operations on the BlueField-2 card falls short compared to the majority of systems used for comparison, with the exception of the RPi4B. However, there were several tests in which the BlueField-2 outperformed the other systems, warranting further investigation. On the other hand, for operations where the BlueField-2 card performed significantly worse, system designers need to be careful when offloading functions that involve such operations, so these operations also deserve detailed examination. A detailed analysis of these two types of operations follows. Note that the number inside the parentheses next to each stressor name represents the ranking (from best to worst) of the BlueField-2 card's performance for that stressor among all test systems.

af-alg (#1): AF_ALG [239, 262] is the user space interface for accessing the kernel crypto API, which enables user programs to leverage the cipher and hash functions provided by hardware cryptographic accelerators. The BlueField-2 SmartNIC features multiple hardware accelerators for cryptography, including the 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, which significantly contributed to the outstanding performance in this stress test.

lockbus and mcontend (#1): The lockbus stressor tests data writes with pointer advancements while injecting write barriers in between. Similarly, the mcontend stressor employs multiple threads to concurrently update and read data residing in virtual memory regions mapped to the same physical memory pages. Both these stressors mimic the access patterns of applications that demand aggressive memory access. The superior performance exhibited by the BlueField-2 card in these two tests may be attributed to its cache subsystem's effectively designed policy.

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 operating system running on the SmartNIC. While the BlueField-2 card excels at some memory operations, it does not perform as well as other systems in more common operations, such as mmap and malloc. Therefore, it is difficult to generalize the memory access advantages of this card, as applications that heavily depend on these operations might encounter a significantly worse memory access performance compared to operating on a general-purpose host.

chattr and inode-flags (#5), ioprio (#6), file-ioctl (#7), dnotify and getdent (#12), copy-file, dentry, dir, and fstat (#13): These stressors focus on exercising filesystem interfaces. Given that the BlueField-2 card only has eMMC storage, the performance outcomes of these operations are as expected, and we do not expect to offload functions that would heavily rely on the performance of the on-card persistent storage. Note that getdent(2) and fstat(2) are system calls that can be easily triggered by simple commands such as "ls."

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 testing the CPU's logical and arithmetic operations using various mathematical algorithms. For example, the cpu stressor performs bit manipulations, computes square roots, determines the greatest common divisor, and calculates the Apéry's constant. It is intriguing to observe that the arithmetic performance of the BlueField-2, as indicated by the relative performance result of the cpu stressor, is even lower than that of the RPi4B. In contrast, the vecmath test on the BlueField-2 displayed comparatively superior performance than certain host systems, such as the m510 with a D-1548

CPU (Q4'15), the r320 with an E5-2450 CPU (Q2'12), and the d710 with a Xeon E5530 CPU (Q1'09). This observation is noteworthy as the vecmath operation could provide benefits to a broad range of data processing applications.

cache (#11), icache (#13): The cache stressor assesses the efficiency of the last-level CPU cache by repeatedly reading and writing values to and from it. The benchmark results indicate that the BlueField-2 slightly outperforms the E5-2630 v3 (Q3'14), the E5-2660 v2 (Q3'13), and the Xeon E5-2450 (Q2'12), but falls short of the RPi4B's performance. This result can be attributed to the fact that the BlueField-2 utilizes an L3 cache as its last-level cache, whereas the RPi4B utilizes an L2 cache. Additionally, the icache stressor evaluates the instruction cache performance of the CPU by simulating load misses caused by modifications to a function pointer. Based on the performance ranking of the BlueField-2 on these tests, it is fair to conclude that its CPU cache does not exhibit a significant competitive advantage over other systems.

sigsegv (#9), timerfd (#10), signal, clock, and timer (#11), itimer, sigpipe, sigsuspend, and sleep (#12), nanosleep (#13): The performance of these stressors reflects the interrupt performance of the operating system. Based on the results, it is recommended that any functions offloaded to the BlueField-2 should, if possible, avoid reliance on the operating system's timing and interrupt interfaces.

readahead (#11), hdd and seek (#13): As the storage IO performance is directly impacted by the performance of the underlying storage media, it is unlikely that these storage IO-related tests will exhibit exceptional performance. In fact, the eMMC flash used by the BlueField-2 is slower than most of the enterprise-class storage drives used by the CloudLab servers.

sem-sysv (#2), fifo (#9), eventfd, poll (#11), futex (#12), hrtimers (#12), clone, exec, fork, nice, and pthread (#13): The stressors under consideration are scheduler-related, with the System V interprocess communication (IPC) [29] serving as a key communication mechanism between processes in Linux. Specifically, the sem-sysv stressor assesses the ability of a pair of processes to increase and decrease a shared semaphore, with exceptions injected to the system-call arguments. Notably, the BlueField-2 SmartNIC demonstrates exceptional performance in this stressor, surpassing all x86_64 systems. This result may be attributable to certain architectural advantages of the ARM processors, as evidenced by the high rankings of the m400 ARM (#1) and RPi4B systems relative to other stressor tests. However, the BlueField-2 performed poorly in other scheduler-related stressors, such as the futex stressor test, which evaluates the futex(2) [258] system call utilized to wait for a specific condition to become true.

sockabuse (#11), epoll, sockmany, sock, udp-flood, and udp (#13): The stressors in question evaluate the performance of the kernel network stack. Based on the rankings, it appears that the networking performance of the BlueField-2, in conjunction with the kernel network stack, is inferior to that of most other systems. In Section 4.3, we will conduct an in-depth evaluation of the network performance of this device.

Table 4.2 displays the performance ranking of the BlueField-2 SmartNIC for each stressor test conducted on all test platforms. Due to the considerable number of stressors evaluated, we have opted to present only the best and worst results of the BlueField-2. This list is valuable in gaining insights into the types of operations that should or should not be offloaded to the BlueField-2 SmartNIC.

Finally, to investigate the potential impact of thermal or caching effects on

Table 4.2: Stressor performance ranking of the BlueField-2 SmartNIC

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

* We only show the stressors that the BlueField-2 SmartNIC ranks $\leqslant 10$ or the last among all test systems.

Table 4.3: Changes in the performance ranking of the BlueField-2 SmartNIC between 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

stressor performance, we conducted a secondary round of testing where each stressor was run for only 10 seconds instead of 60 seconds. The results are presented in Table 4.3, which includes only tthose stressors where the BlueField-2 saw more than a two-place shift in the rankings. The stressors that showed the greatest changes in the ranking were those related to CPU and CPU cache performance. For instance, the bigheap stressor tests virtual memory performance by increasing the allocated memory size of a process using the REALLOC(3) [140] system call. Moreover, we found that the BlueField-2's stressor rankings in the 60-second test were generally higher than those in the 10-second test. This suggests that the ARM CPU on the BlueField-2 may require a warm-up period for optimal performance. As for thermal dissipation, we observed no significant influence on the BlueField-2's performance.

Class Results Analysis

`stress-ng` has categorized stressors into 12 classes, each representing a crucial aspect of the system's hardware or software. To ascertain whether the BlueField-2 possesses a domain of operations in which it outperforms the other systems, we examined the average relative performance of each stressor class for every test system. Full details on the results of this investigation can be found at [153].

In general, we observed that the BlueField-2's average performance aligns with that of the 12-year-old x86_64 server d710 and the ARM server m400. Nonetheless, the relative performance of the BlueField-2's stressor classes exhibits significant variations, with no single class of operations demonstrating dominant performance over other systems, except for the DEV class. The DEV class, comprising five stressors, probes multiple Linux device interfaces under /dev, such as /dev/console, /dev/full, /dev/null, /dev/loop*, among others. The efficacy of these interfaces, however, may not easily translate into tangible benefits for the functions offloaded to the BlueField-2.

However, the effectiveness of these interfaces may not easily translate into tangible benefits for offloaded functions. While the class-based stressor analysis did not reveal any promising information, it does suggest that the SmartNIC requires more robust resource integration than what the BlueField-2 currently offers for executing traditional system operations. This enhancement is necessary to handle relatively complex, latency-sensitive functions offloaded from hosts. In contrast, asynchronous or throughput-sensitive functions have different resource requirements in the execution environment. By leveraging potential advantages such as data locality and cost-efficient parallelization offered by many of these low-power systems, it is still possible to develop a more optimized solution for executing these asynchronous or throughput-sensitive functions within the SmartNICs.

4.2.4 Summary

In summary, our evaluation has revealed that the BlueField-2 SmartNIC exhibits superior performance only in a limited set of traditional system operations compared to general-purpose server systems. These operations span diverse domains of the system, including memory contention, cryptographic, and IPC operations. For functions that primarily rely on these operations, offloading to the BlueField-2 SmartNIC could lead to improved performance. However, it is important to note that the superior performance of an individual operation on this card does not necessarily lead to the better performance of the operation class as a whole. For instance, while the mcontend and stack stressor tests exhibit the best performance on the BlueField-2 among all test systems, the average relative performance of the memory class stressors on the card is inferior to that of most other systems used for comparison in our experiment. Therefore, when offloading functions to the BlueField-2 SmartNIC, it is crucial to carefully tailor the functions to leverage the limited advantageous operations available on the card, unless hardware accelerators can be utilized.

Generally, the BlueField-2 SmartNIC shows better performance when handling memory-related operations compared to CPU-, storage IO-, and kernel network stack-related tasks. However, there are notable exceptions within the CPU operations, particularly those involving encryption and vector calculations. These exceptions can be attributed to the built-in accelerators on the SmartNIC that enhance these specific operations. The optimized ARM architecture of the BlueField-2 SmartNIC may also contribute to the superior performance observed for vector calculations.

Based on our analysis, one type of function that has the potential for profitable offloading to the SmartNIC is transparent encryption/decryption or compression/decompression functions utilized in data serialization. Moving these functions to the SmartNIC could significantly save the host's CPU cycles for applications while reducing function execution latency. Other types of functions that may be profitable for offloading are those that can effectively leverage the efficiency of the SmartNIC's (virtual) memory access operations, IPC operations, and vector mathematics operations. A concrete example could be a data transformation function utilizing Apache Arrow [147], the de facto in-memory data processing library. This library stores data in a dense columnar format and provides an IPC mechanism for transferring Arrow columnar arrays across processes and systems. This approach enables data communication between different applications to use a well-defined data representation and eliminates the potential transformation overhead caused by translating between on-disk and in-memory representations. Apache Arrow has a large open-source community that has attracted data scientists and software developers to build Arrow-based data processing systems [188, 187, 238, 6, 117].

4.3 Network Processing

When analyzing the embedded computing resources of a SmartNIC, a key question arises regarding the amount of processing headroom available after accounting for the network stack overhead. Accurately quantifying the processing headroom on a SmartNIC enables applications to identify the most efficient network stack to use within the embedded environments, and to establish a performance envelope that stipulates the complexity level of latency-sensitive network functions targeted for offloading.

4.3.1 Benchmark Considerations

Handling high-speed network transfers can consume significant computing resources, depending on the network stack in use. To measure the processing headroom during network transfers, we aim to obtain a measurement result as close to the upper bound as possible. This means determining the maximum remaining CPU time that can be allocated to an offloaded function without impacting normal network performance. To achieve this objective, it requires an efficient and lightweight traffic generator that imposes minimal overhead on network data transfers.

Our initial tests of commonly used network performance measurement utilities, such as iPerf [228], nuttcp [227], and Netperf [119], showed suboptimal performance. The poor performance of these tools may be due to their high-overhead communication between the user and kernel space. Therefore, we tend to solutions where the core network function is run only in the kernel or user space.

The Linux `pktgen` [232] is a kernel space module that generates UDP packets within the kernel space and injects them directly into the kernel IP network stack. Utilizing `pktgen` for processing headroom measurements offers several advantages. First, we observed that its single-thread performance was roughly 15% higher than the aforementioned measurement tools in resource-restricted environments. Second, `pktgen` has built-in support for symmetric multiprocessing by binding a generator thread to each CPU core. This feature is crucial for high-speed network environments, especially for network speeds at 100Gb/s, which are hard to saturate with only a single core (e.g., a single instance of iPerf achieved less than 40Gb/s in a previous study [72]). Finally, `pktgen` natively supports multi-queue devices, enabling the mapping of the socket buffer's transmission queue, network interrupts, and associated generator thread to the same CPU core. This feature reduces the overhead of cross-core communication, thus significantly improving the throughput that a single thread can generate.

Additionally, `pktgen` offers several parameters for performance tuning, of which we highlight three as being critical to our experiments. First, the "delay" parameter controls the duration allocated for emitting a burst of packets (not the gap between consecutive bursts). By setting the delay to a value smaller than that required for sending a burst of packets, throughput remains unaffected. However, setting the delay larger than the send time of a burst results in the generator thread spinning until the next allocated time. Altering this parameter allows for the capture of free CPU time that is not involved in handling network traffic. Second, the "clone_skb" parameter governs the policy of packet reuse for transmission. Setting this parameter to zero eliminates the overhead of memory allocation for packets. Although this parameter is useful for modeling the data pattern based on its repetition, for measuring the upper bound of the processing headroom, it is recommended to set it to zero. Finally, the "burst" parameter specifies the maximum number of packets that can be queued before triggering the bottom half of the network stack. This parameter can be adjusted to optimize interrupt coalescing.

4.3.2 Methodology

Measuring the processing headroom with `pktgen` requires a two-step process, primarily because each `pktgen` generator thread operates in an infinite loop and fully occupies the associated CPU core when activated. The first step involves determining the minimum configuration necessary to achieve the highest possible bandwidth with the SmartNIC. This is accomplished by performing parametric sweeps over the packet size, the number of generator threads, and the value of "burst", while recording the variations in throughput. In the second step, the value of "delay" is gradually increased to change the allocated time for sending a burst, with the goal of finding the maximum delay that the SmartNIC can accommodate while still maintaining the same network throughput. The processing headroom can then be calculated by subtracting the time spent sending a batch of packets without delay, as determined in the first step, from the maximum delay evaluated in the second step.

In addition to measuring the processing headroom for the BlueField-2 SmartNIC, we applied this method to a general-purpose host for comparison purposes. The results are discussed in Section 4.3.3. However, a different approach was necessary to evaluate the processing headroom when the BlueField-2 SmartNIC is running under the embedded function mode, as this mode uses a different data path for host communication. In this case, we did not run `pktgen` within the SmartNIC itself but instead ran it on the host to allow the SmartNIC to forward packets using its ARM cores. The evaluation for this mode is discussed in Section 4.3.3.

4.3.3 Network Processing Headroom Evaluation

Headroom Evaluation in Separated Host Mode

In the initial step of our performance evaluation, we conducted a sweeping analysis of packet sizes ranging from 128B to 10KB. For packets larger than 10KB, it resulted in the termination of the `pktgen` process and hence were excluded from the analysis. The throughput measurement results are depicted in Figure 4.3.

While it was anticipated that the BlueField-2's embedded processors would be less powerful than those of host processors, it was unexpected that the SmartNIC would struggle to saturate the hardware bandwidth with its ARM cores. Our experiments revealed that the BlueField-2 was only capable of saturating 60% of the total bandwidth, even with the largest packet size and all CPU cores. This finding underscores the fact that achieving 100 Gbps throughput is still a resource-intensive task, even for modern ARM cores in the absence of network stack optimization.

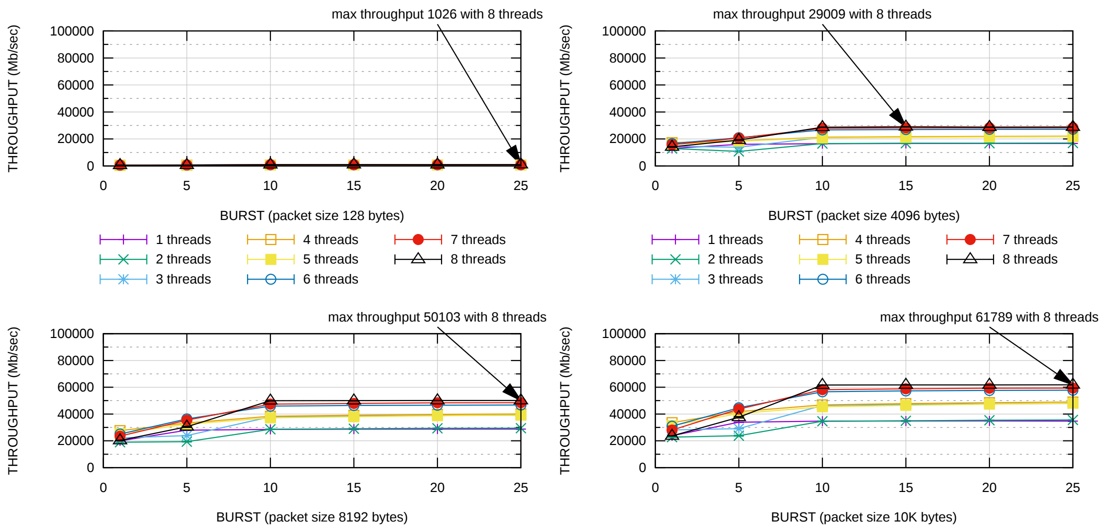


Figure 4.3: BlueField-2 throughput performance in the separated host mode

Additionally, the suboptimal performance gain observed by simply offloading host functions with traditional network stacks to the SmartNIC underscores the need for adapting network stacks to leverage the strengths of the SmartNIC. In other words, offloading network functions to the SmartNIC alone is not sufficient to attain optimal performance. Rather, optimizing network stacks is also essential.

Given that `pktgen` cannot saturate the full network bandwidth of the BlueField-2 with its ARM cores, we seek to investigate the amount of CPU time that can be allocated to offloaded functions if the required bandwidth is reduced to half of the full specification. As indicated in Figure 4.4, the maximum delay for achieving 50 Gbps throughput with a packet size of 10KB and burst size of 25 is approximately 320μs. In the case of transmission without delay, we can deduce from Figure 4.3 that the time required to send the same number of packets of the same size in a burst is 253μs. The difference between the two values represents the remaining wall clock CPU time available for each core to execute the offloaded functions' computation logic, accounting for 21% of the time taken to send a packet burst. It is important to note that these results are based on the assumption that we aim to only achieve 50% of the total bandwidth.

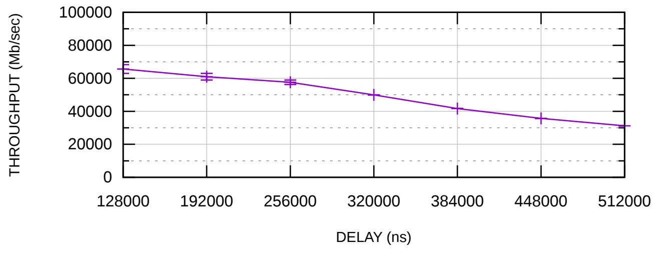


Figure 4.4: BlueField-2 throughput performance by varying the delay configuration (8 threads, packet size 10KB, burst 25)

To provide a comparison of processing headroom between a general-purpose host and the BlueField-2 SmartNIC with the same network interface, we conducted a similar evaluation on the host where the SmartNIC card was installed. The host was a CloudLab machine of the r7525 type, featuring two 32-core AMD 7542 CPUs operating at 2.9 GHz and 512 GB DDR4-3200 memory. Through experimentation with varying packet sizes from 128B to 1KB, we determined that a packet size as small as 832B is sufficient to saturate the full link. Figure 4.5 presents the results of throughput as a function of the burst and packet size. The host's superior system resources allow the network link to be fully saturated with only five threads (equivalent to 5 vCPU cores) and a burst size of 25. Although additional threads have little impact on throughput for the 832B packet size, we observed a decrease in throughput with larger packet sizes (e.g., 1KB) when more threads were utilized. We attribute this decrease in performance to resource contention.

To assess the host's capacity to maintain full bandwidth, we introduced delays for bursts with the minimum packet and burst size identified in the previous step, as depicted in Figure 4.6. The result indicates that the host can sustain an 8-us delay per burst under the same number of threads. This delay accounts for less than 1% of the CPU time available to handle additional computation logic on the

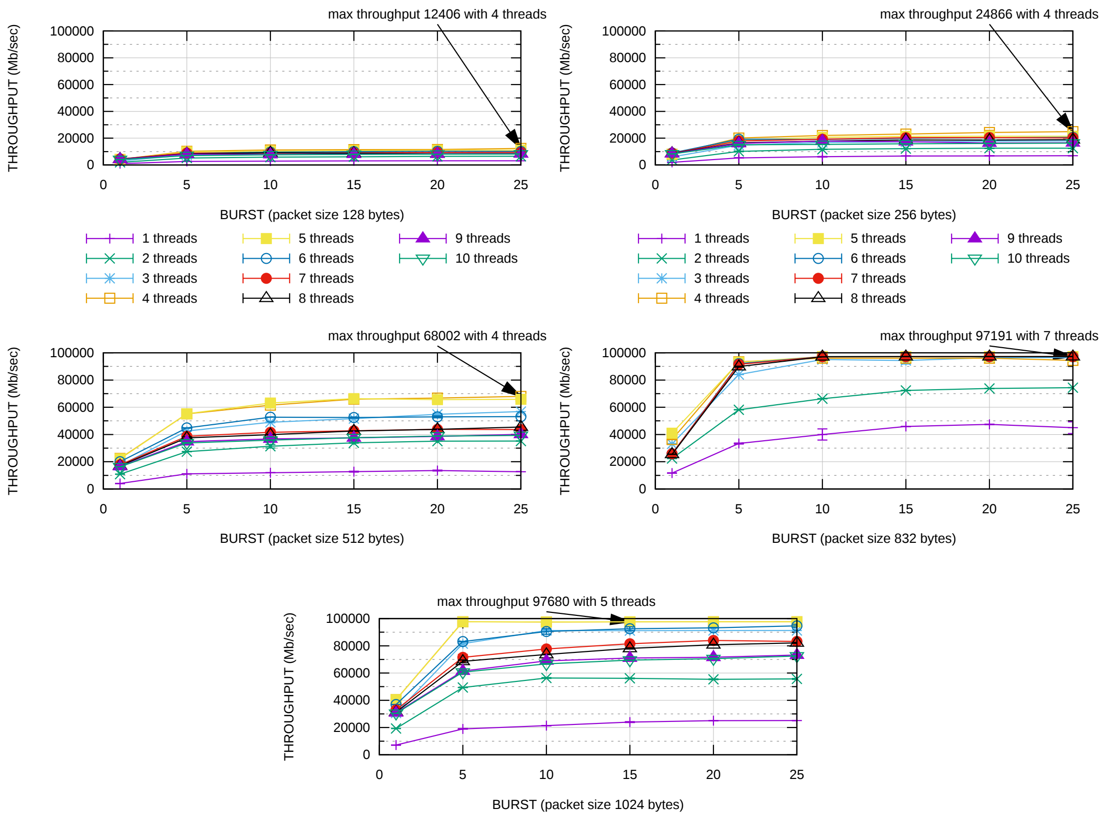


Figure 4.5: r7525 throughput performance in the separated host mode

aforementioned five cores. It is, however, noteworthy that the host still has the remaining 123 vCPU cores available for utilization by applications.

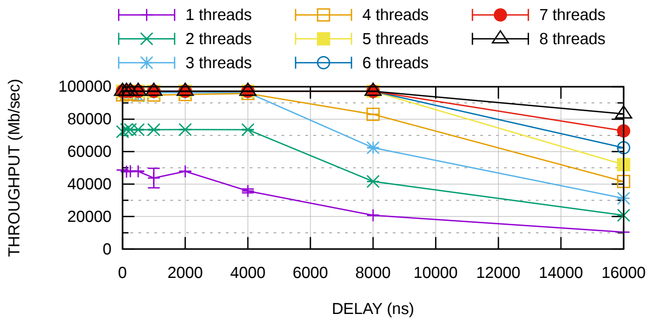


Figure 4.6: r7525 throughput performance by varying the delay configuration (packet size 832B, burst 25)

Headroom Evaluation in the Embedded Function Mode

Given that the BlueField-2 utilizes a distinct data path for communicating with the host in its embedded function mode, it prompts the question of whether the SmartNIC experiences a change in the processing headroom in this mode. In the embedded function mode, the ARM processor situates itself between the network port and the host, mandating that all traffic between the network interface on the host's operating system and the physical network port cross through the ARM system. To evaluate the upper bound processing headroom of the SmartNIC, we ran `pktgen` on the host rather than the ARM system to maximize the SmartNIC's processing resources intended for traffic bridging. The BlueField-2 supports both the kernel IP stack and the userspace network stack (i.e., DPDK [42]) for bridging Ethernet traffic. The throughput observed from the host under these two network stacks is illustrated in Figure 4.7. Our findings reveal that, when utilizing the kernel IP stack, the SmartNIC has a CPU availability of 78.5%, whereas when operating under DPDK, it has an availability of 87.5%. These results indicate a significant leap in processing headroom compared to the separated host mode, suggesting that the embedded function mode makes for a more advantageous configuration for in-transit data processing in scenarios where the functionalities of the separated host mode are not mandated. Furthermore, our analysis indicates that, similar to host systems, DPDK remains the more efficient option for managing traffic on this embedded system.

4.3.4 Summary

Quantifying the processing headroom for network functions running on a SmartNIC enables the delineation, in a quantitative manner, of the capacity that the SmartNIC reserves for dealing with streaming data without curtailing its network

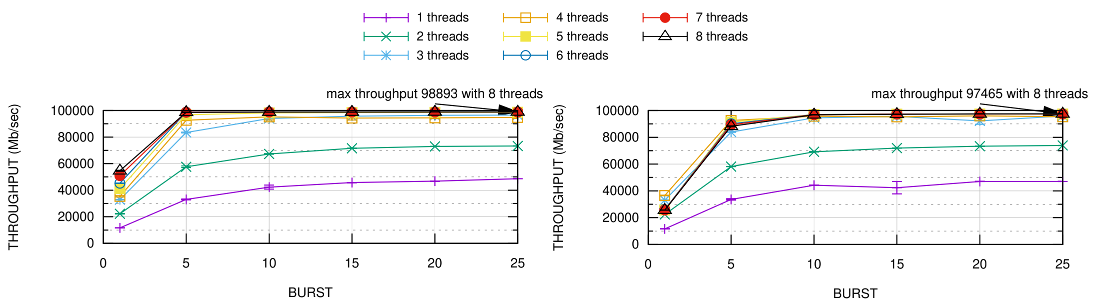


Figure 4.7: r7525 throughput performance in the embedded function mode (packet size 832B). The left and right figures show the performance with DPDK and kernel IP network stack for traffic bridging, respectively

bandwidth performance. Despite the noticeable overhead experienced by the current generation of the BlueField SmartNIC when processing Ethernet packets with the Linux kernel IP stack, it is anticipated that future generations will exhibit significant improvements in this area, making it easier for host network functions relying on traditional network stacks to harness the performance benefits derived from offloading to the SmartNIC. The embedded function mode of the BlueField-2 offers an alternative means for running network functions targeting packet interception and manipulation. This mode involves significantly lower overhead and, as such, appeals to network use cases operating at a lower level. The DPDK userspace network stack provides an opportunity for offloaded network functions to trade protocol functionality for enhanced network performance when running in this mode, as evidenced by our results, where it incurred the least overhead from this network stack.

4.4 Data Partitioning

Transitioning from the detailed discussion of opportunities for offloading general functions to SmartNICs, it is important to also consider practical applications that could potentially benefit from this shift. Notably, HPC and Geographic Information Systems (GIS) are two areas that often deal with intricate data management services and complex data flows, especially those pertaining to particle data.

4.4.1 Particle Data Flows

Many HPC and GIS applications operate on particle data and rely on complex data management services to route particle state information between producers and consumers distributed throughout the network. While particle datasets are smaller in size than multimedia datasets, application data flows can be challenging to implement because the datasets contain many small items that are tedious to inspect. As such, it is beneficial to consider hardware environments that can offload the task of reorganizing and sifting through the in-transit data. In this section, we provide application examples from the HPC and GIS spaces, and discuss a common data-flow use case where data is transitioned from a spatially-organized form to a temporally-organized form.

Particle Simulations in Scientific Computing

Many simulations in scientific computing employ particle-in-cell (PIC) methods [106, 15, 62] to model different phenomena. These simulations manage a large collection of discrete particles and track their progress as they transit through time and space. A particle is defined by a small amount of state information, such as its position, velocity, charge, and type. Given that simulation fidelity improves as the number of particles in the simulation increases, researchers typically leverage parallel simulation techniques to distribute the data and work across many compute nodes. Although a simulator's particle data may contain a treasure trove of information for scientists, the sheer size of this data makes it infeasible to save except in the case of occasional checkpointing. Analysts may run supplemental analysis applications in parallel with the simulation and inspect subsets of the data without impeding the simulation. To enable this, the ability to rapidly sample and export sizable portions of the data would allow users to apply external analytics in a workflow and inspect how the individual particle states evolve over time.

Asset Tracking with Geographic Information Systems

A similar need to collect and process large amounts of particle data can be found in GIS applications that manage sensor data about real-world assets, such as airplanes, ships, and land vehicles. Although the "particles" in these systems are significantly larger in physical size than those in the simulations, the processing challenges for manipulating the data flows are the same: different distributed sensor systems produce continuous feeds of observation data that need to be reorganized to be of value to downstream consumers.

Reorganizing Spatial Indices to Temporal Indices

One hardship of working with particle data flows is that there are significant differences between the way producers and consumers expect data to be organized. Producers typically organize data in a spatial manner, where each sensor generates an update for all items in a physical region during a particular time interval. In contrast, analytics consumers often need data organized temporally for each item. As illustrated in Figure 4.8 with airplane data, temporal tracks (b) can yield better insight into patterns of activity than positional snapshots (a) alone.

Distributed, log-structured merge (LSM) trees [190] are a convenient mechanism for converting particle data flows from a spatial organization to a temporal

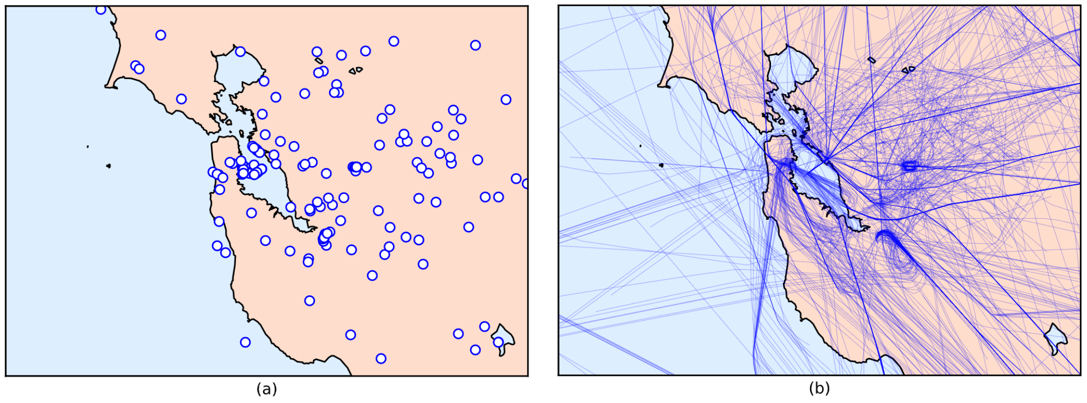


Figure 4.8: Airplane position data in a (a) point form for a snapshot in time and (b) track form for a window of one hour

organization when multiple producers and consumers are involved. In this approach, a collection of processing elements is used to reorganize data as it moves through different stages in the tree. Each processing element absorbs incoming data until it reaches its storage capacity. When compaction is necessary, data is split based on particle IDs and transmitted to the next appropriate processing element in the tree. While processing elements only need to store, sift, and transmit blocks of particles, they can greatly improve the searchability of online datasets.

4.4.2 Software Infrastructure for In-transit Processing

Data-intensive applications in both science and commercial enterprises are often constructed using multiple systems with independent implementation histories and choices of programming languages. A key operating expense of these applications is the movement of data across these systems. But what sounds like a problem of moving data between systems is really the challenge of efficiently (1) converting the data from a system's internal in-memory representation to a wire format and (2) accessing large amounts of data via record-by-record API calls.

This is precisely the challenge that Apache Arrow[1] set out to address, an open-source project that since 2016 has been quickly gaining adoption within the data science community.

Apache Arrow

The key insight underlying the design of the Apache Arrow ecosystem is that by creating an efficient, open data processing platform around a common, and efficient in-memory data representation, with many different programming language bindings (including C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust), data can move efficiently between the ecosystem's data processing engines running on different systems. Data processing and exchange can be implemented with a number of building blocks, including the Parquet file format [242], the Flight framework for efficient data interchange between processes [170], the Gandiva LLVM-based JIT computation for executing analytical expressions by leveraging modern CPU SIMD instructions to process Arrow data [191], the Awkward Array for restructuring computation on columnar and nested data [193], and the streaming data processing engine named Acero [53] to process complex user queries on tabular data. On top of these building blocks exist a number of Arrow integration frameworks, including the Fletcher framework that integrates FPGAs with Apache Arrow [188], NVIDIA's RAPIDS cuDF framework that does similar for GPUs [197, 223], the Plasma high-performance shared-memory object store [181], the Skyhook distributed storage plug-in to embed Arrow processing engines within Ceph storage objects [43, 45], and the Substrait effort to standardize an open format for query plans between query optimizers and processing engines [177]. Today, many more projects are adopting the Apache Arrow in-memory representation and the Dataset Interface that ab-

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

stracts over a variety of file formats and other data sources [14], such as Apache Spark [215], Dask [206], and Polars. The amount of significant investment poured into this ecosystem is reflected by its recent cadence of four major version releases per year, with the most recent being version 9.0.0, which resolved 1061 issues by 114 distinct contributors over three months.

Data Organization in Apache Arrow and Opportunities

Apache Arrow represents tabular data in a columnar, randomly-accessible, in-memory format that allows for nested data structures and null values. The format is designed to maximize CPU throughput by optimizing the data layout for pipelining, SIMD instructions [178], and cache locality, enabling zero-copy access in shared memory. Data is communicated by schema information involving one or more optional metadata dictionary batches followed by record batches. A record batch is composed of multiple arrays, each representing a part of the data from one or more fields of a table. Record batches are designed to be the unit of data processing communicated to and from processing engines. Batching of records minimizes the need for record-based API calls, and the batch size can be optimized for pipeline processing, while the columnar layout allows for SIMD instructions.

Arrow IPC format is a protocol that encodes record batches into contiguous bytes for storing in either files or memory. This encoding process is known as serialization. Fig. 4.9 shows how a typical Arrow table is serialized into a byte sequence in the IPC format.

The schema of a table is first serialized and written to the output memory buffer. Then, for each record batch, the arrays it contains will be serialized one after another according to their types. With all record batches being serialized

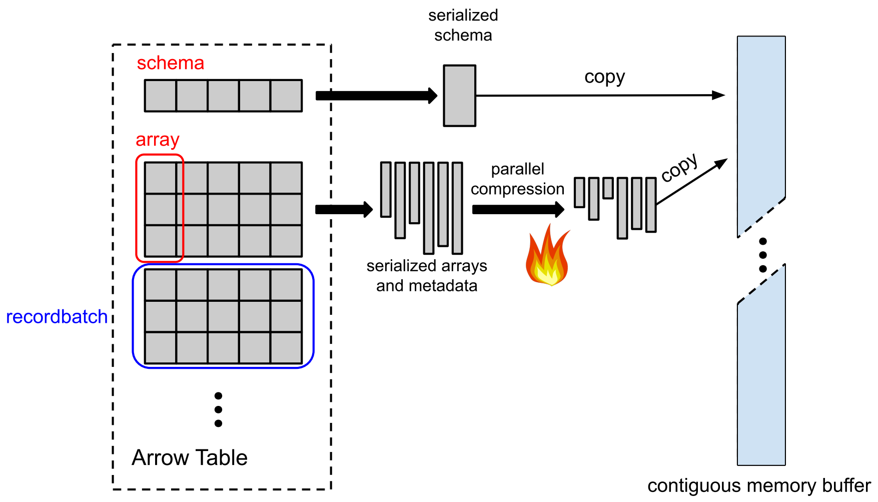


Figure 4.9: A simplified data serialization process of an Arrow table

resulting in a buffer vector, these buffers will be compressed in parallel for better compression throughput. The number of threads that will be spawned in this process typically matches the size of the buffer vector. For most array types, serializing an array produces two buffers — a data buffer and an additional buffer containing the metadata called the validity bitmap. As a result, the total number of threads started is a multiple of the number of columns, which can keep many cores busy. For example, in one of our reference datasets for experiments, the loaded Arrow table contains 17 columns; however, the compression phase spawns 35 threads occupying more cores than the ones available from a CPU socket. As such, the generated compression workload may hinder or stall performance-critical applications such as simulations that are running on the same host. Leveraging the compression accelerator from the BlueField-2 SmartNIC provides an opportunity to break the dependence of compression performance on intensive computing resource occupation.

Applicability to other data management libraries

While Apache Arrow meets several of our needs, our work can be adapted to other important data management libraries. HDF5 [98] is an established library for representing scientific data in stored data. Although it does not include a rich set of primitives for dispatching queries on in-memory data, it does provide a modular interface for extending the library's capabilities. Kokkos [75] is a computational library that aims to provide performance portability across different data-parallel architectures. Its data views provide a simple structure for hosting data vectors in a way that simplifies transport. Similarly, VTK-m [174] is a library for facilitating data-parallel visualization operations.

4.4.3 Performance of Partitioning Particle Data

In an effort to more comprehensively evaluate the potential of SmartNICs in processing particle data flows, we implemented a data partitioning algorithm used in LSM trees. In this work, we used Apache Arrow to represent particles in a tabular form that is suitable for transfer over the network and leveraged Arrow's filtering operations to split a table into smaller tables based on particle IDs. We measured the amount of time required to unpack, partition, and repack data for three particle datasets from different communities to demonstrate the flexibility of this approach.

Implementation

We constructed a C++ program that inspects and processes in-transit data objects in network data flows. This program is supplied with a contiguous-memory data object and is expected to provide one or more contiguous result objects that are to be sent to different locations. For this work, we use Apache Arrow's

IPC methods to handle transformations between a serialized object that can be transported in the network and an in-memory format that is suitable for tabular computations.

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. Although Arrow provides a group-by function that would be useful for performing a split in a single pass, it is currently limited to statistical operations. As such, we implemented the partitioning as a multistep algorithm that executes a select query to generate each table. While far from ideal, this approach is acceptable in the LSM tree work because of the low-fanout requirements of the distributed algorithm.

Reference Datasets

Three particle datasets were used in these experiments to provide better insight into the performance of the algorithm with different data:

- **TrackML Particle Tracking Challenge** ("Particles") [12]: 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") [217]: The OpenSky Network collects worldwide ADSB information for airplanes from volunteers. Entries contain 16 fields composed of a mix of numerical and string values.

- **NOAA Maritime** ("Ships") [179]: 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.

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.

Experiments

Performance experiments were conducted on a 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 tabular data into 2 to 16 output partitions. As depicted in Figure 4.10, 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 [51], or Zstd compression [52]. Figure 4.11 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. Repacking the data, however, is similar in all cases. This overhead highlights the fact that serialization by itself is an expensive operation.

Examining the output sizes of the individual, serialized partitions generated in the second experiment provides greater insight into how partitioning affects

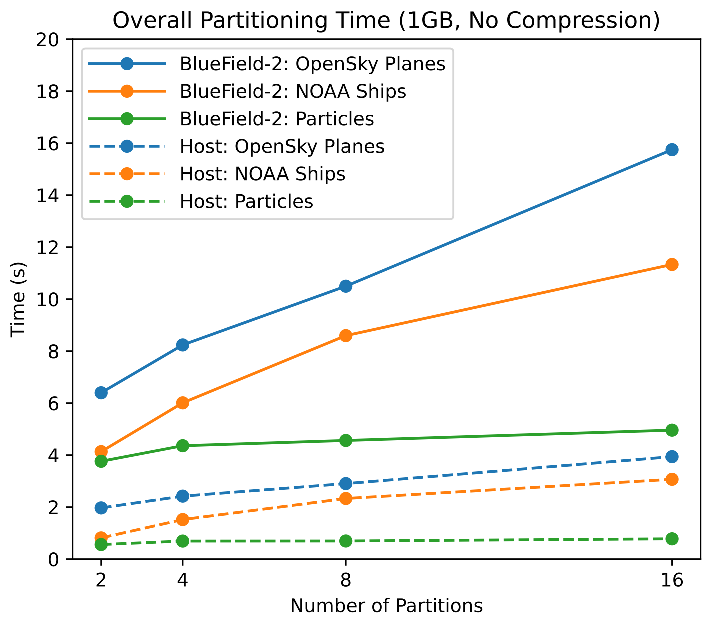


Figure 4.10: Overhead for partitioning without compression

compression results. Figure 4.12 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.

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 under-

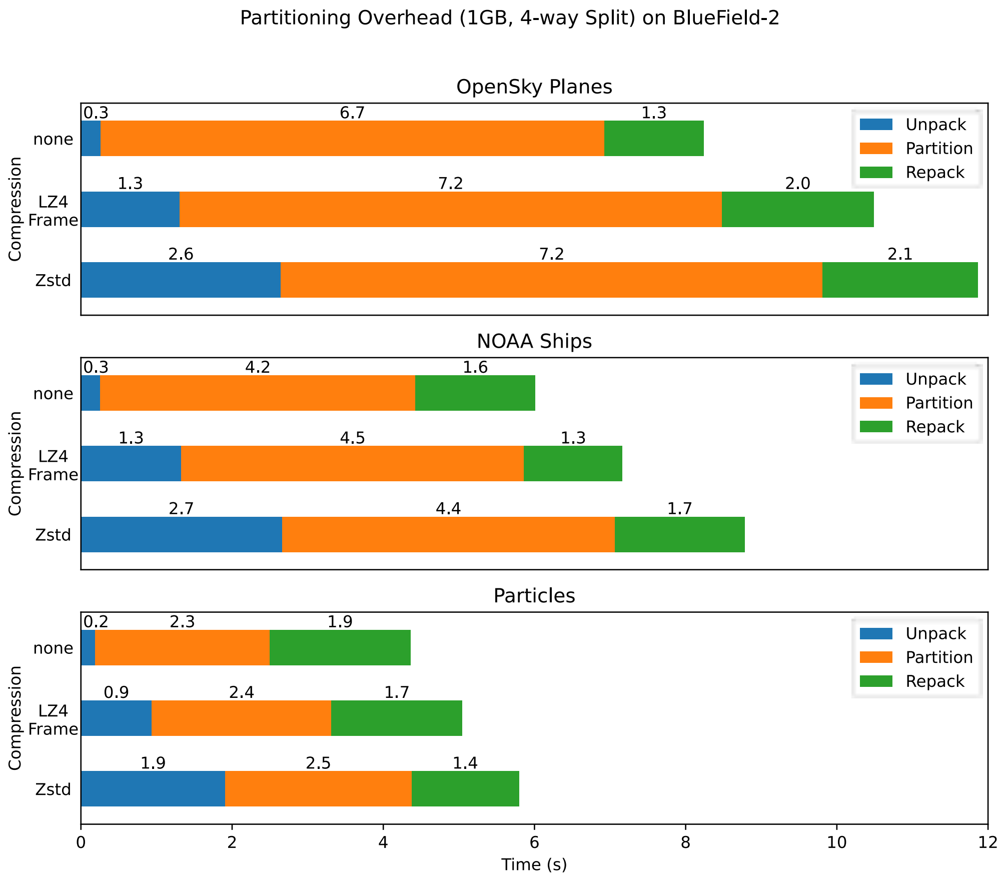


Figure 4.11: Timing breakdown for a 4-way split on the BlueField-2 using different software compression methods

stand the characteristics of their data and select partition address bits that will result in balanced outputs.

4.4.4 Summary

While the host processors in our data partitioning experiment yielded better performance, it is important to note that the BlueField-2's embedded processors were performant enough to be of value in many data flows. Scenarios where producers generate periodic bursts of data are applicable, as the SmartNIC can absorb the bursts and process the data before the next wave arrives.

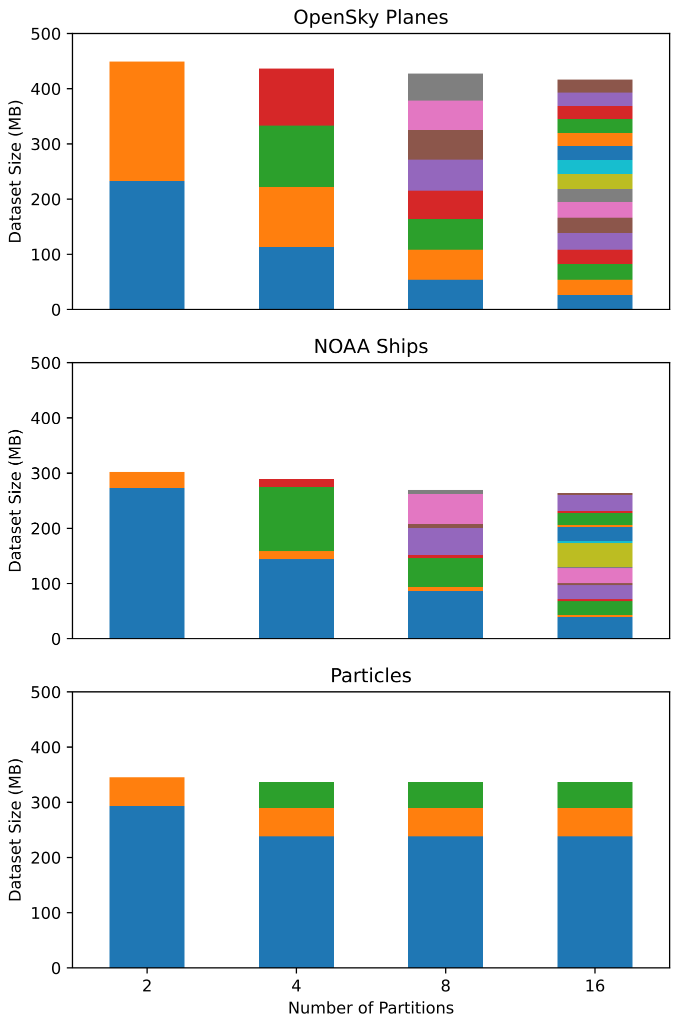


Figure 4.12: Aggregate dataset sizes when varying the number of partitions and compressing with Zstd

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 all three datasets without modification, even though each dataset had different data components and ID bit widths. Although the current version of Arrow does not have all the primitives of a higher-level library such as Pandas [171], it contains adequate primitives to implement a variety of operations.

4.5 Parallel Data Processing

HPC workflows and applications typically employ *composable data service* libraries [210], such as low-level remote direct memory access (RDMA) software, key-value data stores, and lightweight query engines, to establish and tailor data flows among various compute-intensive tasks for modeling, simulation, and analysis (or ModSim). These libraries constitute multiple software components that can be combined to construct application-specific services. Despite the importance of composable data service libraries in workflows, one criticism of current work is that services run on the system's compute nodes, consuming resources that could otherwise be available to ModSim tasks [231, 26, 70, 32]. Moreover, these services can generate interrupts due to their asynchronous execution, potentially introducing unnecessary delays to ModSim tasks that generally rely on periodic synchronizations to proceed. With the advent of SmartNICs, offloading data service workloads to these devices presents an opportunity to isolate services from compute nodes, thereby recuperating valuable resources. In light of the constrained-resource environment of SmartNICs, the immediate research questions arise regarding the performance of running data services on these devices and their capability to optimally harness the device's parallelization whenever possible.

4.5.1 Scientific Computing Workflows

Advanced scientific computing workflows may involve multiple, parallel tools that run on different nodes in an HPC platform at the same time. For example, the workflow depicted in Figure 4.13 first uses a low-fidelity simulation to generate coarse-grained results that deep-learning tools can use to make predictions about the simulation's general behavior. These predictions are then used during a high-fidelity simulation to make better decisions about optimizations such as load balancing. Output results from the high-fidelity simulation are then routed through visualization and I/O staging tools to extract insight and reorganize data before it is archived to disk.

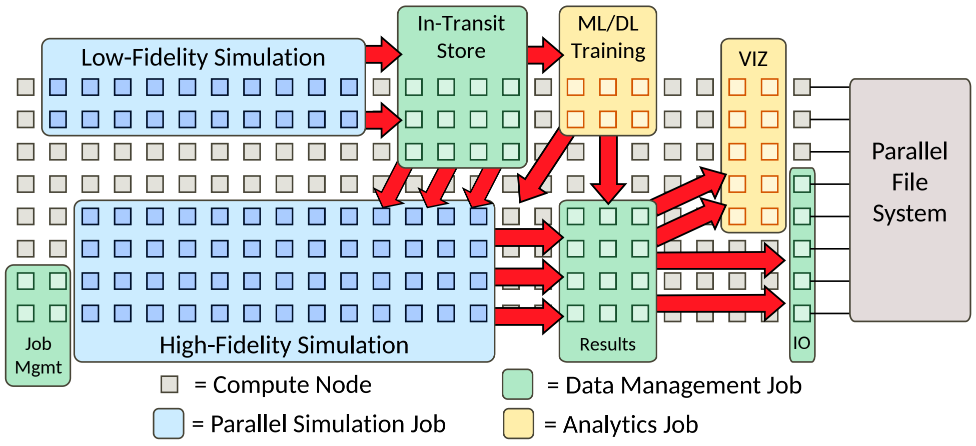


Figure 4.13: A workflow is mapped to HPC compute resources

The traditional means of passing data between workflow tasks has been to write intermediate results to disk [61]. While NVMe storage has dramatically improved performance [24], I/O is still a significant impediment in workflows as data must be transformed from an in-application representation to an archival, on-disk format. Additionally, file I/O libraries can be inconvenient for developers as the interfaces are primarily designed to read and write data rather than process it.

4.5.2 Composable Data Service Libraries

As a means of improving how data flows between workflow tools, research groups have constructed *composable data service* libraries for HPC platforms, including DataSpaces [67], Mochi [210], and Faodel [234]. These 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 to enable data extraction and retrieval customized to the requirements of the data consumer. In addition to low-level RPC, RDMA, and data query 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.

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 [189], RoCE [100], and Cray Aries [10] 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 [23], coupling visualization applications to simulation codes, and insulating users from platform-specific storage

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

issues [224].

Data Processing Library Extensions

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 [175], Kokkos [76], and Apache Arrow.

4.5.3 Service Placement

There are currently three locations in HPC platforms where researchers typically host data management services: in situ, in vitro, and in storage. In-situ approaches place services inside the individual actions of a workflow. This approach reduces the overhead of interacting with a service, but increases build complexity, sacrifices application resources to the service, and introduces fate sharing between the application and the service. In-vitro approaches host services in external nodes within the platform. This approach provides fault isolation but adds extra communication overhead and increases the overall node count for a workflow. Finally, in-storage approaches such as Skyhook [44] embed data services within the platform's storage nodes. However, system policies may forbid users from executing code in these servers for security and reliability reasons.

Over the last decade, hardware vendors have introduced programmable network interface cards or SmartNICs that enable users to place custom computations at the edge of the network fabric. While the original motivation for developing SmartNICs was to allow security researchers to monitor and inspect network flows in real time [138, 213], the need for tighter infrastructure control in cloud computing platforms has driven SmartNIC vendors to create more powerful cards. Vendors such as NVIDIA (formerly Mellanox), Fungible, Chelsio, Intel, and Xilinx have constructed SmartNICs that allow users to embed computations at the network's edge. While some SmartNIC architectures employ FPGAs or ASICs to maximize packet processing performance, most feature a multicore embedded processor that is easier for developers to leverage.

Following the release of the InfiniBand-based BlueField-2 adapter, multiple institutions have deployed HPC platforms that feature SmartNIC-enabled compute nodes [65, 124, 21, 164]. These architectures offer an opportunity to migrate data services into SmartNICs. We see multiple advantages in this approach. First, hosting services in SmartNICs enables services to be placed near applications in an isolated space that does not consume host resources. As such, the host can offload low-priority or asynchronous tasks that might otherwise impede applications. Second, SmartNIC-enabled compute nodes add compute and memory resources to the platform without requiring additional network infrastructure. Finally, vendor roadmaps indicate that future generations of SmartNICs will include processor and accelerator enhancements. While current SmartNIC hardware is sufficient for basic data management tasks, upcoming products may take on greater responsibilities in processing data pipelines.

Figure 4.14 shows an example of offloading a data reorganizing service to multiple SmartNICs. The SmartNICs are logically assigned to multiple levels to expand the data processing capability. When local SmartNICs receive simulation data from the host, they split it into segments and forward the results to the SmartNICs at the subsequent level for more detailed organization or concatenation with supplementary processing. Since SmartNICs now manage in-transit data, it is also possible to map data retrieval services to these devices and use lightweight query engines to facilitate tailored data flows, thereby relieving the query overhead that would otherwise be paid by compute nodes.

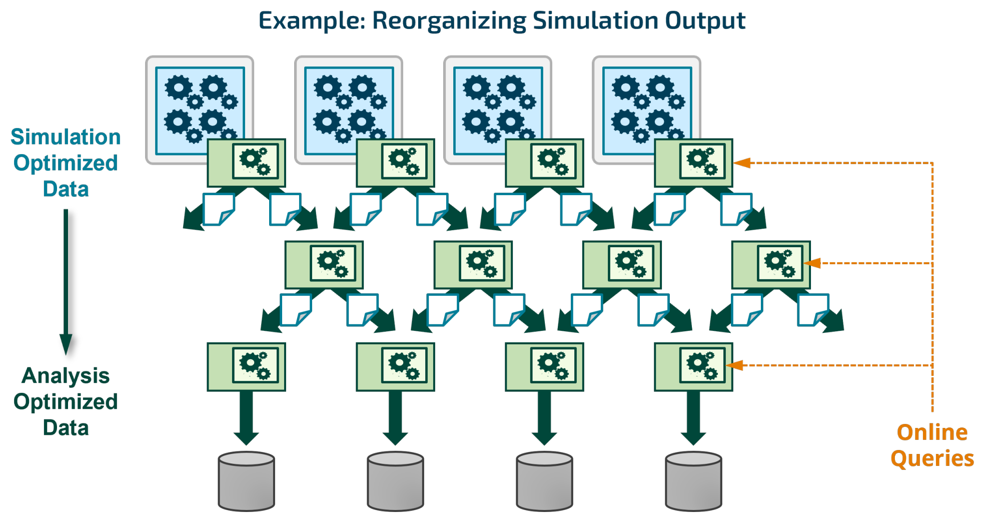

Example: Reorganizing Simulation Output
Simulation Optimized Data
Analysis Optimized Data
Online Queries

Figure 4.14: A data reorganizing service is mapped to levels of SmartNICs: Blue boxes represent simulation tasks that run on compute nodes and generate simulation data. Green boxes represent SmartNICs that receive and process the handover data through the data service pipeline. Online queries can be delivered to the SmartNICs to achieve customized data retrieval needed for the next simulation task.

4.5.4 Performance of Multi-threaded Data Services

We conducted multiple experiments to examine the low-level performance characteristics of the BlueField-2's embedded processors while executing different operations with composable data service libraries. This section presents the results of our experiments, with a specific focus on the multi-threaded performance of bookkeeping using Faodel and data processing using Arrow Acero.

Bookkeeping Overhead on the SmartNIC

Faodel provides a stress-test tool for measuring how quickly a system can perform different tasks. Similar to stress-ng [131], performance numbers lack meaning in isolation, but provide a useful way to compare different architectures. 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 Fig. 4.15, aggregate performance (decreases/increases) as thread counts increase in the collision (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 [151].

Processing Arrow Data

Our Arrow experiments with the BlueField-2 focused on creating queries with inherit parallelism and verifying that execution performance improves as the number of threads increases. For this work, we selected two types of queries that operate on three-dimensional particle data. The first query filters an input dataset

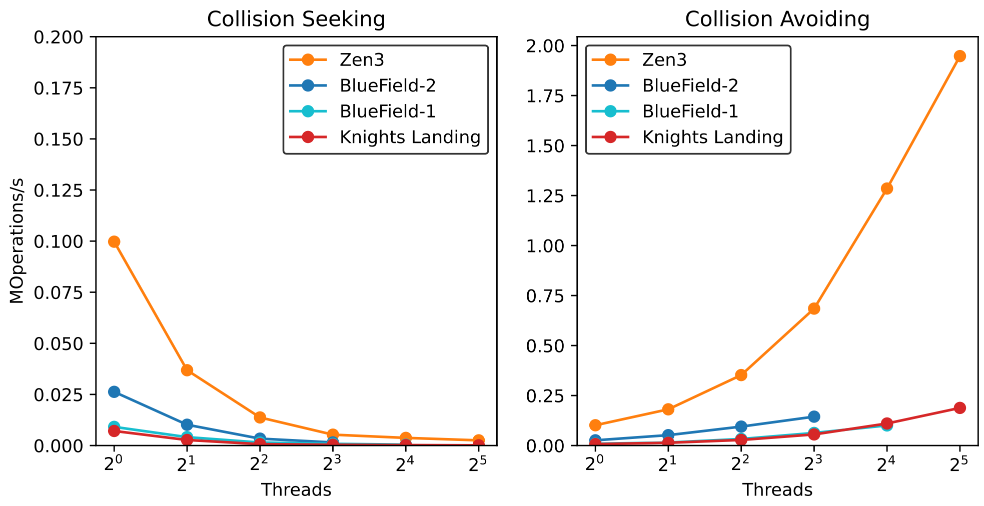


Figure 4.15: Performance in Faodel's LocalKV stress test

based on a bounding box that is picked to select 1/8$^{\text{th}}$ of the original particles. The second query computes the squared magnitude of the velocity of each particle and returns the minimum and maximum values. We created a particle dataset with 8M records and then measured the amount of time required to complete the queries using a variable number of threads on the BlueField-2 and a host system with a total of 32 Xeon E5-2698 processor cores.

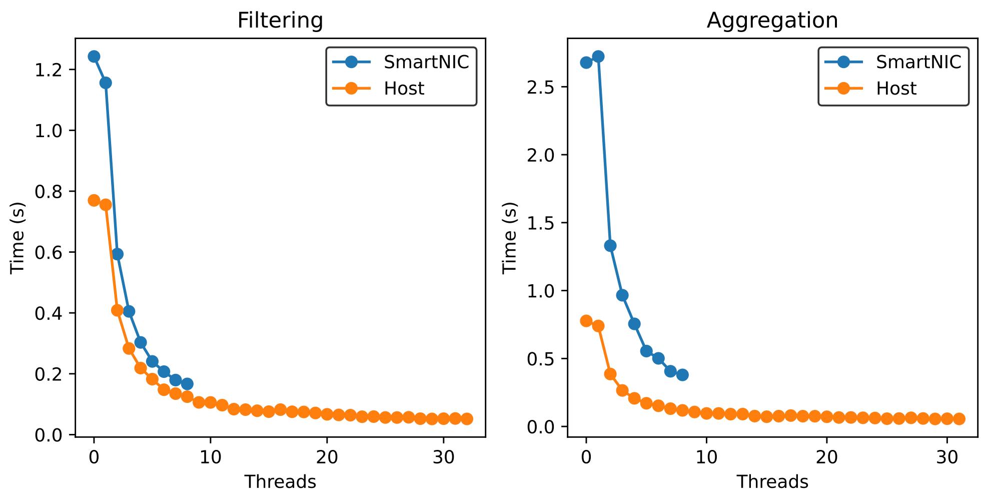


Figure 4.16: Apache Arrow threading performance

The performance results presented in Fig. 4.16 confirm that Apache Arrow can parallelize queries and leverage multiple processor cores to improve performance. Latency drops significantly when moving from 2 to 3 threads for both systems. However, there are only minimal improvements beyond 5 threads. While the host is much more powerful than the embedded processor on the SmartNIC, it is only 37.8% faster than the SmartNIC when using 8 cores for this workload.

4.5.5 Summary

The flexibility of HPC applications to be composed and deployed across various systems provides numerous benefits. Our study found that composable data services exhibit high adaptability in multicore embedded systems and are capable of parallelizing data processing workloads to achieve better performance. Despite their inferior performance on the BlueField-2 when compared to powerful HPC hosts, offloading these services to adjacent yet separated compute environments facilitated by SmartNICs, can potentially alleviate resource contention on HPC hosts, which has long been criticized as an issue.

4.6 Conclusion

In this chapter, we have explored the potential of offloading general operations using techniques similar to MBWU, where we normalized incommensurate performance results with RPWU. We identified several operations as beneficial candidates for offloading that can help alleviate resource contention on hosts. In addition, we evaluated the processing headroom for network functions offloaded to a SmartNIC, and our results showed that network stacks and configurations can substantially influence the performance of offloaded functions. These components, namely RPWU and processing headroom, emerged as potential metrics from our exploration of a SmartNIC, and could complement those introduced in Chapter 3. However, as there may be more undiscovered metrics, relying solely on individual metrics could underestimate the opportunities of a device. Therefore, it is vital to carefully explore hardware in the context of specific data services. Building upon this, we further investigated the offloading potential of data partitioning and parallel data processing in HPC workflows.

It is important to recognize that the processing resources available to a SmartNIC are considerably less than those available to a host. In this study, we observed that the BlueField-2's ARM processors performed approximately an order of magnitude slower than host systems, primarily due to a combination of processor and memory bandwidth limitations. The gap between embedded and server processors is *not unexpected* and is unlikely to change in the foreseeable future.

While SmartNICs are not general-purpose accelerators, there are several scenarios where we expect the hardware to be beneficial to applications. First, SmartNICs are sufficient for performing simple data processing operations that do not involve complex computations. Second, data flows that involve memory contention operations, IPC operations, or encryption can leverage the card's hardware architecture advantages and accelerators to achieve a speedup. Finally, despite their relatively slower performance compared to servers, it may be advantageous to offload low-rate, asynchronous event processing to SmartNICs, due to the disturbances these operations have on other tasks that run on the host. Offloading to SmartNICs offers resource isolation and locality benefits that are attractive for many data services and applications.

Chapter 5

Offloading Strategies

Offloading functions to embedded devices is not a panacea for enhancing system performance. As the previous chapters show, offloading requires strategic navigation of a myriad of trade-offs tailored to the specifics of individual use cases while designing data services and applications to ensure offloading the right functions at the right time. However, accomplishing this is a challenging endeavor, as it entails identifying appropriate workloads and enabling proper distribution of these workloads among multiple embedded devices, addressing the interactions between functions, and scheduling workloads across functions to achieve optimal performance improvements.

This chapter continues to use the example of optimizing particle data flows in the context of HPC scientific workflows with SmartNICs, with a focus on articulating the strategies we have developed for identifying workloads, enabling workload distribution, and addressing function interactions and scheduling. I begin with an in-depth examination of the requirements for offloading data services to SmartNICs in Section 5.1. To address some of these requirements, I developed a data compression library called "Bitar," which employs hardware compression to reduce data serialization overhead, as discussed in Section 5.2. Section 5.3

presents a comprehensive account of the implementation of an embedded processing pipeline that enables data transformation and facilitates data-sifting tasks using SmartNICs. Finally, in Section 5.4, I dive into dynamic offloading and exhibit a mechanism and the corresponding implementation as a decision engine that empowers SmartNICs to make informed decisions regarding the placement of data query workload execution.

5.1 Requirements

The potential evaluation of embedded devices provides crucial insights into the significance of communication and computation to achieve optimal performance for offloaded data services. Based on the evaluation results, we identify the following high-level requirements for devising strategies to offload data services to SmartNICs.

5.1.1 Communication

Composable data services utilize various approaches to effectively organize and reroute large volumes of data to adapt to consumption patterns, leading to dynamic changes in data management criteria across nodes as data propagates through the processing pipeline. For example, a data service may use different strategies to aggregate input data based on its current stage in the pipeline. In early-stage processing, where received data is more prone to shuffling as generated by ModSim tasks, aggregating this data with lower precision can reduce computational overhead. Conversely, for processing running at later stages, higher precision may be utilized for aggregation, as the received data is mostly processed and thus, the workload can be efficiently reduced even with parameters of higher precision. Offloading these dynamically changing workloads to SmartNICs is challenging, as it requires a mechanism to specify data processing workloads based on the runtime parameters in diverse scenarios. Additionally, to enable the offloading of customized data retrieval workloads to these devices, it is essential to have a flexible means of defining workloads by the request initiators, allowing for the delivery of workloads through the network and their execution on SmartNICs without incurring significant preprocessing overhead. In order to address these demands, a workload definition specification that encompasses the following features is required and is expected to support:

- **data inputs of different types.** For example, some data may be in string type, and some are in integer type. Simulation particle data typically contains arrays of data, so the specification should also support defining data of different array types;

- **different operations on data input.** Data services can utilize various operations to transform the input to be more readily consumable. This transformation process may involve applying operations such as arithmetic, logical, comparison, and aggregation functions to the data;

- **user-defined functions (UDFs) to facilitate different use cases.** User-defined functions offer valuable flexibility in the data processing. These functions are not necessarily resource-intensive and can be executed efficiently even on resource-constrained devices like SmartNICs. An example of UDF may be to apply a ceiling to a value only if it falls below a certain threshold;

- **constructing composable definitions.** Data services often involve a series of operations, typically chained in sequence. For instance, a workflow might require executing a comparison operation after an arithmetic one, or conducting an aggregation operation following a projection and a series of logical operations;

- **referring to local or remote data inputs.** Workload definitions often require to reference data located on remote nodes or disparate SmartNICs. This scenario is particularly prevalent when one node delegates a particular task to another, yet the requisite data for the said task is not under the ownership of the delegating node;

- **constructing definitions with high-level interfaces.** A workload definition may contain hierarchical operations, which can be tedious to construct and challenging to manage using low-level expressions. Providing high-level interfaces to abstract away the complexity of breaking it down into corresponding operations and applying conversion and canonicalization when necessary aids in building data services, enabling service developers to express their desired logic in a more intuitive and straightforward manner. This can be especially important when working with large datasets or intricate workflows that require multiple steps to achieve the desired result;

- **constructing serializable definitions.** This is because instructing a SmartNIC to perform a specific task requires sending a workload definition over the network.

The second facet of the communication requirements lies in the overhead associated with the network transfer of data generated by offloaded workloads. As discussed in Section 4.3, data transfer using the traditional Linux kernel TCP/IP stack can lead to significant overhead, especially on systems with limited computing resources. Additionally, when it comes to data marshalling or serialization, achieving efficiency is crucial to allow the majority of SmartNICs' computational resources to be dedicated to data processing. Therefore, the data transfer process should possess the capability to:

- **leverage low-overhead or accelerated data transports (e.g., RDMA).** HPC platforms typically employ RDMA for communication between different services. Being able to use these transports can improve the performance of data processing pipelines;

- **serialize data into a concise on-the-wire format.** As a counterexample, although JSON is a widely-used data format that represents structured data in a string format suitable for network transmission, it is more optimized for readability than binary size, making it less suitable for data-intensive services. Formats like Google Protobuf [25] and FlatBuffers [143], on the other hand, emphasize reducing the size of serialized data, making them more suitable for such contexts;

- **serialize and deserialize data efficiently.** Data serialization can consume significant computing resources on embedded systems, even those equipped with multicore processors and multiple gigabytes of memory. This is due to the segmented memory copying involved in processing small-sized data items that require rearrangement on a contiguous memory space. The memory space may even be reallocated multiple times due to the uncertainty of its final size. To minimize the impact of data serialization on system performance, it is crucial to optimize the process using efficient algorithms and data structures.

5.1.2 Computation

Given the limited computing resources on SmartNICs, the ability to explore parallelism when possible for data service workloads is crucial. Although workload definitions may not provide sufficient information to guide parallelization, the workload execution engine needs to have the capability to partition a workload into segments and delegate them to distinct threads by harnessing the available computing power on the system. However, care should be taken to avoid over-parallelization that could potentially overload the system. Overall, the execution engine is expected to be capable of:

- **formulating execution plans to facilitate parallelism.** Workload definitions are not necessary to specify the parts that can be executed in parallel. In fact, the execution engine can use algorithms, such as topological sorting, to identify independent operations within a workload definition. These independent operations can subsequently be scheduled to run in parallel. Ideally, the complexity of this process, including identifying and locating the necessary data inputs, should be managed internally by the execution engine, keeping it abstracted from workload definitions;
- **leveraging local resources to maximize execution efficiency.** Given that modern SmartNICs are equipped with multicore computing capabilities to manage advanced data services, the execution engine should, once the workload execution plan is established, make the utmost use of the SmartNIC's available computational power.

In certain domain-specific areas, security considerations may take precedence and therefore need to be taken care of and included as part of the requirements. HPC systems are often administered with firewalls and authentication controls and adhere to a reserve-before-use model to ensure secure access. As our current research is primarily focused on optimizing HPC workflows, we have consciously chosen to exclude these security considerations from our current scope of study.

5.2 Bitar: Optimizing Data Compression for Serialization

Data compression plays a crucial role in data serialization and is particularly significant for data-intensive applications, as it minimizes the amount of data that needs to be transmitted over networks, cached in memory, and stored on disk. Most large-scale data I/O libraries, such as Avro [241], Parquet, ORC [235], and Arrow IPC, come equipped with built-in support for a myriad of compression codecs. As such, any application that processes this data must be capable of decompressing and compressing the data in a manner that complies with the library's data format.

However, it is well-known that data compression is a computationally intensive task [149, 81, 1, 37] for general-purpose host processors, not to mention that if it is handled only by embedded processors. The BlueField-2 SmartNIC is equipped with a hardware compression accelerator that supports the DEFLATE algorithm [64]. DEFLATE is widely used and is a key part of standards such as PNG [28], HTTP [173], TLS [111], and SSH [267]. More importantly, this algorithm is part of the zlib compression family, which enables the use of hardware compression as needed, with the option to decompress using software in applications and vice versa. This interoperability is critical because it enables data processing tasks to be "pushed down" and offloaded from the host or "pushed back" when the accelerator becomes saturated with work. The BlueField-2's compression hardware is currently accessed through the Data Plane Development Kit (DPDK) [42, 83], 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 streams of data packets in an efficient manner, with DMA hardware facilitating the movement of data between the accelerator and memory.

5.2.1 Hardware Compression for In-transit Data

As demonstrated in Section 4.4, converting between on-the-wire and in-memory formats is an important and time-consuming task for systems that process in-transit tabular data. Given that the Bluefield-2 SmartNIC provides a compression accelerator and multiple cores that Apache Arrow can leverage, it is worthwhile to explore the different compression options that are available for packing and unpacking data. We conducted three experiments to answer the following three questions: (1) Is the compute overhead caused by software-based compression significant enough to justify offloading the (de)compression to hardware accelerators? (2) How does the throughput performance of hardware-based compression compare with software-based compression in a threaded environment? (3) Does the compression ratio change between the hardware- and software-based methods?

Hardware Compression Challenges

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 slice input and output buffers into packet-sized segments and then generate a packet that connects 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.

5.2.2 Compression Interfaces

To simplify accessing the compression hardware for data compression, we implemented the *Bitar* [154] 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. Notably, Bitar is designed to function without the need for root privileges, a feature not commonly seen 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.

5.2.3 Performance Advantages

All experiments in this section were carried out on a CloudLab [73] host with two AMD EPYC 7542 CPUs (a total of 64 cores), 512 GB of DDR4 memory, and a BlueField-2 SmartNIC connected with PCIe 4.0 x16 lanes. Each experiment was run on all three reference datasets (see Section 4.4.3) 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.

Software Compression Overhead with 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 measuring the time for a single thread to pack and unpack 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.

Timing results for the "Particles" dataset (see Figure 5.1) indicate that serial-

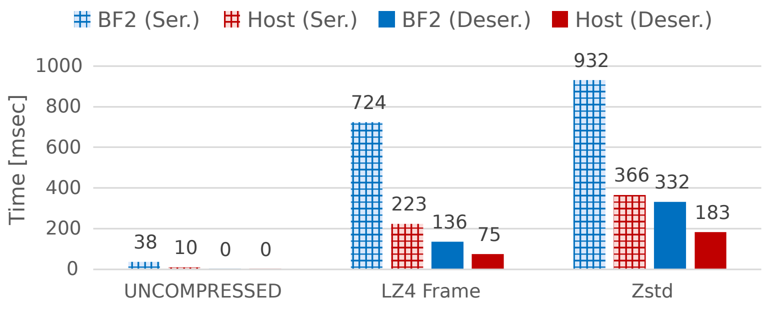


Figure 5.1: Single-thread (de)serialization time with different compression codecs

ization 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.

Serialization 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 the (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.

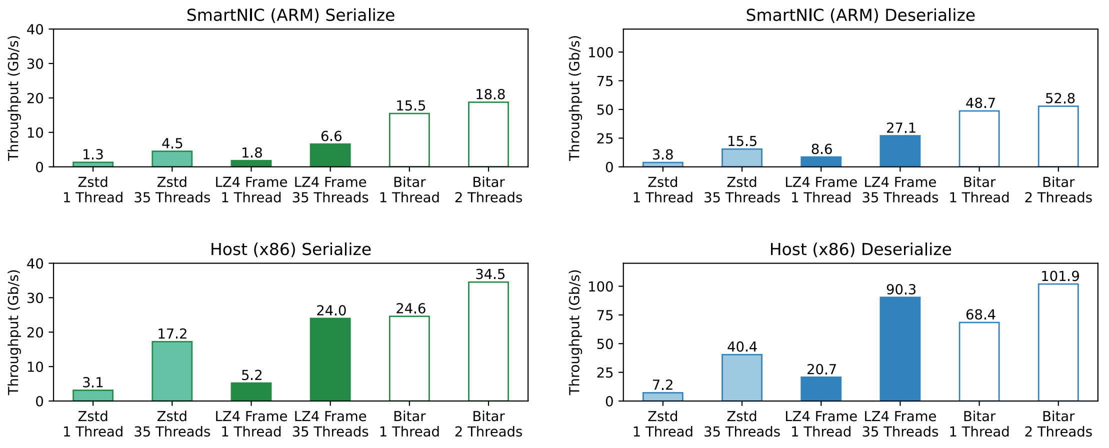


Figure 5.2: (De)serialization throughput with different compression codecs and degrees of parallelism

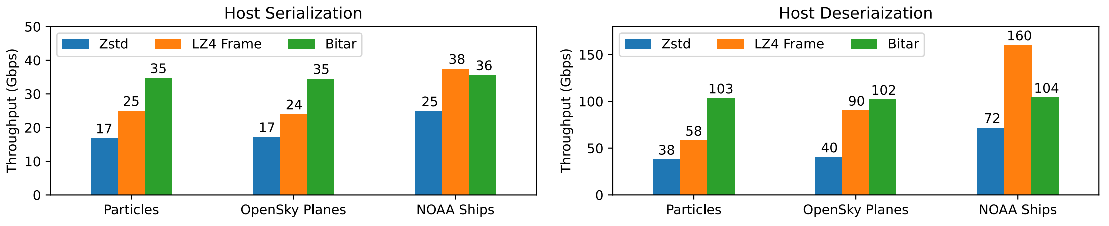


Figure 5.3: The maximum throughput performance on the host for all three datasets

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. 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, depending on the codec and dataset used, serialization with Bitar can achieve between 4.6-8.6x higher throughput than serialization with software-based compressions. For compression with multiple threads, the use of Bitar can speed up the serialization throughput on the host by 1-2x. 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 that has maxed out the SmartNIC's network bandwidth, the marginal benefit of the additional (de)compression throughput above the NIC's network capability achieved by the resource-intensive software-based approach is minimal considering the limited local storage support on the NIC, especially for tasks focusing on transferring in-transit data. 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.

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

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

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.

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

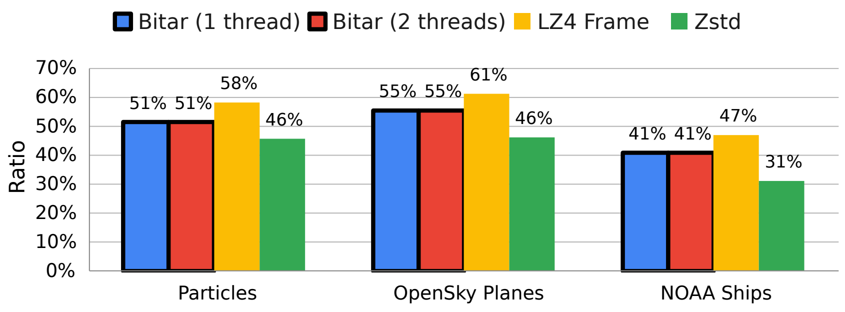


Figure 5.4: Compression ratios under different compression approaches. Thick black borders indicate hardware (de)compression results.

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 [37, 260]. 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.

5.2.4 Summary

While current-generation SmartNICs are slower at processing data than servers, they can perform fundamental data-sifting tasks that are commonly required by different workflows. The compression hardware is particularly appealing for this work, as it allows users to efficiently unpack, process, and repack in-transit data products. However, the current interface for accessing the hardware is challenging to leverage and an obstacle for developers. We present Bitar as a reusable library for simplifying compression on the BlueField-2 cards.

Apache Arrow provides a data model and a collection of operators that are particularly well-suited for processing data on embedded devices that are part of a eusocial processing environment. Arrow's tabular notation allowed us to devise a general framework for storing and processing particle data that did not need to be adjusted when switching between datasets. We note that other types of data may not map to a tabular form as elegantly.

There are multiple paths forward from this work. Having completed the on-card processing work, we will transition to network tasks related to distributing data between SmartNICs and coordinating resource utilization across a distributed system. Based on the TCP bottlenecks observed in previous work, it is imperative that these operations take place with RDMA primitives. For the compression work, Arrow will need minor adjustments to allow general users to take advantage of Bitar. These adjustments include modifying Arrow's IPC format to support the DEFLATE codec, incorporating Bitar into Arrow's list of approved third-party libraries, and updating Arrow to route data through Bitar when appropriate. These changes would allow for finer-grained access to Arrow data than our current work, as the compression would be applied to individual columns instead of serialized tables.

5.3 Embedded Processing Pipeline

The criticism of running composable data services on the compute nodes of an HPC platform, occupying simulation tasks' resources, catalyzes to seek answers to two research questions: *How should we construct software to implement services on these devices? Can distributed services perform useful work on SmartNICs?* In this section, we focus on answering these questions by defining requirements to enable interoperability and data service pipelining with SmartNICs and discussing a software-stack prototype and its performance on current hardware. Finally, we present a case study in which a distributed particle-sifting service runs on a 100-node HPC cluster that features BlueField-2 SmartNICs.

5.3.1 SmartNIC Software Stack for Data Services

To efficiently bridge the gap between resource-rich hosts and resource-constrained SmartNICs running data management services, we need a software stack that can orchestrate these services among hosts and SmartNICs while minimizing the impact on applications and maximizing the reuse of existing software. This software stack must address communication issues (e.g., How do applications interact with remote SmartNICs over the network? How can SmartNICs work collectively?) as well as computational issues (e.g., How are computations defined and executed by services? How can the system be extended with new operations?). In this section, we define our list of requirements for this software stack and discuss how a suitable environment for hosting services in SmartNICs can be constructed through the combination of the Faodel and Apache Arrow libraries.

Service Requirements

Based on our experiences with workflow environments, we identify five basic requirements we expect from an environment where services execute in embedded devices (Figure 5.5). (1) Each service endpoint requires a unique identity that other entities in the platform can reference and access via efficient communication mechanisms. (2) Users must be able to control the mapping of services to physical resources at runtime and group several devices together in a way that allows the devices to work together. (3) Users must be able to trigger service computations locally and remotely. (4) The stack should present a flexible data-processing API that is robust and has community acceptance. (5) Data-parallel computations must automatically exploit available CPU resources.

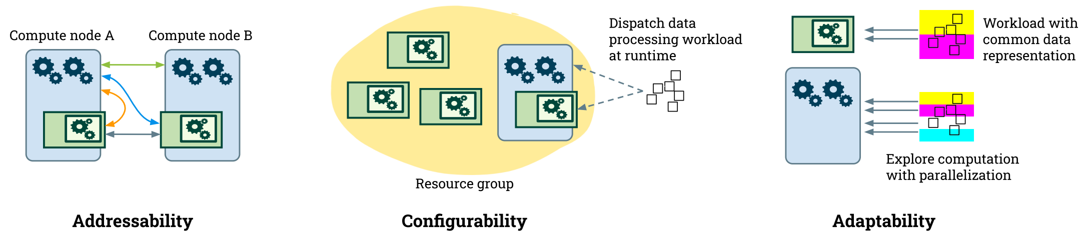

Compute node A
Compute node B
Addressability
Dispatch data processing workload at runtime
Resource group
Configurability
Workload with common data representation
Explore computation with parallelization
Adaptability

Figure 5.5: Requirements on the environment for embedded service execution

Communication: Faodel

We selected the Faodel library to serve as a foundation for the communication portion of our software stack prototype because it is open source, written in C++, has support for both x86 and Arm, and includes existing primitives for working with endpoints scattered about a platform. Specific details about how Faodel fulfills our requirements follow.

- **System-wide Accessibility:** Faodel assigns a unique identifier to each endpoint that is used to establish both HTTP and RDMA communication. Faodel's Kelpie library provides an easy-to-use mechanism for safely transferring key-labeled objects between endpoints using RDMA mechanisms. Users can put, get, list, and delete objects on local or remote endpoints.

- **Resource Pools:** Kelpie uses a simple pool abstraction for grouping multiple endpoints together for related work. A pool contains a list of endpoint members and a distribution policy that maps key labels to pool members. By supplying different pool configurations at start time, users can change the behavior of their data flows.

- **Dispatching Computations:** While Kelpie is agnostic about data formats and computations, it provides two methods for invoking computations at endpoints. First, an endpoint may run its own main loop that periodically inspects the state and reacts to changes. Second, users may invoke computations on objects at remote endpoints through user-defined functions.

Computation: Apache Arrow

Apache Arrow was selected to implement the data computations in this work because it provides a rich set of primitives for storing and querying tabular data, is open-source C++, and is actively developed by a large community. Specific aspects of Arrow that meet our requirements follow.

- **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.
- **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 [53] 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.

Integration Challenges and Requirements

We faced two integration concerns while constructing our software stack for data management services. First, small portions of Faodel and Arrow target processor-specific features. While both libraries had previously been ported to x86 and Arm processors, extensive testing was required to ensure data handoffs between the two architectures functioned correctly. The second integration challenge involved finding a means of transporting Arrow data using Faodel's native objects. Our current solution is to use Arrow's IPC serialization mechanisms to embed one or more tables in a Faodel object. A wrapper class was developed to convert between an in-memory Arrow table and the payload section of a Faodel object.

Additionally, 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.

5.3.2 Distributed Particle Sifting

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 Fig. 5.6, 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 splits all of its accumulated data into smaller objects based on bits in each record's particle ID field [158], and 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 info is embedded in a separate portion of the key to avoid collisions with the data from other PEs.

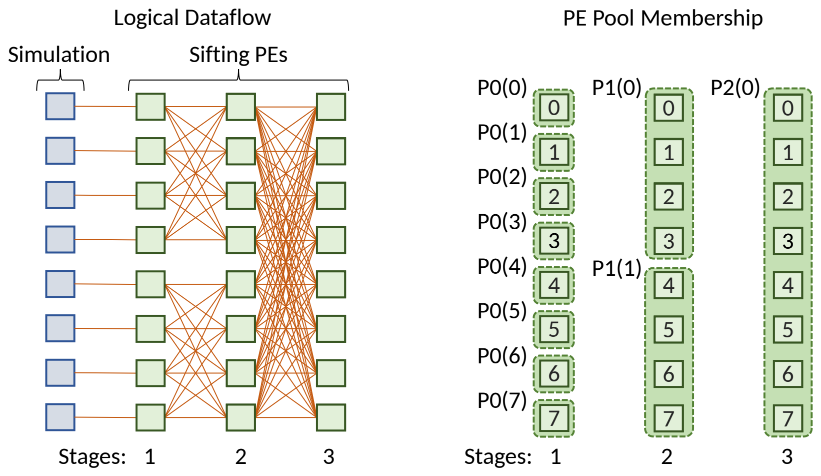


Figure 5.6: Dataflow and placement for sifting particle data

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 notation 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 Fig. 5.6 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.

Injection Overhead

The first step in reorganizing the particle data is for each host in the simulation to sample its current data, convert it to serialized Arrow data, and then transfer it to the local SmartNIC. We constructed a benchmark to quantify injection overheads and varied the transfer size from 1M–64M particles (37MB–2.4GB). As presented in Fig. 5.7, transferring the data to the card through Faodel’s primitives consumed 81% of the overall injection time. For 64M particles, we observed an overall transfer rate of 1.32GB/s.

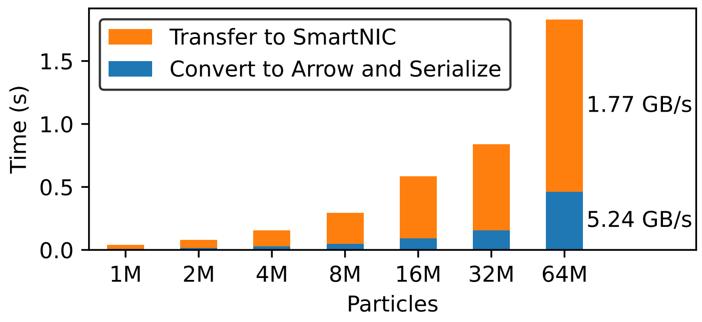


Figure 5.7: Data preparation and injection overhead

Impulse Response

To explore sifting performance for different configurations, we constructed an impulse response benchmark that injects uniform data to each of stage 1’s PEs and then measures the amount of time required for all compaction events to take place in a synchronous manner. We varied the number of splits performed by each

PE and selected the minimum number of stages that would be required to fully distribute data across 100 SmartNICs.

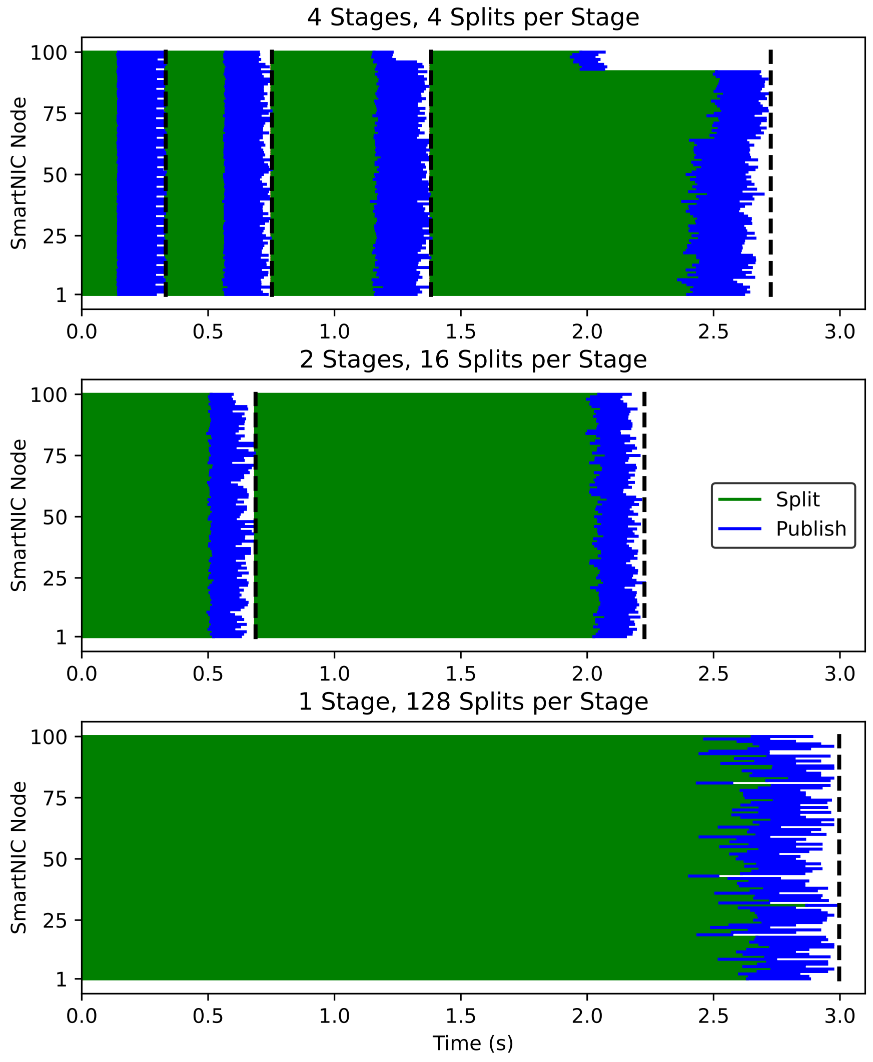


Figure 5.8: SmartNIC sifting time for 100M particles

Fig. 5.8 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.

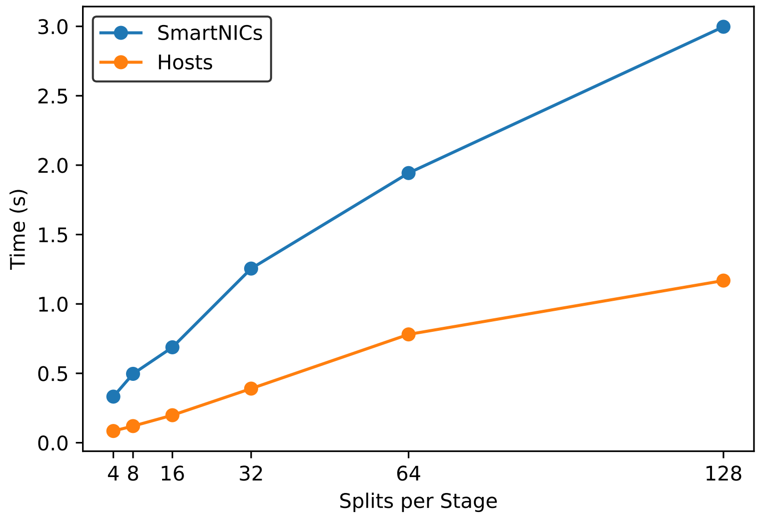


Figure 5.9: First stage overhead for 100M particles

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 Fig. 5.9, the 32-core host processors were roughly four times faster than the 8-core Arm processors.

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 Fig. 5.10. Performance scaled linearly in both cases. The host systems were again roughly four times faster than the SmartNICs.

Discussion

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

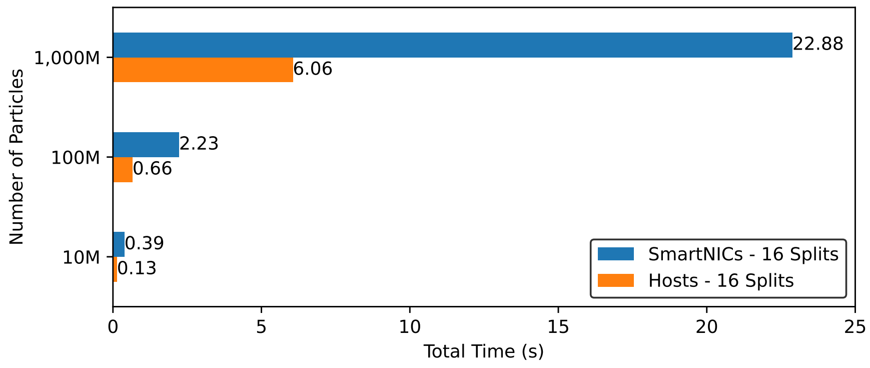


Figure 5.10: Total sifting time for different input datasets

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.

The overhead of serializing data and injecting data to the SmartNIC was substantially higher than expected and a significant opportunity for improvement. Future work will focus on optimizing the transfer path between the host and its local SmartNIC. NVIDIA's recent DOCA library [36] includes host-to-card DMA transfer software that is expected to remedy this problem. It is also likely that converting, serializing, and injecting data in smaller fragments will help pipeline the process.

This case study demonstrates that Faodel and Arrow can provide a useful environment for hosting data management services on a collection 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 rebuilding the software.

5.3.3 Summary

SmartNICs offer a new location in HPC architectures for hosting data management services. Constructing a software stack that can support these services involves developing a communication plane that allows different endpoints in the platform to interact with the SmartNIC, and data processing software that can efficiently dispatch computations on datasets that adhere to a well-defined data model. Future work with SmartNICs must create a stronger coupling between the host and its local SmartNIC, and take advantage of vendor-specific features for accelerating performance.

5.4 Dynamic Offloading

Enabling data services and software stacks to run within distributed SmartNICs only unilaterally tackles the offloading challenges. Given the constrained resource environment on SmartNICs, running data services without a dynamic offloading mechanism that allows SmartNICs to decide and "push back" operations may result in suboptimal performance of the services and potentially propagating performance lags to subsequent simulation tasks. For example, when a SmartNIC is loaded with operations for data sifting, the concurrent demands for customized data streams from the SmartNIC may be impeded due to the insufficient availability of parallel processing cores or memory resources for effective data processing. To address this problem, it is essential to have an efficient mechanism that SmartNICs can execute in situ and react to workload requests based on the runtime resource context and the performance requirements of the respective workflows.

Retrieving customized data streams is a standard requirement in HPC simulation to allow different simulation tasks to consume only the relevant data parts.

For example, a simulation task may be interested only in particle data collected from events occurring within a specific region for analysis. Given that in-transit data is hosted by SmartNICs, the dynamic offloading of data query workloads presents an ideal use case that relieves performance concerns by potentially reallocating workloads to hosts, while continuing to explore the data locality benefits from running within the SmartNICs.

In this section, we begin by exploring the classes of push-back strategies that enable dynamic offloading. Subsequently, we delve into the requirements that extend beyond the implementation of one-way offloading. Finally, we provide a comprehensive account of our experience building a dynamic offloading decision engine and evaluating its performance. The statistics generated by the decision engine can be harnessed by schedulers based on job sizes or predictive models, therefore rendering it a highly versatile component that can offer substantial benefits to a wide range of data services.

5.4.1 Push-back Strategies

Different push-back strategies reflect different goals for offloading workloads. Similar to the various congestion control strategies used in the network domain to optimize packet data workloads, push-back strategies for data service workloads can be categorized into two classes.

The first class of strategies is *utilization-oriented*. In the network congestion control arena, Cubic TCP [103] is an example of this strategy class that efficiently achieves high bandwidth utilization in the face of high latency. Previous research [161] has explored this direction for offloading distributed applications, such as real-time data analytics engines and replicated key-value stores, onto SmartNICs with the goal of maximizing NIC compute utilization.

The second class of strategies is *performance-oriented*. TCP Vegas [30], among other TCP congestion control algorithms, exemplifies this approach by prioritizing packet delay over packet loss as a signal to determine when packets should be rejected. These strategies may emphasize a particular performance metric over others, such as prioritizing low latency at the expense of reduced throughput performance by limiting the queue size. Similarly, a performance-oriented strategy for data service workloads may push back workloads to hosts based on latency reasons, despite the presence of available resources on the SmartNIC for workload execution. We are interested in implementing a strategy that optimizes the latency of query workloads, as simulation tasks tend to be more sensitive to the latency performance in data retrieval than the resource utilization of individual SmartNICs.

5.4.2 Additional Requirements

Dynamically offloading workloads that depend on in-transit data in HPC differs from assigning workloads to serverless functions across nodes in cloud computing. First, HPC simulations are subject to crashes, revisions, and periodic modifications. Therefore, accessing current data takes precedence over historical data regarding data availability. Second, the sheer volume of the in-transit data makes maintaining up-to-date replications on external resources unaffordable. Third, the limited computing and memory resources available on SmartNICs prohibit the management of data replications within the data service pipeline, as it can hamper the SmartNICs' ability to provide offloaded data services. As such, for a given data service workload, the placement of the workload execution is binary: it can either run on the SmartNIC where the required data resides or on a host if pushing it back is deemed necessary.

Moreover, while previous work on TCP congestion control exhibits similarities with dynamic offloading for data service workloads, TCP congestion control typically adjusts to workload patterns based on packet sizes, as evidenced by constantly updating the congestion window size [9]. However, the job size of a data service workload, such as a data query, is not immediately obvious.

Additionally, to enable workloads to be dynamic offloading, the execution engine is required to be architecture-agnostic and capable of running on systems of different architectures (e.g., x86 and ARM) to produce consistent results when given the same workload definitions and data inputs. This is not possible if architecture-dependent features are exposed to the definition or input, causing the interfaces of the engine to rely on those features. Finally, given the streaming nature of the data on SmartNICs, it is important for the decision engine to adapt to different workloads during runtime without compromising the efficiency and reliability of the overall system. We expect that this engine is capable of:

- **determining which workloads to push back to meet the current workflow's requirements.** For example, deciding not to push back a workload may improve the system's processing throughput but cause a negative impact on the execution latency of the current workload. Therefore, deciding when to push back is policy relevant as different policies may lead to conflicting decision results;

- **estimating the job size of a workload.** Many job scheduler algorithms (e.g., shortest remaining time first and credit-based fair queuing) rely on the assumption that job size is either known or easily accessible when the job comes. To effectively utilize schedulers that have been developed through years of effort in our query workload dynamic offloading, it is essential to convert the workload definition received by the decision engine into a job size representation that can be used as input by the scheduler.

- **generating micro-level decisions for workloads defined by composable definitions.** A workload definition may contain complex operations that are compute-intensive for embedded processing and operations that are IO-intensive but cheap in computation. Being able to extract the benefits of offloading such a workload from hosts at a finer granularity and taking advantage of data locality amplifies the advantages of offloading to embedded systems;

- **working with dynamic workloads and data inputs.** Given the streaming nature of data services running on SmartNICs, it is imperative that the decision engine is capable of processing various inputs in real time;

- **being accurate enough to justify the value of offloading services to embedded systems.** Note that the accuracy improvements may not have a linear relationship with increases in offloading value. We should carefully balance efforts to improve accuracy toward the specific goal of offloading and the potential impacts of the improvement;

- **being efficient in initializing and running.** Decision engines may require a warm-up period to collect statistics or adapt to current workload patterns before functioning correctly. The initialization process should be efficient and impose trivial overhead on the running services. Once complete, the decision-making process regarding 'push-back-or-not' should rely solely on current statistics with minimal computing and memory overheads;

- **mitigating the push-back storm.** This could happen when multiple SmartNICs push back workloads to the same host, causing a reduction in performance compared to when the workloads are processed by these Smart-NICs individually.

5.4.3 Query Representations

Database management systems (DBMS) follow a multi-layered approach to process queries, wherein a query is translated into different representations at various system layers [94, 63]. Although a SQL statement is precise, parsing, and converting it into a logical plan, which specifies the data sources and operators to apply, demands significant computational resources. The logical plan is subsequently transformed into a physical plan that considers the data placement and local resource capacity to optimize query execution.

Leveraging composable data services in HPC simulation offers the advantage of customizing the placement of different services to enhance workflow performance. Considering the constrained resource availability on a SmartNIC, it is prudent to leave functions with substantial overhead, such as query parsing, on more capable systems and place only the core execution function that consumes logical plans on SmartNICs. Therefore, having serializable logical query plans as workload definitions becomes necessary to enable the dynamic offloading of query workloads.

Logical Query Declaration with Arrow Acero

Arrow Acero is a data computation execution engine that supports a variety of data processing operations, including filtering, sorting, aggregation, and custom user-defined functions. Importantly, it also provides complete semantics for constructing composable logical query declarations in C++. For instance, we can define an aggregate declaration on top of a projection declaration. Listing 3 and 4 shows how an example SQL query for aggregation maps to the corresponding composable declaration in Arrow Acero.

```
select count(id) as column_hash_count_id from particles
    where x >= 0.3 and y < 0.42 and z <= 0.68
    group by particle_id
```

Listing 3: A SQL query for counting the number of records of particles captured within a certain region

Intermediate Query Representation with Substrait

Substrait is an open-source project[1] that aims to create a cross-language specification for data computing operations. It focuses on the semantics of each operation and provides a consistent way to describe them. The goal of this project is not to replace SQL but to work alongside it to provide capabilities that SQL lacks, such as a standard and open format for query plans. By leveraging substrait, it becomes possible to convert a logical query plan into a binary representation as a substrait plan in either ProtoBuf or JSON format. This plan can then be sent over the network to the workload execution engine running on a SmartNIC. For any remaining operations that the SmartNIC pushed back, a portion of the substrait plan can be piggybacked with the intermediate result and forwarded to the host where the original data retrieval request was initiated. Upon receipt, the host can detect the presence of the substrait plan, execute it on the data, and complete the push-back operation. Listing 5 presents a snippet of a substrait plan in JSON.

[1] `https://substrait.io/`

```
auto declaration = arrow::compute::Declaration::Sequence({

  {"named_table",
      arrow::compute::NamedTableNodeOptions{{"particles"},
          table_schema}},

  {"filter",
      arrow::compute::FilterNodeOptions{arrow::compute::call(
          "and_kleene",
          {arrow::compute::call(
               "and_kleene",
               {arrow::compute::call("greater_equal",
                  {field_x, arrow::compute::literal(0.3)}),
                arrow::compute::call("less",
                  {field_y, arrow::compute::literal(0.42)})}),
           arrow::compute::call("less_equal",
              {field_z, arrow::compute::literal(0.68)})})}},

  {"aggregate", arrow::compute::AggregateNodeOptions{
      {{"hash_count", "id", "column_hash_count_id"}},
      {"particle_id"}}}

});
```

Listing 4: The corresponding Arrow Acero declaration of the above SQL query

```
{
  ...
  "extensions": [
    {
      "extensionFunction": {
        "functionAnchor": 0, "name": "count"
      }
    }
  ],
  "relations": [
    {
      "root": {
        "input": {
          "aggregate": {
            "input": {
              "read": {
                "baseSchema": {
                  "names": [
                    "id", "time", "particle_id",
                    "x", "y", "z", "vx", "vy", "vz"
                  ],
                  ...
                },
                "namedTable": { "names": ["particles"] }
              }
            },
            "measures": [
              {
                "measure": {
                  "functionReference": 0,
                  "phase": "AGGREGATION_PHASE_INITIAL_TO_RESULT",
                  "invocation": "AGGREGATION_INVOCATION_ALL"
                }
              }
            ]
          }
    ...
}
```

Listing 5: A substrait plan snippet in JSON represents the same work as in SQL `select count(*) from particles`. Ellipses indicate omitted content.

5.4.4 Decision Engine Scheduling

We assume that the massive amount of particle data generated by HPC simulations is streamed to the data pipeline managed by SmartNICs. To maximize memory utilization, each SmartNIC is required to manage gigabytes of data in its service lifespan. This substantial amount of data necessitates that the execution of each query maximizes the utilization of SmartNIC's processor cores by leveraging the parallelism provided by the Arrow Acero execution engine. Therefore, although multiple queries can possibly be queued up locally, a SmartNIC can execute only a single query at a time.

On the other hand, simply queuing queries in a decision engine for scheduling without dissecting the query and estimating the job size is insufficient, as highlighted in Section 5.4.2. Most data storage and network schedulers rely on a predefined unit of work to determine the queue length, such as the "block" unit of data request for storage schedulers or the "packet" unit of network request for network schedulers. While the size of a unit of work is generally assumed to remain constant throughout a given workload, a query by itself does not provide enough insight into the job size. The amount of work required by a query is also dependent on the dataset size associated with the query. Therefore, before applying well-known schedulers, whether they are reactive or proactive, it is necessary to evaluate the job size of the query.

Reactive scheduling: A scheduler of this type can incrementally revise its internal state in response to the current workload pattern. Reactive scheduling is often used in dynamic environments where predicting what will happen next is difficult. One example of reactive scheduling is the deficit round-robin algorithm [219] used for scheduling tasks in real-time systems or packets in network devices. Because of its reactive nature, the optimal scheduling decision can be delayed for individual requests. Previous research [161] has explored implementing this type of scheduling in SmartNICs for offloading distributed applications.

Predictive scheduling: This type of scheduling tries to forecast the cost of a workload request by analyzing the current context (e.g., historical data and predictive models) and proactively making scheduling decisions accordingly. This approach is commonly employed by database query optimizers [114] to determine the most effective query plan to execute a given query.

As a means of evaluating the efficiency of SmartNICs in making offloading decisions using predictive scheduling, the following section discusses the components of our decision engine and prediction performance.

5.4.5 The Decision Engine

Cost Analysis

In order to decide whether to push back a query workload, the decision engine must consider several factors regarding its time consumption. For any query workload that is executed by a SmartNIC when offloaded or a host when pushed back, the time it takes for the query initiator to receive the final result is conceptually broken down and depicted in Figure 5.11.

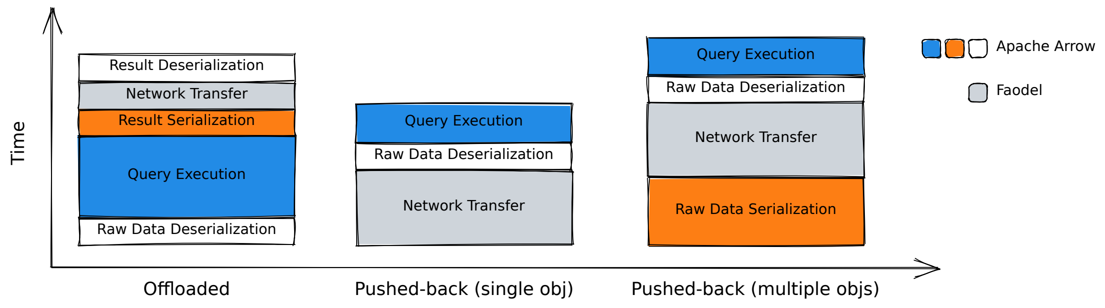


Figure 5.11: Conceptual time breakdown for offloaded and pushed-back cases

Serialization Time: SmartNICs store particle data objects in Arrow IPC streaming format, requiring deserialization before query execution. Arrow IPC's efficient data structure allows for zero-copy reconstruction of Arrow tables from IPC buffers by simply establishing reference pointers to the data source with a single thread. Figure 5.12 (left) demonstrates that deserialization performance is independent of table size, remaining nearly constant within each platform.

After query execution, the resulting table is required to be serialized into an IPC buffer before it is transmitted back to the request initiator. However, as depicted in Figure 5.12 (right), the time taken for serialization depends on the table size.

Figure 5.11 presents a scenario under the third bin where queries rely on multiple data objects necessitating their merging prior to network transfer. This necessity arises because, despite each data object being managed as a serialized object on SmartNICs, the simultaneous transmission of multiple objects could incur significant overhead from memory and network management. This is particularly relevant for efficient network transports like RDMA, where their setup and completion costs favor larger transfers, thus further emphasizing the need for merging. In practice, during data processing, it is common for SmartNICs to handle a greater number of smaller data objects. This can be exemplified by the particle sifting process (Section 5.3.2), which generates many smaller tables as data traverses the reorganization pipeline. Consequently, it is not unusual for a query workload to depend on multiple data objects. The significance of this lies in the fact that the merging process inflates the cost of pushing back workloads, thus making multi-object queries more favorable for direct execution on SmartNICs. Our evaluation, depicted in Figure 5.13, shows that due to the efficiency of the Apache Arrow in-memory data format, the performance overhead associated with data merging via memcpy is comparable to the combined cost of data deserialization and serialization, even for smaller data objects of size less than 10 MiB. Therefore, the process of serialization can be utilized as a means to merge data. Conversely, for queries that reference a single object, the corresponding data can be transmitted directly to the network, bypassing the cost of data merging.

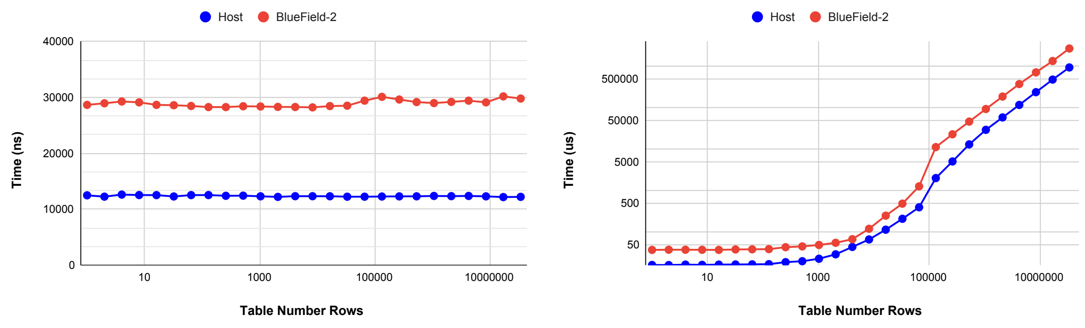


Figure 5.12: Arrow table (de)serialization performance. The host has two Intel Xeon 16-core CPUs running at 2.30GHz and 512 GB of memory. The host outperforms the BlueField-2 by 57% in deserialization (left figure) and by 60% in serialization (right figure).

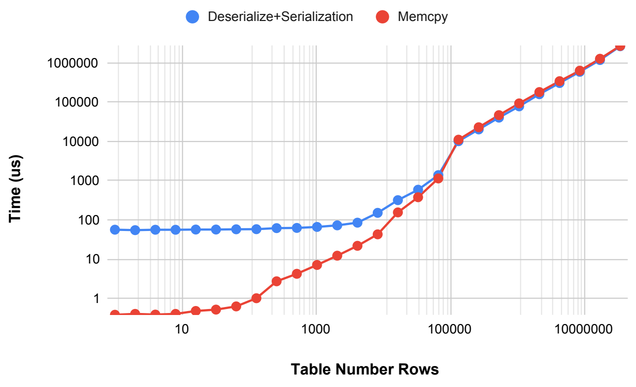


Figure 5.13: Performance of deserialization + serialization vs. memcpy on the BlueField-2

Network Transfer Time: Performing data filtering within SmartNICs can substantially reduce network data transfer overhead, depending on the query's selectivity. To evaluate the change in network overhead between offloaded and pushed-back execution, we can measure the difference in round-trip time required

Table 5.3: Particle data schema

Field	Type	Range
id	uint64	
time	uint64	
particle_id	uint64	
x	double	0.0 ∼1.0
y	double	0.0 ∼1.0
z	double	0.0 ∼1.0
vx	double	-100 ∼100
vy	double	-100 ∼100
vz	double	-100 ∼100

to request data buffers of different sizes from a SmartNIC.

Query Execution Time: This is the most challenging part because it involves the analysis of a query to estimate the amount of work required based on the complexity of its `input table` and `query conditions`. Without loss of generality, our analysis is based on data that conforms to the schema shown in Table 5.3. Let's consider the following simple SQL filter query

```
select * from particles where x < 0.5
```
(5.1)

The execution engine first locates the columnar data of `x`, compares the value of each `x` with the constant, and stores the comparison results (i.e., boolean values) in an array that has the same size as the column `x`. Each value in the boolean array indicates whether the current row of `x` should be included in the final result set. This process can be extended to queries with compound conditions by recursively merging internal conditions. Specifically, for a query with the filtering conditions `x < 0.5 and y >= 0.3`, we can evaluate the boolean values for the left-hand and right-hand sides of the logical "and" function and then merge the values of the two generated boolean arrays at the same index. In this way, multiple boolean arrays can be "collapsed" into a single array. Once this process is complete, rows that are marked as selected are copied, and the final result table is constructed.

For a simple query like 5.1, the computational requirements can be partitioned into three distinct components: 1) The comparison component, which involves a number of comparison operations equal to the number of rows in the table; 2) The filtering component, which requires a number of scan operations equal to the product of the number of rows in the table and the number of projected columns, to facilitate data preparation for selection; and 3) The selection component, which necessitates a number of copy operations equal to the product of the number of rows that satisfy the filter condition and the number of projected columns, to assemble the final result set. It is worth noting that different functions (e.g., comparison and selection) may have significantly different computational overheads. For instance, the power function consumes 10 times more CPU time than a logical function such as "and" on a BlueField-2 SmartNIC.

A similar analysis can be performed for more complex queries by counting the number of operations of different functions involved. However, care should be taken when evaluating a query involving statements with composed conditions. Consider the following SQL query, which involves a filtering statement composed of two conditions:

```
select particle_id from particles where
    power(x, 2) < 0.3 and power(y, 2) > 0.1
```
(5.2)

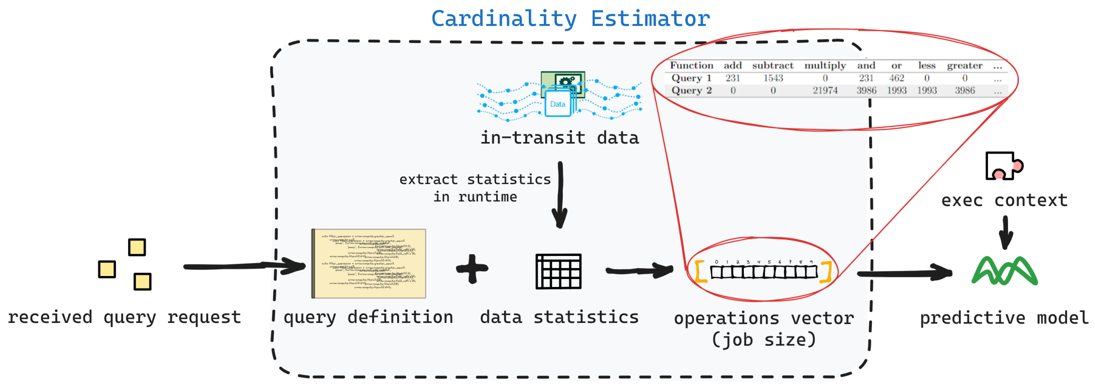
When estimating the computational requirements of this query, the relationship between the conditions must also be considered. Specifically, this query can be represented by two logical plans, one where both filtering conditions are expressed together in a single logical expression, and the other where the two conditions are separated into two logical expressions, resulting in optimized execution. By splitting the conditions, the data reduction achieved by the first filter can be utilized in the second filter, reducing the overall computational cost. Thus, for this optimized logical plan, the count of comparison and power operations involved in the second filter should be based on the result set size obtained after the first filtering.

In database systems, estimating the result set size of a query is referred to as cardinality estimation [268], and it represents a fundamental task in query processing and optimization. This field is a highly active area of research due to its wide-ranging applications, including network security monitoring [80], data streams [4], search engines and online data mining [108, 172].

To estimate the cardinality of a query or condition, we employ statistical tools grounded in prior research and commonly used assumptions in modern data management systems (e.g., PostgreSQL [71] and Spark SQL [16]). Specifically, for each column in a particle partition staged on a SmartNIC, we create a histogram that provides valuable insights into the distribution of values. By querying the quantile of the constant specified in a filtering condition within the appropriate histogram and applying the uniformity assumption to the statistics of each bin of the histogram, we are able to accurately estimate the cardinality of the condition. For compound conditions, we apply the independence assumption on sub-conditions to estimate the overall cardinality. In the case of aggregation conditions, we use tools that summarize the degree of data uniqueness of a column to estimate the cardinality of a query like `select count(particle_id) from particles where x < 0.5`.

To optimize the accuracy of our estimates in the face of complex queries and dynamic data flows, we progressively update and revise each histogram as new particle data is received, merged, or migrated.

Ultimately, we use the quantified number of operations as a vector to capture the job size of a query, which can then be incorporated into offloading schedulers based on relative job sizes or predictive models. Figure 5.14 presents a high-level view of the cardinality estimation process, with the results used as input for model prediction.



Function	add	subtract	multiply	and	or	less	greater	...
Query 1	231	1543	0	231	462	0	0	...
Query 2	0	0	21974	3986	1993	1993	3986	...

Figure 5.14: The process of cardinality estimation to generate vectors of operations as input to the model to predict query execution time

Implementation

To realize a decision engine for predictive scheduling, it is necessary to accurately predict the time required for each factor identified in the previous section. Once a SmartNIC receives a query workload request, the predicted time consumption for each factor can be aggregated, and a decision to push back the workload can be made by comparing the total predicted time consumption to that of pushing back the workload for execution by the host.

Predicting Serialization Time: Predicting the time required to serialize an Arrow table is a relatively straightforward task using machine learning techniques. To collect a training dataset, we measured the serialization time of 877 Arrow tables filled with randomly generated particle data on the BlueField-2 SmartNIC.

Each table was generated with a random number of rows ranging from 1 to 2^{25}. We used random forest regression as the algorithm for training. The prediction performance for serialization on the BlueField-2, as observed in a randomly generated test set, is presented in Figure 5.15. Notably, the prediction performance is seen to have an error rate within 7%, indicating the robustness and accuracy of our model.

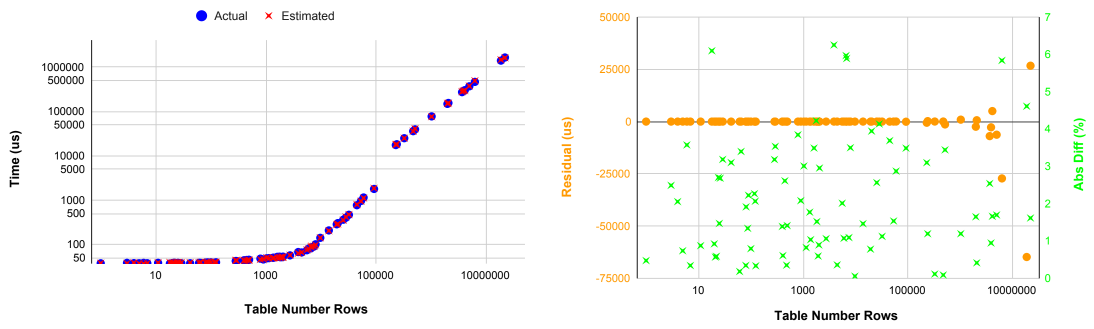


Figure 5.15: The left figure illustrates the actual and estimated serialization time on the BlueField-2 SmartNIC as a function of the number of rows in a table. The right figure displays the residual (on the left axis) and the absolute difference in percentage (on the right axis) between the actual serialization time and the estimated time predicted by the model.

Predicting Network Transfer Time: We used the Faodel library to handle data delivery and dispatch computations required for executing query workloads. While it is possible to predict the network transfer time required for a query workload based on both the schema of the resultant table object and the number of rows in the table, our approach was to anchor the prediction on the size of the table object to be transferred, which could be deduced from a prior prediction. This strategy was chosen as it isolates the network time modeling from the complexity of the table content, thus protecting the network time model from potential changes in the table schema in the future. The task of capturing the complexity of the table content can then be delegated to a separate model, which translates it into a value representing the size of the serialized table object to be used as input for the network time model.

In scenarios where the query workload is executed on the SmartNIC, the size of the serialized object can be predicted by considering the number of rows in the resulting table, as depicted in Figure 5.16. Conversely, when the query workload is to be pushed back for execution, the total serialized size of the data objects referenced by the query can be determined by aggregating their respective sizes.

To construct a training dataset for predicting network transfer time based on the data size to be transferred, we measured the round-trip time of requests dispatched to a SmartNIC to fetch local IPC buffers of varying sizes. We trained our model using the random forest regression algorithm, and the performance results for the communication between an HPC compute node and a local BlueField-2 SmartNIC on a randomly generated table test set are illustrated in Figure 5.17. This model consistently maintained error rates within single-digit percentages on the test set.

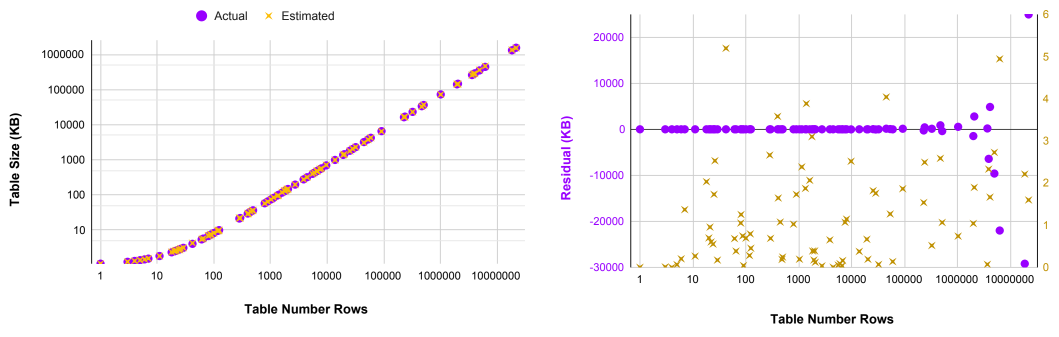


Figure 5.16: The left figure depicts the actual and model-estimated serialization sizes for Arrow tables with varying row numbers. The right figure illustrates the residual and absolute difference in percentage between the actual and estimated serialization sizes. The error rates observed with this model on the test data set do not exceed 6%.

Predicting Query Execution Time: We used the Theta Sketch [57] and KLL Sketch [125] algorithms from the Apache DataSketches library [199] to derive the distinct counting and histogram statistics of particle data tables, respectively.

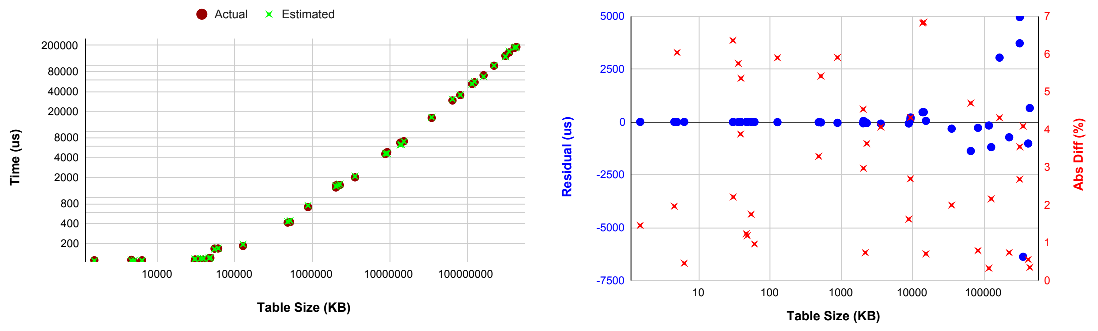


Figure 5.17: The left figure compares the measured time of network transfer using Faodel with the estimated time, plotted against the serialized table size. The right figure shows the residual (left axis) and absolute percentage difference (right axis) between the actual and estimated network transfer times. In this experiment, the table objects were hosted on the SmartNIC, and the host sent retrieval requests.

One of the significant advantages of this library is its ability to handle streaming data efficiently using the provided interfaces to update the created statistics over time, which is the key feature to be leveraged in our use case. Specifically, the library's single-update function satisfies the requirement for processing data one item at a time, while the bulk-update function enables merging statistics created for different tables. Additionally, the library offers parameters that allow us to fine-tune the accuracy of the estimation, enabling us to assess the trade-offs between estimation performance and memory consumption. Note that when counting the number of operations of different functions required by a query, the accuracy of the operation counts depends on the estimated cardinality of the sub-conditions in the query. Therefore, improving the accuracy of cardinality estimation can positively impact the precision of job size estimation for a query.

Our cardinality estimator currently supports queries that involve filtering, projection, aggregation, or any combinations of these operations, as demonstrated in Table 5.4. The estimator is also capable of estimating reducible conditions, such as the condition `sqrt(vy) > 20` in query **Q4**, by utilizing statistics based solely on fields. Another example of this feature is the condition `abs(vx) < 30`, which can be internally transformed to `vx > -30 and vx < 30`, allowing the use of histogram statistics for `vx` to estimate its cardinality. Furthermore, the estimator can estimate queries that rely on multiple data sources, as exemplified by query `Q5`. This capability is particularly significant because it enables workloads to query multiple partitions of data tables on the same SmartNIC simultaneously.

Performance evaluations show that the majority of estimations have an error rate within 1% of the actual cardinality, though some estimations may experience relatively high errors due to the aggregation of multiple statistics' biases. For instance, in query `Q8`, the estimator employs histogram statistics for the three filtering conditions and distinct counting statistics for the one aggregation condition. Additionally, it 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. The final column of Table 5.4 highlights the minimal cost of conducting cardinality estimation on the BlueField-2 SmartNIC, enabling the prediction of query workload placements with minimal resource utilization.

To convert an operation counts vector into execution time, we again utilized machine learning techniques. Our training dataset was created using query templates and randomized particle tables of varying sizes. Each query template was a C++ logical plan with placeholders that were filled with randomly generated constants to generate concrete logical plans. For instance, the query template `select * from particle_id where x >= k1 and y < k2 and z <= k3` could be instantiated by substituting `k1`, `k2`, and `k3` with numerical values. Table 5.5 presents some of our query templates in SQL form. Each record in our training dataset contained information on the number of operations performed by each function, the number of rows in the table queried, the number of threads used to execute the workload, and the actual execution time of the query. We employed the random forest model to train the data, and the prediction performance of a test set of queries is displayed in Table 5.6. It is worth noting that our cardinality estimator supports counting operations for sub-conditions, and therefore the execution time model can predict the time required to execute sub-conditions of a query. This capability enables us to make micro-level decisions for workloads defined by composable definitions.

Table 5.4: Performance of cardinality estimation

ID	SQL	Total Input Rows	Estimated Rows	Actual Rows	Diff (%)	Cost (%)
Q1	select ∗ from particles where x >= 0.7 and y < 0.3 and z <= 0.1	1048580	9566.87	9469	*1.03%*	*0.14%*
Q2	select ∗ from particles where x >= 0.7 and power(vx, 2) + power(vy, 2) + power(vz, 2) <= 10591.0	5044220	867170	861682	*0.64%*	*0.01%*
Q3	select x, power(vx, 2) + power(vy, 2) + power(vz, 2) as square_of_velocity from particles where square_of_velocity <= 1310.0	685359	18683	17138	*9.02%*	*0.02%*
Q4	select ∗ from particles where 3 + vx ∗ 2 <= 100 or sqrt(vy) > 20	1761670	1311370	1308000	*0.26%*	*0.06%*
Q5	select ∗ from (select ∗ from particles_1 where x >= 0.7 union select ∗ from particles_2 where y < 0.3) where z <= 0.1	786432	23338.2	23428	*0.38%*	*0.08%*
Q6	select count(id), min(x), max(x) from particles	1373630	1	1	*0.00%*	*0.04%*
Q7	select count(column_id), min(column_round_x), max(column_round_x) from (select id as column_id, round(x, 2) as column_round_x, bit_wise_and(particle_id, 15) as column_bit_wise_and_particle_id from particles) group by column_bit_wise_and_particle_id	2146820	16	16	*0.00%*	*0.01%*
Q8	select count(id) as column_hash_count_id from particles where x >= 0.7 and y < 0.3 and z <= 0.1 group by particle_id	1555010	13562.7	11277	*20.27%*	*0.09%*

- **Diff (%)** column shows the absolute percentage difference between the estimated and actual output rows of the query, calculated by using the formula *abs(estimated_rows - actual_rows) / actual_rows*.
- **Cost (%)** column shows the CPU time for estimation as a percentage of the CPU time for query execution on a BlueField-2 SmartNIC.
- Each query is evaluated in its logical query plan representation. The presentation of SQL queries in the table is for the purpose of illustration and ease of understanding.

Table 5.5: Example query templates for predicting execution time

ID	Template
T1	select ∗ from particles where x >= k1 and y < k2 and z <= k3
T2	select ∗ from particles where vy > k1 or (vx ∗ k2) <= k3
T3	select ∗ from particles where power(vz, 2) ∗ k1 > k2
T4	select ∗ from particles where power(x, 2) <= k1 or z ∗ k2 < k3 or vz > k4
T5	select count(∗) from particles where x < k1
T6	select particle_id, power(vx, 2) + power(vy, 2) + power(vz, 2) from particles where x >= k1
T7	select particle_id, x ∗ k1, y ∗ k2, z ∗ k3 from particles where x >= k4 and y < k5 and z <= k6
T8	select particle_id, x + k1, y - k2, z - k3 from particles where z > k1 and z <= k4
T9	select particle_id, (vx + k1) ∗ k2, (vy - k3) ∗ k4, (vz + k5) ∗ k6 from particles where (vx + vy + vz) > k1
T10	select particle_id, x + y - z, vx - vy + vz from particles where x > k1 and y < k2 and z > k3

Table 5.6: Prediction performance of execution time for different queries on the BlueField-2. The last column shows the absolute difference in percentage between the predicted and actual execution time.

Function	**add**	**and_kleene**	**filter**	**greater**	**greater_equal**	**less**	**less_equal**	**multiply**	**or_kleene**	**power**	**select**	**subtract**	**table_rows**	**num_threads**	**abs diff(%)**
R1	0	0	4.8242e+06	0	0	4.8242e+06	0	0	0	0	3.01624e+06	0	4824195	5	*0.190388*
R2	0	0	4.8242e+06	0	4.8242e+06	0	0	4.8242e+06	0	4.8242e+06	3.10695e+06	0	4824195	2	*3.87432*
R3	0	4.8242e+06	4.8242e+06	9.64839e+06	0	0	0	4.8242e+06	0	0	3.71326e+06	0	4824195	7	*11.3242*
R4	4.8242e+06	0	4.8242e+06	4.8242e+06	4.8242e+06	0	0	4.8242e+06	4.8242e+06	4.8242e+06	4.61516e+06	0	4824195	8	*24.7212*
R5	0	9.64839e+06	4.8242e+06	9.64839e+06	0	0	4.8242e+06	4.8242e+06	0	4.8242e+06	404500	0	4824195	7	*19.5861*
R6	0	4.8242e+06	4.8242e+06	0	0	0	9.64839e+06	4.8242e+06	0	0	1.10714e+06	0	4824195	1	*8.7674*
R7	0	0	4.8242e+06	4.8242e+06	0	0	0	4.8242e+06	0	0	281308	0	4824195	8	*35.9421*
R8	0	9.64839e+06	4.8242e+06	4.8242e+06	0	4.8242e+06	4.8242e+06	4.8242e+06	0	9.64839e+06	322406	0	4824195	3	*23.7687*
R9	0	0	4.8242e+06	9.64839e+06	0	0	0	0	4.8242e+06	1.14134e+07	3.80448e+06	0	4824195	4	*4.76398*
R10	0	9.64839e+06	4.8242e+06	4.8242e+06	9.64839e+06	0	0	0	0	0	249446	0	4824195	2	*2.52941*
R11	0	0	0	0	0	0	0	0	0	4.8242e+06	0	4.8242e+06	4824195	5	*0.398528*
R12	0	0	4.8242e+06	4.8242e+06	0	4.8242e+06	4.8242e+06	1.32448e+07	9.64839e+06	0	4.41494e+06	0	4824195	3	*23.6184*
R13	3.17116e+06	0	4.8242e+06	0	0	4.8242e+06	0	0	0	4.75675e+06	1.58558e+06	0	4824195	1	*4.42845*
R14	1.024e+06	4.8242e+06	4.8242e+06	0	0	4.8242e+06	4.8242e+06	0	0	0	512000	512000	4824195	7	*17.0599*
R15	1.17328e+07	0	4.8242e+06	4.8242e+06	0	0	0	6.25331e+06	0	0	2.08444e+06	4.16888e+06	4824195	1	*4.24343*
R16	4.06556e+06	0	4.8242e+06	9.64839e+06	0	0	4.8242e+06	0	9.64839e+06	0	4.06556e+06	1.21967e+07	4824195	3	*29.389*

Case Studies

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 5.11. 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 sets of data, we could evaluate the effectiveness and accuracy of our decision engine, which bases its scheduling decisions on aggregating all the estimated time consumption factors. Note that in both case studies, the queries only rely on individual data objects, so there was no overhead from data merging.

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

```
select * from particles where
    x >= 0.7 and y < 0.3 and z <= 0.1
```
(5.3)

The actual execution of this query results in a total of 55,517 rows, comprising all columns from the dataset and accounting for 0.9% of the total rows. The cardinality estimator projects an output of 55,036.7 rows, exhibiting a difference of 0.865% from the actual row count. Additionally, the estimator generated the following operations vector for the query on the given dataset:

Table 5.7: The operations vector produced for the case study query 5.3

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

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

```
select * from particles where
    x >= 0.5 and y < 0.55 and z <= 0.67
```
(5.4)

Executing this query on the same dataset yields 1,136,847 rows, representing 18.4% of the total row count. The cardinality estimator predicts a return of 1,152,860 rows, marking a minor discrepancy of 1.41%. The operations vector is generated as follows:

Table 5.8: The operations vector produced for the case study query 5.4

and_kleene	filter	greater_equal	less	less_equal	select	table_rows
12355500	6177730	6177730	6177730	6177730	1152860	6177731

The estimated and actual time consumption for each of the time factors in the two scenarios are summarized in the table 5.9. 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%. This outcome can be attributed to the significant 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. While the estimation slightly leans towards pushing back, keeping execution on the SmartNIC reduces latency by only 1.38%.

Table 5.9: Analysis of time consumption for offloaded vs. pushed-back execution with case study queries. The first and second tables correspond to results from queries 5.3 and 5.4, respectively.

	Offloaded			**Pushed-back**		
	Estimated (us)	Actual (us)	Abs Diff	Estimated (us)	Actual (us)	Abs Diff
Raw Data Deserialization	28.928	28.928	0%	12.353	12.353	0%
Query Execution	54737.5	49316	10.99%	7781.469	8936	12.92%
Result Serialization	1012.84	1039.47	2.56%	-	-	-
Network Transfer	2289.95	2258.25	1.40%	189298	198676	4.72%
Result Deserialization	12.353	12.353	0%	-	-	-
Sum	**58081.57**	**52655**	**10.31%**	**197091.82**	**207624.35**	**5.07%**

	Offloaded			**Pushed-back**		
	Estimated (us)	Actual (us)	Abs Diff	Estimated (us)	Actual (us)	Abs Diff
Raw Data Deserialization	28.928	28.928	0%	12.353	12.353	0%
Query Execution	93173	88511	5.27%	16799.339	15525	8.21%
Result Serialization	87124.1	86932.8	0.22%	-	-	-
Network Transfer	35034.4	35781.1	2.09%	189298	198676	4.72%
Result Deserialization	12.353	12.353	0%	-	-	-
Sum	**215372.78**	**211266.18**	**1.94%**	**206109.69**	**214213.35**	**3.78%**

Discussion

There are alternative approaches to gathering insights to make workload placement decisions. Yang [264] proposed an online sampling-based approach for handling dynamic workload offloading to computational storage devices. While this approach may be more intuitive to reason about, it necessitates periodic sampling to maintain prediction accuracy in the face of dynamic workloads. This periodic sampling requirement may negatively impact workload performance on these embedded devices due to their limited resources. In contrast, constructing a prediction model using offline data resolves the limitations on data size and computational requirements. Our case study found that offline data modeling can deliver high accuracy while minimizing the cost of generating decisions for query workload placements.

Our next step is to apply the decision engine to facilitate dynamic offloading between hosts and SmartNIC pipelines that govern data flows, and to assess the performance and potential trade-offs. Although the model is specifically designed for SmartNICs, it can also be extended to encompass embedded storage devices typically deployed at the end of the data processing pipeline. Additionally, the results produced by our decision engine can be used not only for making micro-level decisions on composable workload definitions to allow partial workload offloading, but also for implementing reactive schedulers in cases where throughput performance is prioritized.

Our current decision engine does not support join table queries, as we have primarily focused on the data model of the particle dataset. Despite extensive research [145, 265, 167, 79] on estimating the cardinality of join queries, findings suggest that prediction accuracy may significantly decrease with an increase in the number of joins [136, 144]. Thus, careful evaluation is required before extending our decision engine to support join queries. Additionally, in situations where higher prediction accuracy is necessary, further efforts to improve our current results may be necessary for future work. Nevertheless, our research demonstrates the potential of even basic machine-learning techniques in facilitating dynamic offloading and enabling the extension of data services to embedded systems.

There are multiple directions forward from this work. One possible direction is to explore the potential benefits of implementing caches for serialized particle data, which could help mitigate the overhead of pushing back query workloads. However, this approach immediately raises a question of how much performance can be improved to cache in-transit data. Moreover, as SmartNICs manage partitioned particle data for various reasons, pushing back query workloads for general queries requires merging all relevant data before transferring over the network to minimize network management overhead. As depicted in Figure 5.13, the time taken for memory copying is comparable to the combined time for deserialization and serialization on the SmartNIC, even when the table partition size is relatively small. Therefore, a careful evaluation is required to quantify the effects of caching and determine the scenarios where applying caches could provide performance benefits.

Another possible direction is to explore the classification of query types based on their suitability for offloading to SmartNICs. Certain queries are well-suited for offloading, particularly those that involve scanning large datasets with minimal computation. For example, the query `select count(*) from particles` can almost always be executed more efficiently on a SmartNIC than on a host. In this scenario, the execution engine can scan only a single column of the table and generate the result containing only one row, resulting in improved performance. However, pushing back this query requires serializing all data and transferring it over the network, resulting in substantial overhead.

5.4.6 Summary

Dynamic offloading is an essential function that enables extending data services to embedded systems, such as SmartNICs, without potentially degrading performance. We have discussed two strategies for pushing back data service workloads, the challenges involved in implementing a decision engine to determine whether to push back, and our implementation of the engine. The results are promising, and we are looking forward to applying the decision engine and evaluating the performance of dynamic offloading for HPC data service workloads.

5.5 Conclusion

The implementation of dynamically offloading HPC data service workloads to embedded systems presents significant challenges, requiring efficient solutions to communication, computation, and scheduling complexities. In addressing some of these challenges within a domain-specific area for managing particle data flows, we implemented Bitar to reduce data serialization overhead during communication. Additionally, we developed a particle-sifter pipeline that enables different endpoints within the HPC platform to interact with groups of SmartNICs and transform data into a sorted format. Finally, we discussed our implementation of a decision engine that facilitates making workload placement decisions for data query workloads.

Chapter 6

Conclusion

In this chapter, I draw conclusions by discussing the ongoing work, outlining potential future research directions, and providing a summary that encapsulates our principal findings.

6.1 Ongoing and Future Work

Embedded systems are highly specialized for domain-specific usage. To harness the benefits of employing embedded systems, continuous research efforts are required to identify potential use cases and implement appropriate solutions, as evidenced in the history of the challenges faced in employing embedded systems across diverse applications. While this thesis contributes to the fields from evaluation metrics to potential use cases and strategies for achieving performance benefits, there remains ample space for exploration, such as investigating new hardware designs, exploring novel software architectures, and optimizing the integration of embedded systems with emerging technologies.

6.1.1 Query Performance with Dynamic Offloading

In the chapter discussing offloading strategies, I presented the performance estimation of the decision engine. However, the actual performance of online query workloads, in accordance with the scheduling of the decision engine, is yet to be evaluated. There exist several questions that require further investigation to achieve optimal scheduling performance:

- **Runtime Resource Availability:** Although our model includes the number of threads as a feature for predicting query execution time, there are other runtime factors or resources that can also impact performance, such as the availability of memory, memory and network bandwidth, and CPU core scaling frequency. As more data service workloads are offloaded to SmartNICs, the effects of these resources' availability can become increasingly significant. Further research is required to investigate the impact of these factors on query execution performance and to develop more accurate predictive models that incorporate them. There are tools we may leverage to trace the resource utilization of a system at runtime, such as Jaeger [110] and OpenTracing API[1].

- **Congestion Control:** There are scenarios in which multiple SmartNICs may decide to push back data query workloads to the same host that initiated all these requests. This can lead to the host becoming overloaded, resulting in worse execution performance for the pushed-back workloads than if only a portion of the SmartNICs were to push back their workloads. To avoid overloading the host with pushed-back aggregation, initially, we may incorporate a probability of pushback into requests from the same host. This can help reduce the likelihood of all sent requests returning to the host. We

[1] `https://github.com/opentracing`

can also prorate this probability based on the relative benefit of executing the workload on the host, allowing workloads with a greater relative benefit to be more likely to be pushed back. However, since this is not an optimal solution, it may eventually require SmartNICs and hosts to exchange their runtime status through an additional peer-to-peer channel.

- **Query Workload Classification:** The presence of data serialization overhead when a workload is designated for pushback implies that certain query workloads may be less likely to be pushed back. For instance, executing the query `select count(*) from particle` entails significant I/O operations but fewer computing operations. Moreover, the computational requirement of this query workload could be even lower than that of serializing the referenced dataset. Thus, executing this workload on SmartNICs is preferred, even when the SmartNIC is overloaded. Conversely, there are query workloads that necessitate more computing operations than I/O. These workloads will likely be pushed back to the host for improved performance. Classifying queries based on the likelihood of being pushed back can aid data consumer services in optimizing and scheduling their requests. Additionally, identifying the boundaries of each class can assist embedded system designers in configuring resources to optimize for a given type of workload.

These areas may constitute only a subset of the space that needs to be explored. Moreover, the necessary accuracy of workload execution time prediction may be sensitive to the performance gap between the embedded device and the host. I hope that further efforts will be directed towards these areas to fully realize the benefits of dynamic offloading for query and additional data service workloads.

6.1.2 Cost-benefit Quantification for East-West Data Services

While our MBWU evaluation prototype successfully demonstrated the benefits of offloading data access functions with north-south data movement, it is important to note that the MBWU-based methodology for cost-benefit quantification can also be used for data services moving data in the east-west direction within the system hierarchy. However, additional challenges are associated with measuring an MBWU, as the configurations that can maximize the performance of every involved storage device simultaneously may not be intuitively practical. Additionally, it can be arduous to separate a specific east-west data service from other integrated services within an application. As a result, evaluating the benefits of offloading east-west data services is expected to require significant engineering efforts or may need to compromise with the measurement results reflecting the benefits of a bundle of services. For example, Ceph's data replication service is coupled with the data authentication and encryption service [58], making it difficult to separate the replication service without potentially breaking the service protocol. Our preliminary results show that offloading an entire OSD to embedded storage devices may result in a negative cost benefit.

6.1.3 Security and Performance Isolation

As embedded devices, such as SmartNICs, become more prevalent and widely adopted in data centers, security and performance isolation issues are increasingly important due to multi-tenancy requirements for sharing resources on the same embedded system. Existing security frameworks provided by device vendors mainly focus on communication between hosts and the device [48]. However, functions offloaded to the device also require security isolation between workloads with the same or different functions [269]. Solutions such as eBPF are gaining attention for enabling zero-trust for network functions. However, similar solutions are needed to apply to embedded devices, with the challenge of ensuring efficiency in a constrained environment. In addition to security, enabling performance isolation is critical to avoid workloads with the same priority being starved during resource competition. FairNIC [95] proposes an approach based on static resource partitions such as core IDs and physical memory regions. However, as workloads on embedded systems become more dynamic and the computing power on these devices continues to increase, time-sharing solutions are promising, similar to most of the process schedulers employed in Linux [29].

6.2 Summary

The tension between the increasing demand for computing efficiency in data processing and the narrowing outlook of computing power delivery by general-purpose systems has underscored the significance of domain-specific hardware, or embedded systems, as essential components for improving the performance efficiency of data services. Despite the relevance of past research outcomes, effectively leveraging embedded systems in the current era of big data and advanced technology capable of providing a higher density of computing power integration presents both opportunities and challenges, particularly when considering the specialized nature of these systems. In addressing these challenges, this thesis aims to explore three dimensions, namely why, what, and how, to tap into the potential of embedded systems and overcome these challenges:

Why Offloading (Chapter 3): I tackle this challenge by proposing a set of metrics and evaluation methodologies to quantify the cost-benefit of offloading a data access function. Our MBWU-based efficiency metrics can assist system architects and application developers in measuring the improvement in cost, power, and space efficiency through offloading based on a normalization of the expenses to the performance of a given storage media. I have also developed a prototype that implements the MBWU-based evaluation methodology, which automates the benefit evaluation of offloading the key-value data access function to systems with varying configurations. The results demonstrate that our metrics provide valuable insights into the resource composition of a system for optimizing the efficiency of running the specific data access workload. Additionally, I presented a mathematical model to evaluate the data availability benefit of utilizing embedded storage nodes to construct data storage layers. As embedded storage nodes are more cost- and space-efficient, edge data centers can deploy more of these nodes as independent failure domains, leading to a significant improvement in the data availability benefit compared to storage systems constructed with general-purpose servers that typically have a high degree of storage and computing integration.

What to Offload (Chapter 4): I tackle this challenge by proposing a method with microbenchmarks to create performance envelopes for embedded devices. I further evaluated the processing headroom for network functions offloaded to a specific SmartNIC. These evaluations provide valuable insights into the functions and use cases that applications can leverage. Although the current generation of SmartNICs lags behind general-purpose systems in terms of data processing performance, offloading to embedded hardware is particularly attractive for applications that require a large number of asynchronous operations for data manipulation and distribution, as well as operations that can benefit from hardware accelerators. In addition, I evaluated two specific use cases: partitioning particle data flows and executing parallel data processing from HPC workflows on the BlueField-2 SmartNIC. The results indicate that offloading these functions to SmartNICs can reduce performance interference to host applications by isolating I/O intervention, which is a significant benefit in optimizing HPC workloads.

How to Offload (Chapter 5): I tackle this challenge by proposing strategies to better utilize the resources on embedded systems. Our approach involves the development of a library called "bitar," which enhances the performance of data serialization, a critical component of many data services. Bitar leverages the parallelism and zero-copy capabilities of the accelerator hardware to improve data compression performance, achieving performance efficiency superior to that of modern general-purpose processors. I then demonstrated an embedded processing pipeline with a software stack that applies large-scale data transformation processes on a cluster of SmartNICs. Finally, I designed a decision engine that enables the dynamic offloading of a common data service, the data query service, to SmartNICs. This engine estimates the execution time for both offloaded and pushed-back query scenarios to determine the most efficient placement for executing a query workload. This work builds upon existing research efforts in the fields of database query optimization and predictive models used in data science, thus opening the door to continuously optimizing the dynamic offloading of data services to embedded systems.

The benefits of utilizing embedded systems for data services are multifaceted, often varying based on the specifics of application use cases. Therefore, thoughtful software and hardware design is indispensable for harnessing the performance potential of offloaded data services. This thesis contributes to both software and hardware design dimensions, providing insights that system designers and application developers can utilize to optimize data services on embedded systems. I hope this thesis can serve as a stepping stone toward a future where more efficient, cost-effective data service solutions become the standard.

Bibliography

[1] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the international workshop on openCL 2013 & 2014*, pages 1–9, 2014.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices*, 33(11):81–91, 1998.

[3] Anant Agarwal and Markus Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, pages 750–753, 2007.

[4] Charu C Aggarwal and Philip S Yu. A survey of synopsis construction in data streams. *Data streams: models and algorithms*, pages 169–207, 2007.

[5] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[6] Tanveer Ahmad, Nauman Ahmed, Johan Peltenburg, and Zaid Al-Ars. Arrowsam: In-memory genomics data processing using apache arrow. In *2020 3rd International Conference on Computer Applications & Information Security (ICCAIS)*, pages 1–6. IEEE, 2020.

[7] Israr Ahmed and Abdul Aziz. Dynamic approach for data scrubbing process. *International Journal on Computer Science and Engineering*, 2(02):416–423, 2010.

[8] J Alexander. A vme interface to an ibm mainframe computer. Technical report, CERN, 1985.

[9] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

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

[11] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. AI and compute. `https://openai.com/research/ai-and-compute`.

[12] 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.

[13] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, 2009.

[14] Apache Arrow. The Apache Arrow Dataset Interface. `https://arrow.apache.org/docs/python/api/dataset.html`.

[15] TD Arber, Keith Bennett, CS Brady, A Lawrence-Douglas, MG Ramsay, NJ Sircombe, P Gillies, RG Evans, Holger Schmitz, AR Bell, et al. Contemporary particle-in-cell approach to laser-plasma modelling. *Plasma Physics and Controlled Fusion*, 57(11):113001, 2015.

[16] 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.

[17] Storage Networking Industry Association. What Is Computational Storage? `https://www.snia.org/education/what-is-computational-storage`.

[18] Saurabh Bagchi, Muhammad-Bilal Siddiqui, Paul Wood, and Heng Zhang. Dependability in edge computing. *Commun. ACM*, 63(1):58–66, December 2019.

[19] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. {CORFU}: A shared log design for flash clusters. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 1–14, 2012.

[20] Francisco Barahona, Stuart Bermon, Oktay Günlük, and Sarah Hood. Robust capacity planning in semiconductor manufacturing. *Naval Research Logistics (NRL)*, 52(5):459–468, 2005.

[21] Mohammadreza Bayatpour, Nick Sarkauskas, Hari Subramoni, Jahanzeb Maqbool Hashmi, and Dhabaleswar Kumar Panda. BluesMPI: Efficient MPI non-blocking all-to-all offloading designs on modern BlueField Smart NICs. In *Proceedings of High Performance Computing: 36th International Conference, ISC High Performance 2021*, 2021.

[22] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 49–65, 2014.

[23] 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, 2018.

[24] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn Lockwood, Vakho Tsulăia, Suren Byna, Steven Farrell, Doga Gursoy, Chris Daley, Vince Beckner, Brian Van Straalen, David Trebotich, Craig Tull, Gunther Weber, Nicholas Wright, Katie Antypas, and Prabhat. Accelerating science with the NERSC burst buffer early user program. In *Proceedings of the 2016 Cray User Group Conference*, 2016.

[25] Jakob Blomer. A quantitative review of data formats for hep analyses. In *Journal of Physics: Conference Series*, volume 1085, page 032020. IOP Publishing, 2018.

[26] Simona Boboila, Youngjae Kim, Sudharshan S Vazhkudai, Peter Desnoyers, and Galen M Shipman. Active flash: Out-of-core data analytics on flash storage. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[27] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

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

[29] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[30] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[31] Eric Brewer. Spinning disks and their cloudy future. *14th USENIX Conference on File and Storage Technologies (FAST '16)*, 2016.

[32] Ronald B Brightwell. Challenges and opportunities for hpc interconnects and mpi. *DOE Office of Scientific and Technical Information (OSTI)*, 2017.

[33] Richard Eric Brown, Shalini Gupta, Richard D Christie, Subrahmanyam S Venkata, and R Fletcher. Distribution system reliability assessment using hierarchical markov modeling. *IEEE Transactions on power Delivery*, 11(4):1929–1934, 1996.

[34] Brad Burres, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. Intel's hyperscale-ready infrastructure processing unit (ipu). In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–16. IEEE, 2021.

[35] Idan Burstein. Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20. IEEE, 2021.

[36] Idan Burstein. NVIDIA data center processing unit (DPU) architecture. In *Proceedings of the 2021 IEEE Hot Chips 33 Symposium*, 2021.

[37] 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.

[38] Zhichao Cao and Siying Dong. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.

[39] Michael J Carey, Nicola Onose, and Michalis Petropoulos. Data services. *Communications of the ACM*, 55(6):86–97, 2012.

[40] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. Software defined storage. *Storage Networking Industry Assoc. working draft*, pages 20–24, 2014.

[41] Adrian M Caulfield, Laura M Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *ACM Sigplan Notices*, 44(3):217–228, 2009.

[42] 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.

[43] Jayjeet Chakraborty, Ivo Jimenez, Sebastiaan Alvarez Rodriguez, Alexandru Uta, Jeff LeFevre, and Carlos Maltzahn. Skyhook: Towards an arrow-native storage system. *arXiv preprint arXiv:2204.06074*, 2022.

[44] Jayjeet Chakraborty, Ivo Jimenez, Sebastiaan Alvarez Rodriguez, Alexandru Uta, Jeff LeFevre, and Carlos Maltzahn. Skyhook: Towards an Arrow-native storage system. In *Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*, 2022.

[45] Jayjeet Chakraborty, Carlos Maltzahn, David Li, and Tom Drabas. Skyhook: Bringing Computation to Storage with Apache Arrow. `https://arrow.apache.org/blog/2022/01/31/skyhook-bringing-computation-to-storage-with-apache-arrow/`, January 2022.

[46] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. An overview of business intelligence technology. *Communications of the ACM*, 54(8):88–98, 2011.

[47] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[48] Scott Ciccone and John F. Kim. NVIDIA DOCA: a foundation for zero trust. `https://developer.nvidia.com/blog/nvidia-introduces-bluefield-dpu-as-a-platform-for-zero-trust-security-with-doca-1-2/`, November 2021.

[49] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 37–48, 2013.

[50] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies*, pages 46–66. Springer, 2001.

[51] Yann Collet. LZ4 Frame Format Description. `https://github.com/lz4/lz4/blob/dev/doc/lz4_Frame_format.md`, December 2020.

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

[53] Apache Arrow Community. Acero: A C++ streaming execution engine. `https://arrow.apache.org/docs/cpp/streaming_execution.html`. [Accessed 06-Apr-2023].

[54] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[55] Peter F Corbett and Dror G Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems (TOCS)*, 14(3):225–264, 1996.

[56] Patrick Crowley, Marc E Fiuczynski, Jean-Loup Baer, and Brian Bershad. Workloads for programmable network interfaces. *Workload Characterization for Computer System Design*, pages 135–147, 2000.

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

[58] Anthony D'atri, Vaibhav Bhembre, and Karan Singh. *Learning Ceph: Unifed, scalable, and reliable open source storage solution.* Packt Publishing Ltd, 2017.

[59] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2015.

[60] David J De Witt. Direct—a multiprocessor organization for supporting relational database management systems. *IEEE Transactions on Computers*, 100(6):395–406, 1979.

[61] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D. Carothers, Kerstin Kleese van Dam, Kenneth Moreland, M. Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey S. Vetter. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32:159 – 175, 2018.

[62] Julien Derouillat, Arnaud Beck, Frédéric Pérez, Tommaso Vinci, M Chiaramello, Anna Grassi, M Flé, Guillaume Bouchard, I Plotnikov, Nicolas Aunai, et al. Smilei: A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation. *Computer Physics Communications*, 222:351–373, 2018.

[63] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

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

[65] Noah Diamond, Scott Graham, and Gilbert Clark. Securing InfiniBand networks with the Bluefield-2 data processing unit. In *Proceedings of the International Conference on Cyber Warfare and Security*, 2022.

[66] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: A systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[67] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: An interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[68] Jörg Domaschka, Christopher B Hauser, and Benjamin Erb. Reliability and availability properties of distributed database systems. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, pages 226–233. IEEE, 2014.

[69] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[70] Matthieu Dorier. *Addressing the challenges of I/O variability in post-petascale HPC simulations.* PhD thesis, Ecole Normale Supérieure de Rennes, 2014.

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

[72] Cosmin Dumitru, Ralph Koning, Cees de Laat, et al. Clearstream: Prototyping 40 gbps transparent end-to-end connectivity. *Technical reports*, (UVA-SNE-2011-02), 2011.

[73] 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.

[74] Chris Edwards. Moore's law: What comes next? *Communications of the ACM*, 64(2):12–14, 2021.

[75] H Carter Edwards and Christian R Trott. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24. IEEE, 2013.

[76] 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*, 74:3202–3216, 2014.

[77] Lieven Eeckhout. Is moore's law slowing down? what's next? *IEEE Micro*, 37(04):4–5, 2017.

[78] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 365–376, 2011.

[79] Cristian Estan and Jeffrey F Naughton. End-biased samples for join cardinality estimation. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 20–20. IEEE, 2006.

[80] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166, 2003.

[81] Yuanwei Fang, Chen Zou, and Andrew A Chien. Accelerating raw data analysis with the accorda software and hardware architecture. *Proceedings of the VLDB Endowment*, 12(11):1568–1582, 2019.

[82] Marc E Fiuczynski, Richard P Martin, Tsutomu Owa, and Brian N Bershad. Spine: a safe programmable and integrated network environment. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 7–12, 1998.

[83] Linux Foundation. Data plane development kit (DPDK). `http://www.dpdk.org`, 2015.

[84] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[85] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *USENIX Annual Technical Conference (ATC)*, 2019.

[86] Jiechao Gao, Haoyu Wang, and Haiying Shen. Smartly handling renewable energy instability in supporting a cloud datacenter. In *2020 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 769–778. IEEE, 2020.

[87] Nishant Garg. *Apache kafka.* Packt Publishing Birmingham, UK, 2013.

[88] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Querying and mining data streams: you only get one look a tutorial. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 635–635, 2002.

[89] Sanjay Ghemawat and Jeff Dean. Leveldb, 2011.

[90] Garth A Gibson, David F Nagle, Khalil Amiri, Fay W Chang, Eugene M Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, et al. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–284, 1997.

[91] GoogleTechTalks and Gordon Deborah. How ant colonies get things done. `https://www.youtube.com/watch?v=R07_JFfnFnY`, Apr 2008. [Online; accessed 25-May-2020].

[92] Deborah M Gordon. *Ants at work: how an insect society is organized.* Simon and Schuster, 1999.

[93] Deborah M Gordon. *Ant encounters: interaction networks and colony behavior*, volume 1. Princeton University Press, 2010.

[94] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[95] 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.

[96] Kevin M Greenan, James S Plank, Jay J Wylie, et al. Mean time to meaningless: Mttdl, markov models, and storage system reliability. In *HotStorage*, pages 1–5, 2010.

[97] Brendan Gregg. Computing performance 2022: What's on the horizon. *USENIX SREcon 2022*, 2022.

[98] Junmin Gu, Burlen Loring, Kesheng Wu, and E Wes Bethel. Hdf5 as a vehicle for in transit data movement. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 39–43, 2019.

[99] Ajay Gulati, Ganesha Shanmuganathan, Anne M Holler, and Irfan Ahmad. Cloud scale resource management: Challenges and techniques. *HotCloud*, 11:3–3, 2011.

[100] 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.

[101] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.

[102] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE, 2021.

[103] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[104] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[105] Nikos Hardavellas. The rise and fall of dark silicon. *The advanced computing systems association*, pages 7–17, 2012.

[106] Francis H Harlow. The particle-in-cell computing method for fluid dynamics. *Methods Comput. Phys.*, 3:319–343, 1964.

[107] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[108] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692, 2013.

[109] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[110] Jonas Höglund. An analysis of a distributed tracing systems effect on performance jaeger and opentracing api, 2020.

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

[112] Song Huang, Shucai Xiao, and Wu-chun Feng. On the energy efficiency of graphics processing units for scientific computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[113] IBM Systems Development Division, Department G24, San Jose, California 95114. *Reference Manual for IBM 2835 Storage Control and IBM 2305 Fixed Head Storage Module.*

[114] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[115] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[116] Nayeem Islam and Roy Want. Smartphones: Past, present, and future. *IEEE Pervasive Computing*, 13(4):89–92, 2014.

[117] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. Magpie: Python at speed and scale using cloud backends. In *CIDR*, 2021.

[118] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[119] Rick Jones. Netperf benchmark. *http://www. netperf. org/*, 2012.

[120] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2017.

[121] Sang-Woo Jun, Ming Liu, and Kermin Elliott Fleming. Scalable multi-access flash store for big data analytics. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 55–64, 2014.

[122] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.

[123] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, 2006.

[124] Sara Karamati, Clayton Hughes, Karl S. Hemmert, Ryan E. Grant, Whit Schonbein, Scott Levy, Thomas M. Conte, Jeffrey S. Young, and Richard W. Vuduc. “Smarter" NICs for faster molecular dynamics: a case study. In *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium*, 2022.

[125] 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.

[126] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. A case for intelligent disks (idisks). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[127] Gokcen Kestor, Roberto Gioiosa, Darren J Kerbyson, and Adolfy Hoisie. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 56–65. IEEE, 2013.

[128] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 756–771, 2021.

[129] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory ssds. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, 2011.

[130] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Information Sciences*, 327:183–200, 2016.

[131] Colin King. Stress-ng: A tool to load and stress a computer system. `http://kernel.ubuntu.com/git/cking/stress-ng.git`. [Accessed 06-Apr-2023].

[132] Philip Kufeldt, Carlos Maltzahn, Tim Feldman, Christine Green, Grant Mackey, and Shingo Tanaka. Eusocial storage devices-offloading data management to storage devices that can act collectively. *; login: The USENIX Magazine*, 43(2):16–22, 2018.

[133] HT Kung. Network-based multicomputers: redefining high performance computing in the 1990s. In *Proceedings of the Decennial Caltech Conference on VLSI*. Carnegie Mellon University, 1989.

[134] Ian Kuon, Russell Tessier, Jonathan Rose, et al. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.

[135] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[136] Hai Lan, Zhifeng Bao, and Yuwei Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6:86–101, 2021.

[137] Minh Le, Zheng Song, Young-Woo Kwon, and Eli Tilevich. Reliable and efficient mobile edge computing in highly dynamic and volatile environments. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 113–120. IEEE, 2017.

[138] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, 2017.

[139] Scott Le Grand, Andreas W Götz, and Ross C Walker. Spfp: Speed without compromise—a mixed precision model for gpu accelerated molecular dynamics simulations. *Computer Physics Communications*, 184(2):374–380, 2013.

[140] Doug Lea and Wolfram Gloger. A memory allocator, 1996.

[141] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18–es, 2007.

[142] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. Accelerating external sorting via on-the-fly data merge in active ssds. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[143] Jeff LeFevre and Carlos Maltzahn. Skyhookdm: Data processing in ceph with programmable storage. *USENIX login;*, 45(2), 2020.

[144] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[145] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *Cidr*, 2017.

[146] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[147] 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.

[148] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4):321–374, 1990.

[149] Weigang Li and Yu Yao. Accelerate data compression in file system. In *2016 Data Compression Conference (DCC)*, pages 615–615. IEEE Computer Society, 2016.

[150] Chyuan Shiun Lin, Diane CP Smith, and John Miles Smith. The design of a rotating associative memory for relational database applications. *ACM Transactions on Database Systems (TODS)*, 1(1):53–65, 1976.

[151] 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.

[152] Jianshen Liu. ljishen/SmartNIC-WU: relative performance of individual stressors. `https://raw.githubusercontent.com/ljishen/SmartNIC-WU/main/results/stress-ng/sequential-all_60s.platforms_summary.svg`, 1 2022.

[153] Jianshen Liu. ljishen/SmartNIC-WU: relative performance of stresssor classes. `https://raw.githubusercontent.com/ljishen/SmartNIC-WU/main/results/stress-ng/sequential-all_60s_Npc3000.platforms_summary.per_class.svg`, 1 2022.

[154] Jianshen Liu. Simplify accessing hardware compression/decompression accelerators. `https://github.com/ljishen/bitar`, 7 2022.

[155] Jianshen Liu, Philip Kufeldt, and Carlos Maltzahn. Mbwu: Benefit quantification for data access function offloading. In *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers 34*, pages 198–213. Springer, 2019.

[156] Jianshen Liu and Matthew Leon. Scale-out edge storage systems with embedded storage nodes to get better availability and cost-efficiency at the same time. *HotEdge'20*, 2020.

[157] 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.

[158] Jianshen Liu, Carlos Maltzahn, Matthew L. Curry, and Craig Ulmer. Processing particle data flows with SmartNICs. In *Proceedings of the 2022 IEEE High Performance Extreme Computing Conference*, 2022.

[159] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. Performance characteristics of the bluefield-2 smartnic. *arXiv preprint arXiv:2105.06619*, 2021.

[160] Jianshen Liu, Carlos Maltzahn, Craig D. Ulmer, and Matthew Leon Curry. Performance characteristics of the bluefield-2 smartnic. *DOE Office of Scientific and Technical Information (OSTI)*, 5 2021.

[161] 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.

[162] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *USENIX annual technical conference*, pages 363–378, 2019.

[163] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[164] Wenbin Lu, Luis E. Peña, Pavel Shamis, Valentin Churavy, Barbara Mary Chapman, and Steve Poole. Bring the BitCODE-moving compute and data in distributed heterogeneous systems. In *Proceedings of the 2022 IEEE International Conference on Cluster Computing*, pages 12–22, 2022.

[165] John MacCormick, Nicholas Murphy, Venugopalan Ramasubramanian, Udi Wieder, Junfeng Yang, and Lidong Zhou. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Transactions On Storage (TOS)*, 4(4):1–28, 2009.

[166] Rohan Mahapatra, Soroush Ghodrati, Byung Hoon Ahn, Sean Kinzer, Shuting Wang, Hanyang Xu, Lavanya Karthikeyan, Hardik Sharma, Amir Yazdanbakhsh, Mohammad Alian, et al. Domain-specific computational storage for serverless computing. *arXiv preprint arXiv:2303.03483*, 2023.

[167] Tanu Malik, Randal C Burns, and Nitesh V Chawla. A black-box approach to query cardinality estimation. In *CIDR*, pages 56–67, 2007.

[168] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[169] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[170] W McKinney. Introducing apache arrow flight: A framework for fast data transport, 2019.

[171] 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.

[172] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Why go logarithmic if we can go linear? towards effective distinct counting of search traffic. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 618–629, 2008.

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

[174] 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.

[175] Kenneth Moreland, Christopher M. Sewell, Will Usher, Li-Ta Lo, Jeremy S. Meredith, David Pugmire, James Kress, Hendrik A. Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. VTK-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications*, 36:48–58, 2016.

[176] Rafael Moreno-Vozmediano, Eduardo Huedo, Rubén S Montero, and Ignacio M Llorente. A disaggregated cloud architecture for edge computing. *IEEE Internet Computing*, 23(3):31–36, 2019.

[177] Jacques Nadeau. Substrait: Cross-Language Serialization for Relational Algebra. `https://substrait.io/`.

[178] Jacques Nadeau. Vectorized Query Processing for CPUs using Apache Arrow. `https://www.youtube.com/watch?v=hLm_duqB3Y4`, December 2019.

[179] National Oceanic and Atmospheric Administration. Vessel Traffic: Ais vessel tracks. `https://coast.noaa.gov/digitalcoast/data/vesseltraffic.html`, 2009-2017.

[180] Tatjana R. Nikolić, Goran S. Nikolić, Bojan R. Dimitrijević, and Mile K. Stojcev. From single cpu to multicore systems. In *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, pages 1–8, 2022.

[181] Philipp Moritz and Robert Nishihara. Plasma In-Memory Object Store. `https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/`, August 2017.

[182] Esen A. Ozkarahan, Stewart A. Schuster, and Kenneth C Sevcik. Performance evaluation of a relational associative processor. *ACM Transactions on Database Systems (TODS)*, 2(2):175–195, 1977.

[183] Esen A Ozkarahan, Stewart A Schuster, and Kenneth C Smith. Rap: an associative processor for data base management. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 379–387, 1975.

[184] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. Exploring data analytics without decompression on embedded gpu systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1553–1568, 2021.

[185] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[186] David A Patterson and Carlo H Sequin. Risc i: A reduced instruction set vlsi computer. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 216–230, 1998.

[187] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, H Peter Hofstee, and Zaid Al-Ars. Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow. In *International Symposium on Applied Reconfigurable Computing*, pages 32–47. Springer, 2019.

[188] Johan Peltenburg, Jeroen Van Straten, Lars Wijtemans, Lars Van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277. IEEE, 2019.

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

[190] Ivan Luiz Picoli, Philippe Bonnet, and Pinar Tözün. Lsm management on computational storage. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–3, 2019.

[191] Ravindra Pindikura. Introducing the Gandiva Initiative for Apache Arrow. `https://www.dremio.com/blog/announcing-gandiva-initiative-for-apache-arrow/`, June 2018.

[192] PINE64. ROCKPro64 4GB Single Board Computer. `https://pine64.com/product/rockpro64-4gb-single-board-computer/`, April 2023.

[193] Jim Pivarski, Peter Elmer, and David Lange. Awkward arrays in python, c++, and numba. In *EPJ Web of Conferences*, volume 245, page 05023. EDP Sciences, 2020.

[194] Nicola Plowes. An introduction to eusociality. *Nature Education Knowledge*, 3(10):7, 2010.

[195] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[196] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 772–787, 2021.

[197] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4):193, 2020.

[198] David Reinsel, John Gantz, and John Rydning. The digitization of the world: from edge to core. *Framingham: International Data Corporation*, 2018.

[199] 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/`.

[200] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[201] James Richardson, Rita Sallam, Kurt Schlegel, Austin Kronz, and Julian Sun. Magic quadrant for analytics and business intelligence platforms. *Gartner ID G00386610*, 2020.

[202] Erik Riedel, Christos Faloutsos, Garth A Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[203] Erik Riedel and Garth Gibson. Active disks-remote execution for network-attached storage. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1997.

[204] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[205] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings first international conference on peer-to-peer computing*, pages 99–100. IEEE, 2001.

[206] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of 14th Python in Science Conference*, 2015.

[207] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, October 2010.

[208] Huigui Rong, Haomin Zhang, Sheng Xiao, Canbing Li, and Chunhua Hu. Optimizing energy consumption for data centers. *Renewable and Sustainable Energy Reviews*, 58:674–691, 2016.

[209] Amir Roozbeh, Joao Soares, Gerald Q Maguire, Fetahi Wuhib, Chakri Padala, Mozhgan Mahloo, Daniel Turull, Vinay Yadhav, and Dejan Kostić. Software-defined "hardware" infrastructures: A survey on enabling technologies and open research directions. *IEEE Communications Surveys & Tutorials*, 20(3):2454–2485, 2018.

[210] Robert Ross, George Amvrosiadis, Philip Carns, Charles Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel Gutierrez, Rob Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology*, 35:121–144, 01 2020.

[211] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[212] Zhenyuan Ruan, Tong He, and Jason Cong. {INSIDER}: Designing in-storage computing system for emerging high-performance drive. In *2019*

{*USENIX*} *Annual Technical Conference* ({*USENIX*}{*ATC*} *19)*, pages 379–394, 2019.

[213] Gerald Sabin and Mohammad Rashti. Security offload using the smartnic, a programmable 10 gbps ethernet nic. In *2015 National Aerospace and Electronics Conference (NAECON)*, pages 273–276. IEEE, 2015.

[214] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *2013 international conference on collaboration technologies and systems (CTS)*, pages 42–47. IEEE, 2013.

[215] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*, 1:145–164, 2016.

[216] Allen Samuels. The consequences of infinite storage bandwidth. `https://www.youtube.com/watch?v=-X9BuepxGko`.

[217] 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.

[218] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[219] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.

[220] Raj K Singh, Stephen G Tell, and Shaun J Bharrat. A programmable network interface for a message-based multicomputer. *ACM SIGCOMM Computer Communication Review*, 24(3):8–17, 1994.

[221] Ing Wilhelm G Spruth. The future of the mainframe. *New York Times*, 1989.

[222] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[223] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018.

[224] 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 SAND-2020-9451, Sandia National Laboratories, 2020.

[225] Jonathan Thatcher, Eden Kim, Dave Landsman, Marilyn Fausset, and Arnold Jones. Solid state storage performance test specification v2.0.1. Technical report, SNIA, February 2018.

[226] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[227] Brain Tierney. Experiences with 40g/100g applications, 2014.

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

[229] Hoo-min D Toong and Amar Gupta. Personal computers. *Scientific American*, 247(6):86–107, 1982.

[230] Mahdi Torabzadehkashi, Siavash Rezaei, Ali HeydariGorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. Computational storage: an efficient and scalable platform for big data and hpc applications. *Journal of Big Data*, 6:1–29, 2019.

[231] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. *ACM SIGARCH Computer Architecture News*, 44(3):53–65, 2016.

[232] Daniel Turull. Open source traffic analyzer. *Master's Thesis, KTH Information and Communication Technology*, 2010.

[233] Craig Ulmer, Jianshen Liu, Carlos Maltzahn, and Matthew L. Curry. Extending composable data services into smartnics. In *2nd Workshop on Composable Systems (COMPSYS '23), Co-located with IPDPS 2023*, Florida USA, 5 2023. IEEE. Best Paper Award.

[234] 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.

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

[236] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003.

[237] Rob van der Meulen. What edge computing means for infrastructure and operations leaders. *Gartner, online, available, www. gartner. com*, 2017.

[238] Lars van Leeuwen. High-throughput big data analytics through accelerated parquet to arrow conversion. 2019.

[239] Marek Vašut. Writing drivers for the linux crypto subsystem. *Presentation at LinuxCon Japan*, 2014.

[240] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204, 2010.

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

[242] Deepak Vohra. Apache parquet. In *Practical Hadoop Ecosystem*, pages 325–335. Springer, 2016.

[243] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

[244] Jarred Walton. First PCIe 5.0 M.2 SSDs Are Now Available, Predictably Expensive. `https://www.tomshardware.com/news/first-pcie-gen5-ssds-are-now-available`, March 2023.

[245] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 4. ACM, 2016.

[246] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019.

[247] Michael Wei, John D Davis, Ted Wobber, Mahesh Balakrishnan, and Dahlia Malkhi. Beyond block i/o: implementing a distributed shared log in hardware. In *Proceedings of the 6th International Systems and Storage Conference*, pages 1–11, 2013.

[248] Wikipedia. 10 Gigabit Ethernet — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=10_Gigabit_Ethernet&oldid=1145824511, 2023. [Online; accessed 21-April-2023].

[249] Wikipedia. 100 Gigabit Ethernet — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=100_Gigabit_Ethernet&oldid=1145168172, 2023. [Online; accessed 21-April-2023].

[250] Wikipedia contributors. Java remote method invocation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Java_remote_method_invocation&oldid=859953202, 2018. [Online; accessed 5-June-2019].

[251] Wikipedia contributors. Count key data — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Count_key_data&oldid=916249253, 2019. [Online; accessed 29-October-2019].

[252] Wikipedia contributors. List of interface bit rates — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.phptitle=List_of_interface_bit_rates&oldid=921990189, 2019. [Online; accessed 19-October-2019].

[253] Wikipedia contributors. Direct memory access — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Direct_memory_access&oldid=947273540, 2020. [Online; accessed 6-April-2020].

[254] Wikipedia contributors. Poisson distribution — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Poisson_distribution&oldid=941164666, 2020. [Online; accessed 19-February-2020].

[255] Wikipedia contributors. Posix — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=POSIX&oldid=949039157, 2020. [Online; accessed 11-April-2020].

[256] Wikipedia contributors. Printed circuit board — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Printed_circuit_board&oldid=955370206, 2020. [Online; accessed 9-May-2020].

[257] Wikipedia contributors. Scsi — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=SCSI&oldid=936426925, 2020. [Online; accessed 8-April-2020].

[258] Wikipedia contributors. Futex — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Futex&oldid=1057491809, 2021. [Online; accessed 19-January-2022].

[259] Gwyn Wischmeyer. Soaring Field Service Costs Demand Investments in Process, Technology. https://www.tsia.com/press-releases/2012/soaring-field-service-costs-demand-investments-in-process-technology, February 2012. publisher: Technology Services Industry Association.

[260] Zhongzhe Xiong. Computational Storage: Data Compression and Database Computing Pushdown. https://alibabatech.medium.com/computational-storage-data-compression-and-database-computing-pushdown-d72ff1c7dd74, April 2021.

[261] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10k ssd-related storage system failures. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 961–976, 2019.

[262] Herbert Xu. RFC: Crypto API User-interface [LWN.net]. https://lwn.net/Articles/410848/, September 2010.

[263] Jinfeng Yang, Bingzhe Li, and David J Lilja. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 5(1):1–28, 2020.

[264] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. λ-io: a unified io stack for computational storage. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, pages 347–362, 2023.

[265] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: One cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109*, 2020.

[266] Jieming Yin, Zhifeng Lin, Onur Kayiran, Matthew Poremba, Muhammad Shoaib Bin Altaf, Natalie Enright Jerger, and Gabriel H Loh. Modular routing design for chiplet-based systems. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 726–738. IEEE, 2018.

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

[268] 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.

[269] Zirak Zaheer, Hyunseok Chang, Sarit Mukherjee, and Jacobus Van der Merwe. eztrust: Network-independent zero-trust perimeterization for microservices. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 49–61, 2019.