Virtualized Low Latency Data Acquisition Software Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von der KIT-Fakultät für Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)

angenommene

Dissertation

von

M.Sc. Mostafa, Jalal
geboren in Al Nakra, Libanon

Tag der mündlichen Prüfung:	28.07.2025
Hauptreferent:	Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent:	Prof. Dr. Marc Weber

I stand on the shoulders of the giants.
To my parents. . .
To my beloved Safaa. . .
To my siblings. . .
And to every person who contributed to the man I am today
throughout history and in the present. . .

Abstract

The breakthroughs in detector and computing technologies enabled researchers to observe and measure natural phenomena with higher spatial and temporal resolutions. These improvements yield scientific data at higher rates and in larger volumes. Consequently, the data acquisition systems that are responsible for collecting and reducing the data from high rates to only interesting events have grown in complexity. Despite the extensive data reduction on hardware-based functions of data acquisition systems, researchers employ software-based data reduction functions that run on a dedicated computer cluster located at the experiment's site.

This hybrid hardware-software structure of data acquisition systems imposes critical challenges on the data acquisition system's scalability and operability. As the computing cluster of software-based data acquisition functions is deployed on the experiment's site, scaling, operating, and maintaining the stability and performance of this cluster becomes a demanding responsibility that requires extensive time and manpower. Furthermore, such a computing cluster is used to measure or simulate the observed phenomena and is not shared with other studies in the same research institute. As a result, this data acquisition structure underutilizes the available computing resources and increases the costs to operate and scale the data acquisition system.

This work aims to prioritize the design and development of software-based data acquisition functions over the operational procedure to manage its local computing cluster. This thesis proposes to eliminate this cluster for all scientific experiments in a research institute, and, instead, to run their software-based data acquisition functions on widely available general-purpose campus computing facilities. Such a shift in the execution environment minimizes the operational efforts to run a software-based data acquisition function and increases the computing resources' utilization as it shares the same resources with multiple scientific experiments while provisioning the required data transfer throughput between hardware-based and software-based functions.

To this end, this thesis proposes a new data acquisition paradigm called Data acquisition Functions Virtualization that leverages computer virtualization to isolate multiple software-based data acquisition functions running on the same host computer while sustaining their required input throughput. The thesis explores the key challenges in realizing Data acquisition Functions Virtualization within out-of-control virtualization environments like campus computing facilities and investigates which technologies are best suited for building high-throughput data transfer systems in these settings.

To realize Data acquisition Functions Virtualization, we propose, design, and implement the Data Acquisition Development Kit, a novel framework for software-based functions. The framework handles high-performance data transfer between detector electronics and campus computing facilities by exploiting an emerging networking technology called AF_XDP and by adapting resource allocation to the shared execution environment of general-purpose computer systems used in campus computing facilities. We quantify the performance of the framework with and without computer virtualization, considering different computer virtualization setups. The new data acquisition paradigm is applied to the TRISTAN upgrade of the KATRIN experiment at the Karlsruhe Institute of Technology. The framework can reduce CPU resources by a factor of 2.67x and save up to 15% of the consumed energy.

Kurzfassung

Dank fortschrittlicher Detektor- und Computertechnologien können Forscher natürliche Phänomene heute mit deutlich höherer räumlicher und zeitlicher Auflösung untersuchen. Dadurch entstehen wissenschaftliche Daten in bislang unerreichtem Umfang und mit hohen Erfassungsraten. Um aus dieser Datenflut nur die relevanten Ereignisse herauszufiltern, sind Datenerfassungssysteme zunehmend komplexer geworden. Neben der bereits umfangreichen hardwarebasierten Datenreduktion kommen zusätzlich softwaregestützte Funktionen zum Einsatz, die auf einem speziellen Rechnercluster direkt am Experimentstandort betrieben werden.

Diese hybride Struktur aus Hardware- und Softwarekomponenten stellt hohe Anforderungen an die Skalierbarkeit und Wartbarkeit von Datenerfassungssystemen. Da die softwarebasierten Funktionen auf einem lokalen Rechnercluster laufen, erfordert der Betrieb dieses Clusters erheblichen personellen und zeitlichen Aufwand – insbesondere, wenn es darum geht, Leistung und Stabilität dauerhaft sicherzustellen. Hinzu kommt, dass solche Cluster in der Regel ausschließlich für ein einzelnes Experiment genutzt werden und nicht mit anderen Forschungsprojekten innerhalb der Einrichtung geteilt werden. Das führt zu einer ineffizienten Nutzung der vorhandenen Rechenressourcen und erhöht sowohl die Betriebskosten als auch den Aufwand für eine spätere Skalierung.

Ziel dieser Arbeit ist es, den Fokus von der aufwendigen Verwaltung lokaler Rechnercluster hin zur Entwicklung und Ausführung softwarebasierter Datenerfassungsfunktionen zu verlagern. Anstatt für jedes Experiment eigene Cluster zu betreiben, wird vorgeschlagen, diese Funktionen auf allgemein verfügbaren, zentral betriebenen Rechensystemen innerhalb der Forschungseinrichtung auszuführen.

Ein solcher Paradigmenwechsel reduziert den betrieblichen Aufwand erheblich und verbessert gleichzeitig die Auslastung der vorhandenen Rechenressourcen, da mehrere Experimente dieselbe Infrastruktur gemeinsam nutzen können. Voraussetzung dafür ist allerdings, dass der erforderliche Datendurchsatz zwischen hardware- und softwarebasierten Komponenten zuverlässig gewährleistet wird.

Um dieses Ziel zu erreichen, wird in dieser Arbeit ein neues Paradigma vorgeschlagen: Virtualisierung softwarebasierter Funktionen der Datenerfassung. Dabei werden softwarebasierte Datenerfassungsfunktionen mithilfe von Virtualisierung voneinander isoliert, sodass mehrere dieser Funktionen gleichzeitig auf einem gemeinsamen Host-System betrieben werden können – ohne den erforderlichen Datendurchsatz zu beeinträchtigen. Die Arbeit untersucht die zentralen Herausforderungen bei der Umsetzung dieses Ansatzes in offenen, schwer kontrollierbaren Virtualisierungsumgebungen wie etwa campusweiten Rechenzentren. Zudem wird analysiert, welche Technologien sich am besten für eine leistungsfähige und zuverlässige Datenübertragung in solchen Szenarien eignen. Zur praktischen Umsetzung der Virtualisierung von Datenerfassungsfunktionen wird in dieser Arbeit das Data Acquisition Development Kit vorgestellt – ein neuartiges Framework für softwarebasierte Datenerfassungsfunktionen. Es ermöglicht eine performante Datenübertragung zwischen Detektorelektronik und zentralen Rechensystemen der Forschungseinrichtung, indem es auf die moderne Netzwerktechnologie AF_XDP zurückgreift und die Ressourcenzuweisung dynamisch an die gemeinsam genutzte Ausführungsumgebung anpasst. Die Leistungsfähigkeit des Frameworks wird unter verschiedenen Virtualisierungsbedingungen sowohl mit als auch ohne Virtualisierung systematisch evaluiert. Anwendung findet das neue Paradigma im Rahmen des TRISTAN-Upgrades des KATRIN-Experiments am Karlsruher Institut für Technologie. Dabei zeigt sich, dass das Framework bis zu 2,67-mal weniger CPU-Ressourcen benötigt und den Energieverbrauch um bis zu 15% senken kann.

Acknowledgements

This thesis would not have been possible with the support and guidance from many people at IPE and ITIV institutes at KIT. I want to thank my supervisors Prof. Jürgen Becker and Prof. Marc Weber for their guidance, feedback, and support throughout this journey. I am grateful to my KSETA mentor Prof. Guido Drexlin for supporting this thesis until the end. I also thank Prof. Frank Simon for his support and guidance during my time at IPE.

I thank Suren Chilingaryan and Andreas Kopmann for giving me the opportunity to be part of the IPE family. Their time they spent to support and guide this thesis and all the fruitful discussions are highly appreciated. Andreas, Suren, I am deeply grateful for your mentorship. You are the best! To my colleagues, Timo Dritschler and Nicholas Tan Jerome, thanks a bunch! Special thanks to Tanya Harbaum from ITIV for her support and time. I appreciate your assistance!

To my students: Sara Wehbe, Sandro Melissano, and the others, thank you for being part of this journey and the time we spent learning together.

I cannot but mention my family again whom I dedicate this thesis to. They are the real contributors to this thesis. For my father and my mother, I can't thank you enough. For my wife, Safaa, for her endless support during the PhD time and always, I'm truly grateful. For my eldest brother, Jaafar, for igniting the love of technology in me since I was a kid. For all of my brothers and sisters: Jaafar, Wael, Waed, and Israa, you have contributed to the person I am today and thus to this thesis, thank you! For you all, whether you know it or not, this thesis is *your* achievement.

Contents

1 Introduction

1.1 Motivation

A scientific detector in a scientific research infrastructure consists of one or more electronic sensors that capture natural interactions with physical energy or particles in a scientific phenomenon to acquire data necessary for testing a scientific hypothesis. The data acquisition (DAQ) process is a complex online process that, during the detector operation, samples electronics signals from the scientific detector, reduces data volumes and rates by removing noise and extracting relevant data samples or regions of interest (RoI), and finally stores the data permanently for further later processing.

The breakthroughs in electronic technologies have endorsed scientists efforts to design novel scientific detectors capable of observing scientific phenomena in unprecedented details. These breakthroughs have allowed scientists to increase the measurement's spatial and the temporal resolutions leading to an increase in detectors' data volumes and rates. For example, the raw data rates of the Compact Muon Solenoid (CMS) experiment at the European Organization for Nuclear Research (known as CERN) have developed from $100\,\mathrm{GB/s}$ by end of 2013 to $200\,\mathrm{GB/s}$ by end of 2018 [5]. Doubling the data rates in 5 years only! Such detectors yield an everly growing data throughput that can reach several hundreds of gigabits per second [1], [6]. The data rate of the CMS experiment, for instance, is expected to grow to $50\,\mathrm{Tbit/s}$ with the High Luminosity Large Hadron Collider (HL-LHC) [7].

To handle the increasing data rates, scientists design hierarchical hybrid hardware-software data acquisition functions (DFs) for data reduction. Fig. 1.1 shows an

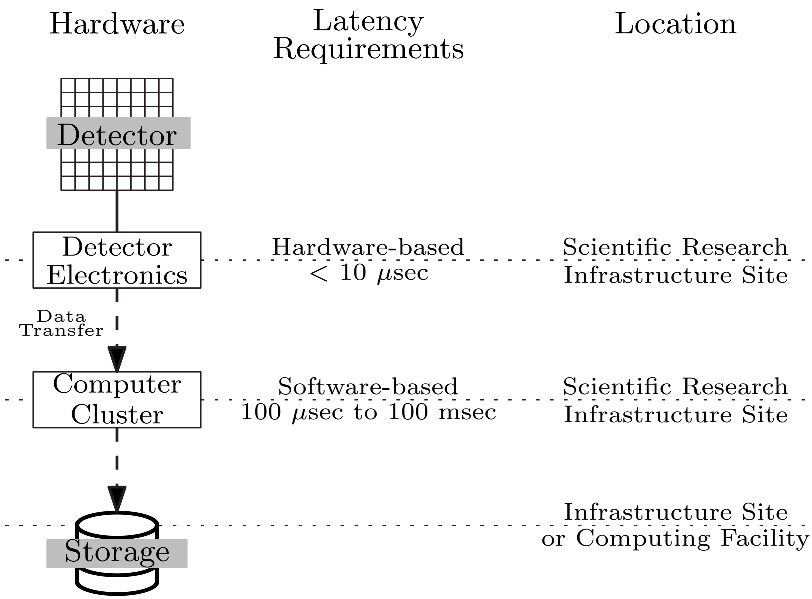


Figure 1.1: Overall Model of the Hybrid Hardware-Software DAQ

overall architecture of data reduction functions in DAQ systems. The hybrid hardware-software approach forms a compromise between performance and flexibility. Detector electronics are deployed close to the detector on the scientific research infrastructure's site and employ hardware technologies that can perform ultra-low-latency data reduction ($< 10\,\mu s$). Examples of such technologies are Field Programmable Gateway Arrays (FPGA) and Application–Specific Integrated Circuits (ASICs). Because of their long test and development cycles, these technologies consume scientists' efforts and time to design and test data reduction functions. After initial data reduction performed by hardware-based data reduction functions, scientists run software-based functions for further data reduction on computer clusters that are operated on the research infrastructure site. Contrary to hardware-based functions, software-based ones are easier to design and test complex reduction and feature extraction algorithms using high-level programming languages, e.g. C or Python, and parallel processing hardware and software frameworks, e.g. General Purpose Graphical Processing Units (GPGPU) and Message Passing Interface (MPI). An example of software-based functions is High-Level Triggers (HLT) that are commonly used in several scientific research infrastructures like CMS [8] and ATLAS [9]. The transition from hardware-based to software-based functions includes data transfer of detector data from detector electronics to computer clusters. The data transfer is either performed over computer networks, e.g. Ethernet networks, or over computer buses, e.g. Peripheral Component Interconnect Express (PCIe).

1.1.1 The Operability Challenge

The hybrid hardware-software architecture imposes what we call the Operability Challenge. Since computer clusters of software-based data reduction functions are located at the scientific research infrastructure site, scientists have to put increasing efforts to manage and operate this computing infrastructure. The following attributes of computer clusters enforce the operability challenge:

Active System Maintenance Computer clusters of software-based data acquisition functions are composed of few to hundreds of computer systems which host and run diverse software middlewares and packages in addition to an operating system (OS). To remain fully operational and secure, these systems require active system maintenance that includes adequate system configuration and administration, troubleshooting software middleware problems, updating software packages, etc. Without such measures, the computer cluster may fail to run data acquisition functions impacting their availability, and it can get vulnerable to software security issues that may jeopardize the infrastructure of scientific instrumentation.

Single-Purpose Under-utilized Clusters Since these computer clusters are located at the research infrastructure site and protected by its security constraints, other research infrastructure cannot access them. They solely belong to one specific research infrastructure or scientific experiment and are sporadically utilized when needed, e.g. a scientific measurement is being performed, or run some simulations. This may lead to under-utilization: each research infrastructure buys its own expensive hardware for the computer that is sitting idle for a considerable portion of time.

Every Research Infrastructure Operates Its Own Computer Cluster In a scientific or an academic institute, scientists usually host and operate more than one scientific research infrastructure. For example, Karlsruhe Institute of Technology simultaneously operates the Karlsruhe Tritium Neutrino (KATRIN) experiment, the Karlsruhe Research Accelerator (KARA), Far-infrared Linac and Test Experiment (FLUTE), etc., in one campus in Karlsruhe, Germany. However, every research infrastructure of them operates its dedicated computer cluster and dedicates manpower to maintain this cluster. This may lead to redundant efforts and increased costs. Contrarily, scientists should unite their computing power in shared computer clusters to minimize efforts and reduce costs.

Low Scalability A single unit in a computer cluster is a computer system. The performance of a computer cluster is limited by the number of its computer systems. Scaling the cluster for higher performance than its capabilities requires increasing the number of its computer systems – a complex lengthy process which involves procuring, installing, and administering of the additional computer systems.

1.2 Research Proposal and Questions

In this thesis, we target overcoming the operability challenge of DAQ systems by rebuilding the hybrid hardware-software architecture for better scalability and performance. We propose **D**ata acquisition **F**unctions **V**irtualization (DFV), a new DAQ paradigm that eliminates the need to operate a computer cluster in a research infrastructure. The DFV paradigm, instead, exploits computer virtualization to run software-based DFs on campus computing facilities that are abundantly available for common use in most scientific and academic institutes. The main goal of this paradigm is to keep scientists' focus centered on the design of data acquisition functions and shift their efforts away from the operation of a local computer cluster by relying on the services provided by the shared computing facilities in an institute.

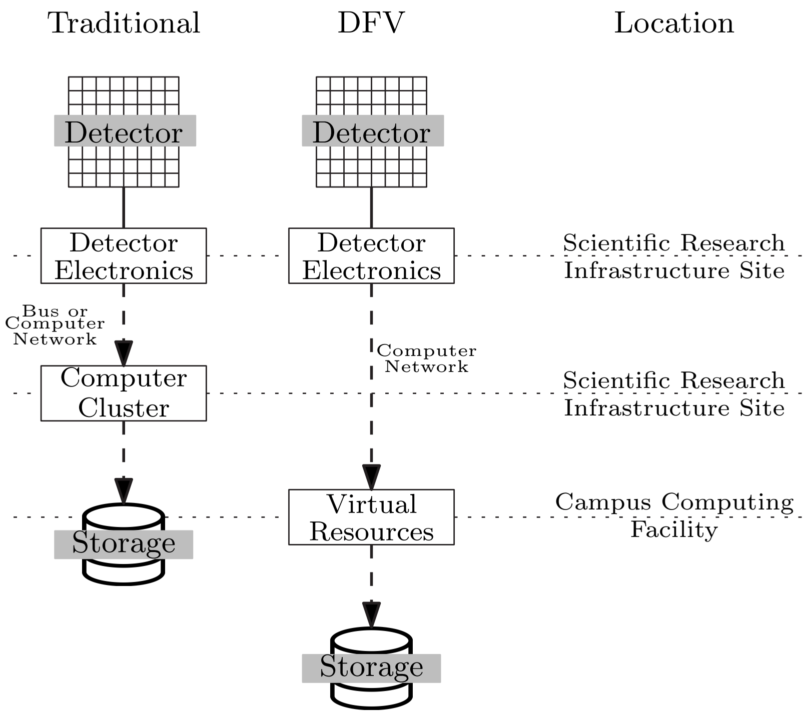


Figure 1.2: DFV vs Traditional Hybrid Hardware-Software DAQ

Fig. 1.2 shows the architecture of DFV in comparison to traditional hybrid hardware-software DAQ. In DFV, detector electronics located at the research infrastructure site still perform hardware-based DFs, but they transfer detectors' data to virtual computing resources located in campus computing facilities over computer networks. DFV does not enforce major changes on hardware-based DFs but only requires an implementation of the used data transfer protocol to be interoperable with the software-based DFs. Since DFV can co-exist with diverse non-DAQ applications like web applications or file servers on the premises of campus computing facilities, it is important that they do not compromise each other's execution environments. DFV utilizes computer virtualization technologies for this purpose. They provide software-defined virtual execution environments that ensure peaceful coexistence of diverse applications by isolating each application in a sandbox that cannot access other sandboxes.

The shift to virtual computing resources on campus computing facilities have significant advantages in regard to the operability challenge compared to the traditional hybrid DAQ paradigm. Campus computing facilities are highly-scalable shared reusable computing resources. Therefore, using them instead of local computer clusters eliminates the efforts to operate a dedicated computing cluster in every research infrastructure and offloads system maintenance to the computing department of the owner institute. Multiple research infrastructures and scientific experiments can consequently use the computing resources of campus computing facilities at the same time which increases the utilization of computing resources in these facilities. Computer virtualization and their abundant computing resources provide easy means to scale DFs on demand without procuring additional hardware. Additionally, computer virtualization employ automatic software-defined mechanisms to support the DFs' availability which would reduce scientists efforts to monitor and track DFs' availability.

DFV runs on general-purpose computer systems in campus computing facilities. These systems can be black boxes of hidden hardware architectures, configurations, and software middlewares. Therefore, DFV should be resilient to adapt to diverse state-of-the-art technologies used on campus computing facilities.

The cornerstone and main challenge of DFV is zero-loss data transfer of detector's data from detector electronics to virtual computing resources in campus computing facilities over computer networks. This is important to ensure the integrity of scientific data in research infrastructure [10]. Without reliable data transfer between DFs, scientific research infrastructure is exposed to probable data loss on the way from detector electronics to virtual computing resources in campus computing facilities rendering the whole infrastructure defective. A major reason for data loss is the *Slow Receiver* problem where the software-based DF, the receiver in our case, drops detector data because it cannot process the stream fast enough due to high data transfer latency [11]–[13]. While there exists some computer networking technologies of low data transfer latency to solve the *Slow Receiver* problem, not all of them might be suitable for campus computing facilities and DFV. Therefore, we raise the following question:

> *Question 1: What are the qualifications to choose a suitable networking technology for Campus Computing Facilities and DFV?*

Based on these qualifications, we raise the second question in this thesis:

> *Question 2: What networking technology is a good compromise between performance and these qualifications to realize DFV?*

Unlike traditional hybrid hardware-software DAQ, campus computing facilities share resources among diverse applications. It is probable and likely that these applications might interfere with each. In addition, general-purpose computer systems in campus computing facilities are not tuned and optimized for applications that demand low-latency like DFV. This brings us to raise to the following question in this thesis:

> *Question 3: What are the sources of higher data transfer latencies in general-purpose computer systems of campus computing facilities?*

Scientists in research infrastructures usually lack the technological expertise to tackle such complex technical optimizations. Therefore:

> *Question 4: How can we autotune a general-purpose computer system for DFV without significant intervention from scientists?*

Computer virtualization is an integral component of DFV. It protects DFs from unwanted malicious accesses from other applications. However, security comes in for a performance cost in computer systems. Therefore, we raise the question:

Question 5: What is a computer virtualization model that can handle hundreds of gigabits per second in throughput?

1.3 Objectives and Contributions

In this thesis, we argue that conventional data transfer technologies used in commodity computer networks are not reliable to realize DFV. An alternative reliable data transfer technology is required to realize DFV. Therefore, we will define the qualifications that would make a data transfer technology suitable for DFV as raised in Question 1. Based on these qualifications, we will survey available data transfer technologies, and we assess their performance to nominate one for DFV as discussed in Question 2. This thesis will also look at the challenges raised by deploying DFs on virtual computing resources on shared general-purpose computer systems like that of campus computing facilities in comparison to local computer clusters like that of the traditional hybrid architecture. We will study these challenges and propose the adequate tuning and mitigations as discussed in Questions 3 and 5. A framework will be proposed to provide easy-to-use DFV-based DFs without significant intervention from scientists to answer Question 4.

1.4 Outline

The rest of the thesis is organized as follows:

Chapter 2 will discuss the required preliminaries and background for a full understanding of this thesis. In the direction of DAQ systems, we will specifically discuss their overall hardware-software architecture, the anatomy and the reliability of a software-based DF, and finally we provide examples of traditional hybrid hardware-software DAQ architecture from real scientific infrastructures and experiments. The chapter also discusses the basics of conventional data transfer and how received data is processed in commodity networks from the point it reaches the hardware of a computer system to the point is ready to be processed by the user application. We also compare DFV to state-of-the-art efforts to incubate computer virtualization and high-performance data processing in scientific research infrastructure and experiments.

Chapter 3 inducts a survey of computer networking technologies to perform reliable data transfer. The chapter discusses the criteria that qualify a computer networking technology for DFV from the perspectives of computer virtualization and the requirements of DAQ systems. Based on these criteria, we choose a computer networking technology called AF_XDP. The rest of this chapter reasons why AF_XDP is suitable for DFV in terms of performance and maturity and in comparison to other conventional and high-performance networking technologies.

Chapter 4 studies the challenges of running a software-based DF on shared general-purpose computer systems. It discusses the problems that may rise due the computer and the memory architecture of these systems and measures their impact on performance. We will propose mitigations to tune and optimize the performance in these systems through a software framework called the Data Acquisition Development Kit in order to realize DFV.

Chapter 5 quantifies the performance impact of computer virtualization setups on DFV. We study hardware-based and software-based network virtualization for both virtual machines and containers as computer virtualization technologies.

Chapter 6 applies DFV on the TRISTAN detector as a use case of a hybrid hardware-software DAQ architecture. The chapter will introduce the TRISTAN detector, its operational modes, and its requirements. We will provide insights on DFV in real-life use cases through the TRISTAN detector.

Chapter 7 proposes a new methodology for monitoring detector systems. It focuses on improving the performance of storing and accessing process data that usually monitors a detector DAQ system to ensure its robust operation. The chapter looks at time-series databases as an alternative to state-of-the-art ACID-based databases. We propose a benchmark named SciTS to evaluate different databases to manage process data and consequently propose a novel monitoring system for scientific infrastructure.

Chapter 8 concludes this thesis with insights about DFV and its applicability to real-world scientific infrastructure and experiments. The chapter discusses the limitations of our contributions. It opens horizons for future works that can support DFV in particular those related to new computer and memory architectures like Smart Network Interface Cards and GPGPUs.

2 Background and Related Work

2.1 Introduction to Data Acquisition and Control Systems

A data acquisition and control system manages the data flow in research infrastructures and scientific experiments. It collects data from its sources, e.g. electronic components, makes it available to the scientists through computer storage, and preserves its integrity during the collection process.

For the successful operation of a research infrastructure, scientists install heterogeneous data sources, e.g. one or more detectors and other sensors of heterogeneous technologies. This produces different data forms and requirements inside a research infrastructure e.g. detector readout data, fast control data, and slow control data. All are important to the operation and experimental integrity of the scientific experiment and should be permanently stored to ensure the successful operation of a scientific experiment.

We can arrange a data acquisition and control system into 3 logical systems: the readout system, the fast-feedback control system, and the slow control system. We explain each logical system architecture and requirements by inspecting the data flow in a research infrastructure. Fig. 2.1 shows the data flow in a data acquisition and control system.

Detecting and measuring a natural phenomenon is performed by a system called the detector. A detector is a system of electronic sensors of physical properties that react to interactions with nature under certain circumstances by producing electromagnetic signals. These signals represent the interactions with nature in

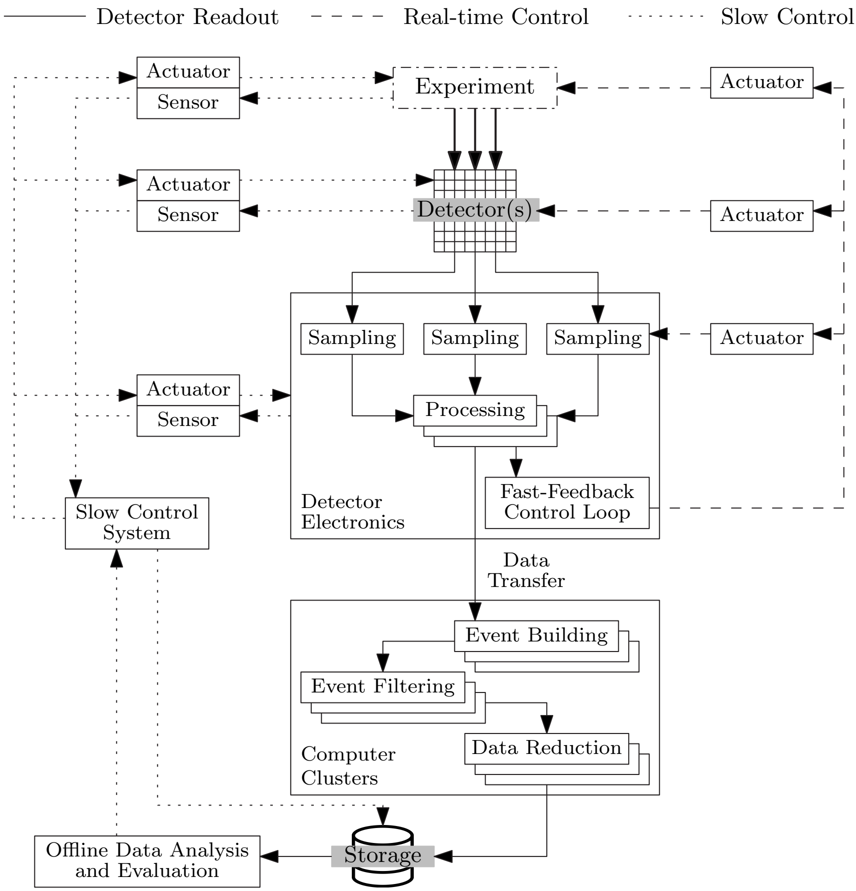


Figure 2.1: Data Flow in Data Acquisition and Control Systems

addition to accumulated noise throughout the measurement procedure. Without collecting and processing these signals, we cannot derive meaningful scientific conclusions. Detector electronics are a group of electronic circuits systems that sample and convert the detector's electromagnetic signals from analog to digital signals. They also perform the required data processing to extract the needed information from the detector's signals. This type of data is called detector readout. Detector electronics are implemented using low-latency electronics e.g.

Field-Programmable Gate Array (FPGA), Application-Specific Integrated Circuit (ASIC), etc. Since these electronics implement complex data processing algorithms, their resources are considered scarce where efficient algorithms implementations are resource-efficient ones [14]–[17]. In addition, modern detector systems generate enormous amounts of data volumes and rates due to their high spatial and temporal resolution. For example, the data rates in the CMS experiment at CERN are expected to grow from $100\,\mathrm{GB/s}$ by end of 2013 to $200\,\mathrm{GB/s}$ by end of 2018 [5], and they are expected to grow up to $6.25\,\mathrm{TB/s}$ in the next few years [7]. Therefore, scientists employ a hybrid hardware-software DAQ architecture which provides high flexibility and extensible computing resources. Further data processing can be performed on a computer cluster to implement resource-demanding algorithms like data compression and feature extraction algorithms. Examples of tasks performed in a computer cluster are: event building, event filtering, and data reduction. Event building is the process where one or more digital signals from detector electronics are constructing in an event that represent the natural phenomenon, e.g. particles' collision in a particle accelerator. Constructing events is followed by filtering interesting events using an algorithm that is specific to the goal of the research infrastructure. Data reduction helps reduce and compress the data even further if needed. Examples of such a computer cluster in research infrastructure are high-level triggers (HLT) and event filtering farms. This computer cluster use high-performance computing technologies like the Message Passing Interface (MPI) and General-Purpose Graphical Processing Units (GPGPU) programming. The data transfer between detector electronics and a computer cluster is performed over computer networks, e.g. Ethernet, computer buses, e.g. Peripheral Component Interconnect Express (PCIe), or serial links. The output data of this computer cluster is stored on persistent computer storage for the subsequent offline data analysis process.

Some research infrastructures require real-time feedback from detector electronics to the detector and other subsystems to control certain parameters of the experiment. An example is controlling micro-bunching instabilities in synchrotron light sources [17]. Due to their real-time constraints, such control systems are implemented on ultra-low latency detector electronics. A mechanism, known as the Fast-feedback Control Loop, generates a control decision based on the processed data available from other detector electronics components. The generated decision is then implemented using actuators to change specific parameters in the detector electronics, the detector itself, or other subsystems of the research infrastructure.

The command center of a research infrastructure is the slow control system. It has 3 main tasks: (1) to control subsystems with *no* real-time constraints, e.g. power supplies, temperatures, high voltages, magnetic fields, etc. across the research infrastructure; (2) to collect data from sensors with slow data rates and volumes, e.g. temperature, magnetic fields, etc.; (3) to make the data available to scientists through visualization software and computer storage; (4) and to provide human-machine interfaces to initiate subsystems and start the operation of the research infrastructure.

This thesis focuses on improving the data flow of detector readout. It studies alternatives of using a computer cluster through employing campus computing facilities.

2.1.1 Anatomy of a Software-based Data Acquisition Function

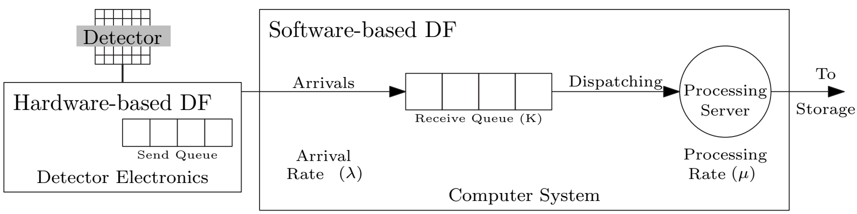


Figure 2.2: A Conceptual Model for Software-based Data Acquisition Functions

A software-based DF is a queuing-based system. Its conceptual model is shown in Fig. 2.2. After detector electronics complete the execution of hardware-based DF, processed detector data is transmitted to the software-based DF running on a computer system. The transmission process involves submitting chunks of detector data to the *Send Queue* in the hardware-based DF that is responsible for managing transmission operations. Detector data exiting the *Send Queue* are dispatched to the data transfer medium, e.g. bus or computer network. The data transfer medium delays the arrival at the software-based DF to complete the transmission process, we denote this by $\tau_{transfer}$. The data arrival at the software-based DF is measured through the *Arrival Rate* denoted by λ. A software-based DF controls the *Receive Queue*, a queue of constant size K to buffer detector's data until it gets its turn for data processing. The end goal of a software-based DF is to execute its data reduction logic. Therefore, data is first *dispatched* from the *Receive Queue* by the processing server. The dispatching process also involves latency because the data passes through a medium from the *Receive Queue* to the processing server, e.g. copying from one memory buffer to another, we denote this latency by $\tau_{dispatching}$ The processing server executes the data reduction algorithm, e.g. a feature extraction algorithm. Processing latency, denoted by $\tau_{processing}$, is defined by the time taken to finish the execution of a single algorithm run and therefore the processing rate is defined by:

$$\mu = \frac{1}{\tau_{processing} + \tau_{dispatching}} \tag{2.1}$$

The total latency from the moment detector's data exits the detector electronics to the moment it is ready for departure from the processing the server in the software-based DF can be calculated as follows:

$$\tau = \tau_{transfer} + \tau_{dispatching} + \tau_{processing} \tag{2.2}$$

According to queuing theory, to prevent data loss the following equation has to be satisfied:

$$\mu \geq \lambda \tag{2.3}$$

This property is defined as the stability of the queuing system. If the system gets unstable, i.e. $\mu < \lambda$, the software-based DF starts to lose data. The system's stability can then be achieved through scaling the system. There are

3 ways to scale the system: (1) increase queue size to provide a larger buffer that can accommodate more incoming received data while waiting their turn for processing; (2) increase the processing rate μ through multiple processing servers that concurrently dispatch from the queue; (3) decrease dispatching latency from the queue which increases the processing rate.

2.1.2 Examples of Hybrid Hardware/Software Data Acquisition in Research Infrastructure

For better understanding of this thesis, we provide three examples of renown research infrastructures that employ a hybrid hardware-software DAQ architecture.

2.1.2.1 DUNE

The Deep Underground Neutrino Experiment (DUNE) is an international experiment aiming to explain the origin of matter, the Grand Unification Theory, and formation of neutron stars and black holes by studying neutrinos, the most abundant particle in the universe [18]. DUNE runs 2 neutrino detectors in a beam of intense neutrino beam. The first detector (or the near detector) will record particle interactions near the neutrino beam source at the Fermi National Accelerator Laboratory in Batavia, Illinois, United States. A larger detector (or the far detector) will be deployed 1300 kilometers away from the neutrino beam and more than 1 kilometer underground at the Sanford Underground Research Laboratory in Lead, South Dakota, United States.

To simplify construction and operational efforts of the far detector DAQ, it will adopt a hybrid hardware-software DAQ architecture. The current far detector prototype at CERN divides the DAQ process into 2 components: hardware-based DAQ and software-based DAQ. Both components of the prototype will be interconnected through standard Ethernet protocols e.g. User Datagram Protocol (UDP) over a $100\,\mathrm{Gbit/s}$ link. The far detector electronics will reduce $\approx 15\,\mathrm{Tbit/s}$ to a $200\,\mathrm{Gbit/s}$ waveform. The software-based DAQ component will take the $200\,\mathrm{Gbit/s}$ waveform as its input. It exploits Single Instruction Multiple Data (SIMD) parallel instructions on x86 computer systems to perform online reduction of electronic noise. The main component, however, is the high-performance data transfer that is performed through the Data Plane Development Kit, a third-party software library for high-performance networking technologies. Future work is scaling this software system to $400\,\mathrm{Gbit/s}$ detector readout [19].

2.1.2.2 CMS at CERN

The CMS is a scientific experiment at CERN to look for evidence of physics beyond the Standard Model in physics by recording collisions of heavy ions at the Large hadron Collider (LHC) particle accelerator. Its notable achievement is the discovery of the Higgs-Boson particle. CMS adopts a hybrid hardware-software DAQ architecture: CMS's detector electronics perform a front data reduction system (better known as the Level-1 Trigger System) and pass its output to a computer cluster known as the High Level Trigger (HLT) and neighboring the detector at Point 5 in the LHC [20]. The HLT performs event reconstruction and filtering for particle identification and collisions [8].

The CMS experiment has been performed in runs of 2-3 years with long shutdowns in between for major detector and hardware upgrades. Throughout the CMS's runs, its data rates and volumes has enormously increased. In Run 1 (2011-2012), the HLT at CMS was designed to accept $100\,\mathrm{kHz}$ $1\,\mathrm{MB}$-events ($100\,\mathrm{GB/s}$), its output rate to persistent storage was $\approx 1.2\,\mathrm{GB/s}$ [8]. The HLT at Run 1 has 13000 CPU cores and was interconnected to Level-1 Trigger system using serial links and a Myrinet network [5]. Run 2 (2015-2017) increased the input data rate to $200\,\mathrm{GB/s}$ because the event size increased to $2\,\mathrm{MB}$, the output data rate of the HLT has also increased to $\approx 3\,\mathrm{GB/s}$. The number of CPU cores also increased to 16000 interconnected to the detector electronics using BSD Sockets over 10GbE and 40 GbE Ethernet [5], [21]. The input rate of Run 3 (2022-2024) remained the same in total, however the HLT algorithm has changed requiring higher performance and so has the HLT itself. Its output rate increased to $\approx 5.5\,\mathrm{GB/s}$. The number of CPUs in the HLT increased to 25600 cores and the HLT was augmented with 400 NVIDIA T4 GPGPUs for the first time [22], its interconnects to the detector electronics were also upgraded to 100 GbE Ethernet [23]. Run 3 ended in 2024 and scientists at CERN are preparing for the High-Luminosity LHC upgrade which will increase the data rates to unprecedented values. It is expected that CMS detector in Run 4 and Run 5 will reach up to $750\,\mathrm{kHz}$ events of size up to $6\,\mathrm{MB}$ yielding up to $6.25\,\mathrm{TB/s}$ as input data rate to HLT, its output rate will reach up to $\approx 31\,\mathrm{GB/s}$ [7].

2.1.2.3 Belle II

The Belle II experiment is a particle physics experiment deployed at the SuperKEKB particle accelerator in Tsukuba, Ibaraki prefecture, Japan. Its goal is to study the properties of B mesons particles. The Belle II infrastructure is formed of mainly the pixel detector and 6 secondary detectors. The event rates of the secondary detectors can reach up to $200\,\mathrm{MHz}$. To overcome this high event rate, it employs a hybrid hardware-software DAQ architecture. The detector electronics reduce the event rates of the Belle 2 secondary detectors to $30\,\mathrm{kHz}$ at maximum each of $100\,\mathrm{kB}$ in size. This is fed to the Belle 2 high-level trigger (HLT) which will reduce the event rates to $10\,\mathrm{kHz}$ each of $200\,\mathrm{kB}$ in size by reconstructing unpacked data from the secondary detectors into new events. The new events are then filtered and used to provide tracking information to calculate regions of interests in main detector data. The HLT employs $\sim$6000 CPU cores in 2024 and interconnects with detector electronics through a third-party software library called ZeroMQ [24].

2.2 Introduction to Data Transfer in General-Purpose Computer Systems

DFV's cornerstone is efficient data transfer in general-purpose computer systems of campus computing facilities. Therefore, it is important to study this process including any inefficiencies.

Data transfer between distributed computer systems is a complex process that involves the participation of the hardware, the operating systems, and user software of computer systems to complete the task.

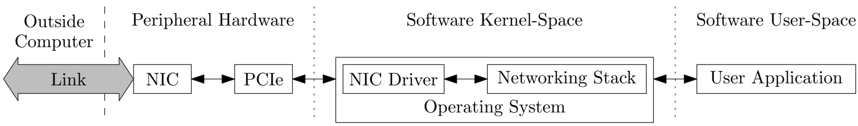


Figure 2.3: Data Transfer Flow inside a Modern Computer System

Fig 2.3 shows the data flow of the data transfer process and the main involved components. A modern computer system has a network interface card (NIC) connected as a peripheral device to the computer system from one side and to other computer systems through a physical link. The physical link employs a physical medium of predefined capacity (or bandwidth) that carry the signal from one computer system to another, e.g. Twisted Pairs, Twinaxial, Fiber Optics, etc. The computer's CPU and NIC communicate via a bus, e.g. modern computer systems extensively use the Peripheral Component Interconnect Express (PCIe) bus. However, this is not sufficient to use and interact with the NIC hardware. The operating system of this device runs an implementation of the NIC driver which defines methods to communicate with the NIC hardware and integrates it within the operating system (OS) of the computer system. The operating system also runs a networking subsystem, called the networking stack, that runs common logic that sits between the operating system and the user application. Components of the networking stack could be: data transfer protocols to reliably move data between heterogeneous computer systems, e.g. the Internet Protocol (IP) or the User Datagram Protocol (UDP), hardware interrupt management, application programming interfaces (APIs), etc. Both the NIC driver and the networking stack runs in a high privileged environment that have direct unmanaged access to all hardware components, this layer is called the operating system kernel-space. Contrarily, the user application interacts with the networking stack through its APIs and runs in an unprivileged execution environment that can only access the NIC hardware by delegating the process to the operating system and its internals.

The rest of this section will discuss each of the components involved in the data transfer process and provide insights about measuring their performance.

2.2.1 Key Performance Indicators

The performance of data transfer in general-purpose computer systems is measured as whole due to the interference of all components involved in this process. In this section, we focus on 3 performance indicators that summarize the overall performance of the data transfer system:

Throughput is a measure of how much data can pass through the data transfer system in a second. In digital data transfer systems, throughput is measured in bits per second. In practical applications, real throughput does not reach the full capacity (the bandwidth) of the physical link. This is due to overhead by data transfer protocol and line encoding used for synchronization between the sender and the receiver.

Round-Trip Latency (RTT) is a measure of how much time it takes to send a data request to a computer system and then receive a response back. Round-trip latency includes the time taken to traverse all the components of Fig. 2.3 4 times (sending request, receiving request, sending response, receiving response) in addition to the time taken to process the request and issuing a suitable response. It is measured in milliseconds or microseconds.

One-way Latency is a measure of how much processing time does a received chunk of data take from the moment it reached the NIC hardware to the moment it is available for processing by the user application. In other words, it is the time taken for a received chunk of data to traverse all the components in Fig. 2.3. In this thesis, we also call it dispatching latency. Dispatching latency is usually measured in microseconds.

2.2.2 Peripheral Component Interconnect Express

Peripheral Component Interconnect Express (PCIe) is as a data transfer bus that connects peripheral devices like NICs to the CPU where the device driver, interrupt handlers, the networking stack, and further data processing are executed. PCIe's design supports any peripheral device that implements its protocol, and therefore it is not limited to NICs, but it can also accommodate storage and GPU devices among others. However, it is important to remark that PCIe is a different interconnect from PCI, its predecessor, with varying speeds and features. For example, PCIe uses a serial bus architecture compared to the shared parallel architecture of PCI. This section explains PCIe architecture, protocol, and its mechanisms to perform data transfer between the NIC and the host computer.

2.2.2.1 PCIe Architecture

PCIe connects PCIe endpoints, i.e. peripheral devices, to the CPU and main memory of the host computer using a point-to-point connection. Fig. 2.4 shows the architecture of PCIe. A *PCIe Domain* consists of a single *Root Complex*, main memory, a processor, and PCIe-compatible devices. The *Root Complex* establishes PCIe connection by generating requests on behalf of the CPU which is interconnected over a local bus. The Root Complex can connect more than one PCIe endpoint, each over a direct interconnect to the Root Complex or by cascading multiple PCIe endpoints over a PCIe switch. A PCIe device's access to main memory is configured through a process called *enumeration* which builds a device memory map for the operating system by querying each PCIe endpoint device connected to the Root Complex. During this process, the operating system of the host computer assigns configuration tables to set up a map of accessible memory addresses of each PCIe device. There are types of configuration tables: *Type 0 Configuration Table* and *Type 1 Configuration Table*. The *Type 1 Configuration Table* is owned by the host operating systems and contains a map of memory addresses for all available PCIe devices. On the other hand, *Type 0*

Configuration Table is owned by the PCIe device consisting from a copy of the memory map corresponding to this PCIe device.

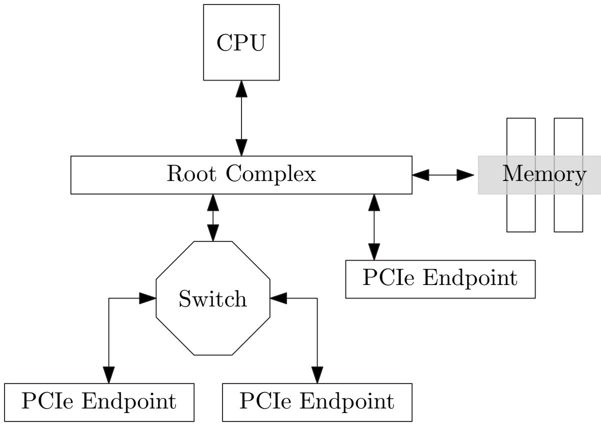


Figure 2.4: The Architecture of a PCIe Domain

Unlike PCI, PCIe performs all point-to-point interconnects using dedicated serial buses. This means every PCIe endpoint is connected to Root Complex on separate serial links. A PCIe *lane* consists of 4 serial links to perform *Dual Simplex* communication that assigns 2 serial links for transmitting data and 2 other links for receiving data. A PCIe endpoint uses at least x1 lane and up to x32 lanes depending on its throughput requirements and the host computer's configuration. The lanes' throughput is also a function of the PCIe generation that both the PCIe device and the host computer hardware are capable of. For example, PCIe 4.0, whose standard was introduced in 2019, is capable of $\sim$2 GB per lane. As a result, a 100 Gbit/s NIC that is connected over PCIe 4.0 needs at least x8 PCIe lanes of bandwidth (in total $\sim$16 GB/s) to accommodate the full 100 Gbit/s throughput ($\sim$12.5 GB/s).

2.2.2.2 PCIe Protocol

The communication between the CPU and the peripheral devices over PCIe lanes uses the PCIe Protocol. The specification of the PCIe protocol describes 3 layers: the Physical Layer, the Data Link Layer, and the Transaction Layer [25]. It is responsible for all communication between the PCIe device and the CPU including data and control messages like interrupts and that used in the configuration space.

The Physical Layer The Physical Layer (PHY) is the closest layer to the serial links in the PCIe protocol and is responsible for actually transmitting and receiving data over the PCIe links. It is divided into 2 sub-layers: the logical and the electrical sub-layers. The logical sub-layer prepares the received packets from the serial links for the Data Link Layer and the sent packets coming from the Data Link Layer for transmission over the serial links. This process involves three primary stages: *data scrambling* to reduce the possibility of electrical resonances on the link, *encoding* for clock recovery, and *packet framing* to identify where a packet starts or ends. On the other hand, the electrical sub-layer transform data in electrical signals that can be transmitted over the serial links by converting parallel data from the logical sub-layer to a serial stream and vice versa.

The Data Link Layer The Data Link Layer has 3 responsibilities for the PCIe link: (1) sequence the packets of the Transaction Layer for identification, (2) ensure reliable delivery through acknowledgements, (3) initialize and manage flow control credits. For transmission, PCIe generates a sequence number and calculates a link cyclic redundancy check (LCRC) for each outgoing Transaction Layer Packet (TLP) as seen in Fig. 2.5. Once received, the receiver checks the sequence number and compares the LCRC to its recalculation to ensure the data link layer packet correctness. If the packer is correct, the receiver sends an acknowledgement (ACK) to the sending PCIe device, otherwise a negative acknowledgement (NAK) is sent requesting re-transmission. The sending device waits until an ACK is received. If ACK is not received after a timeout or if NAK is received, the sending device re-transmits the packet again. The data link layer also generates and consumes data link layer packets, e.g. ACKs and NAKs, power management messages, and flow control credit information. Flow control credit information are issued by the receiver to the transmitter and provide information about its capability to reliably receive TLPs in order to receive congestion and subsequent re-transmission.

The Transaction Layer The Transaction Layer is the final layer in the PCIe protocol and is the one responsible for forming PCIe requests. Each PCIe request then becomes a TLP or a transaction. Each PCIe generation can process a pre-defined number of transactions per second. For example, PCIe 4.0 is capable of 16 billion transactions per second. The TLP contains information about the PCIe request in its header that represents control information including the sender and the receiver addresses, in addition to the payload (for example the data to be sent over PCIe), and another cyclic redundancy check for reliable data transmission called End to End Cyclic Redundancy Check (ECRC). The payload size of a TLP ranges from $128\,\mathrm{B}$ to $4096\,\mathrm{B}$. The selected payload size is negotiated between the PCIe device and the host computer hardware and honoring the supplied user configuration.

Layer	PHY	Data Link	Transaction			Data Link
Data	Start	Sequence	Header	Payload	ECRC	LCRC
Size	4	2	12 – 16	0 – 4096	4	4

Figure 2.5: PCIe Packet Format and Size in Bytes, as of PCIe 3.0

Fig. 2.5 shows the overall format of a PCIe packet as of PCIe 3.0 in addition to the data sizes (in bytes) for each element in every layer of the PCIe protocol. While the payload of a PCIe transaction is up to $4096\,\mathrm{B}$, it has a protocol overhead of $22\,\mathrm{B}$ – $30\,\mathrm{B}$. Larger payload size can increase throughput of the PCIe bus by reducing the transmitted headers and consequently the protocol overhead, but it can increase latency by waiting to accumulate the bytes needed to fill one TLP.

Practical configurations often range from 128 B to 1024 B and thus increasing the protocol overhead.

2.2.2.3 PCIe Device Discovery and Configuration

In addition to devices' memory maps, the enumeration process that is initiated by the host computer constructs the PCIe configuration address space. During this process, the host computer and PCIe protocol exchange information to identify the device address, its vendor, and its class (storage, networking, GPU, etc.) and to negotiate the PCIe generation, lanes and the maximum payload size based on the capabilities and the configurations of the PCIe device and the host operating system. The enumeration process also includes mapping the *Base Address Registers* (BAR) of the PCIe device to main memory to enable the host computer to write configuration to the PCIe device like Type 1 and Type 0 Configuration Tables. This process is usually done during computer startup by the BIOS or the operating system. However, the PCIe protocol also supports *hot swapping* PCIe devices where they can be plugged in or out at runtime without rebooting the system and consequently constructing the configuration space happens at runtime by the operating system.

2.2.2.4 PCIe Data Transfer via Direct Memory Access

While PCIe BARs provide means for data transfer between the CPU and the PCIe device, they are not suitable for bulky data transfer. PCIe devices use *Direct Memory Access* (DMA) to perform this kind of data transfer where the PCIe device directly writes to and reads from the main memory through the Root Complex without involving the CPU and its caches. Before DMA being introduced as a data transfer mechanism, peripheral devices relied on the CPU to fetch the device data to its registers and caches as an intermediary buffer for writing the data to main memory. This device-to-memory data transfer mechanism is inefficient as it includes unnecessary memory copies to the CPU caches and registers in addition to handling several CPU interrupts that are involved in the completion

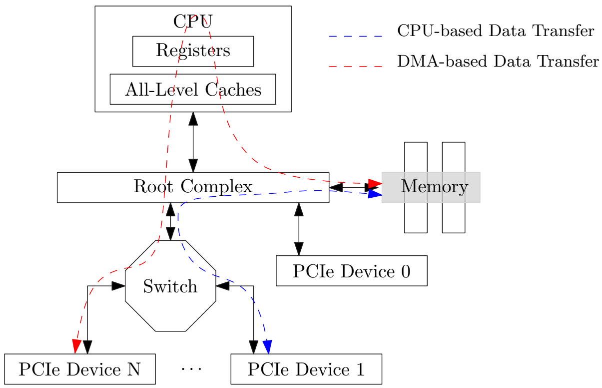


Figure 2.6: CPU-based vs DMA-based Data Transfer in Peripheral Devices

of this process. Alternatively and as seen in Fig.2.6, DMA-capable PCIe devices reduces the CPU involvement in the data transfer by directly acting as bus masters that can access the main memory with high privileges similarly to a CPU. A PCIe device operating as a bus master implements a configurable component called the DMA Engine which manages access to main memory through the main memory addresses provided by the operating system. To perform DMA, the DMA engine in a PCIe device requires contiguous memory to prevent the need to assemble and disassemble scattered buffers. The mechanism of assembling and disassembling scatter buffers is called *Scatter-Gather DMA*. Another mechanism is to provide an overlaying layer of addresses, also known as input/output virtual addresses, that are contiguous from the PCIe device perspective but can be mapped to non-contiguous physical memory. This technique requires the availability of a memory management unit (MMU) for PCIe devices similar to that used by CPUs to translate virtual contiguous addresses to their corresponding scattered physical addresses. Modern computer systems call this device the I/O Memory Management Unit (IOMMU).

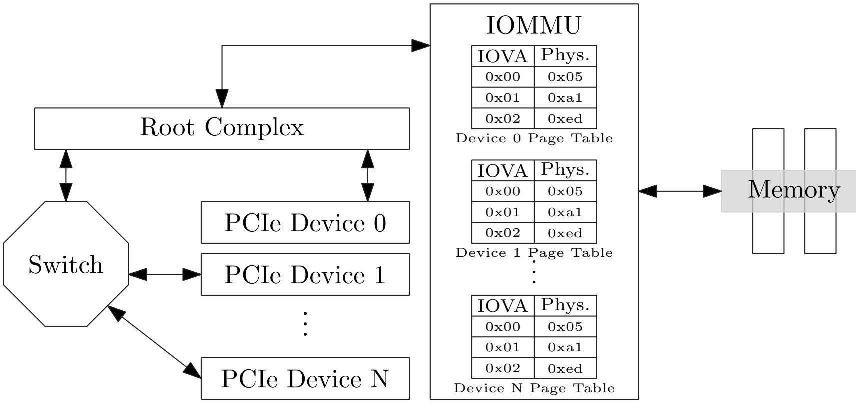


Figure 2.7: IOMMU and its Page Table with respect to the PCIe Bus

2.2.2.5 Input/Output Memory Management Unit

While the IOMMU is not part of the PCIe specification itself, it plays a major role in driving data transfer for PCIe devices. It allows each peripheral device have allows each device to have its own per-device contiguous virtual address space and takes over the responsibility of translating each device's address space to the corresponding physical addresses where the actual DMA buffers are located. This has 2 main advantages: (1) Contiguity: the virtual addresses of a device, hereby called I/O Virtual Addresses (IOVA), are also contiguous but can be mapped to non-contiguous DMA buffers in memory without using Scatter-Gather DMA; (2) Security: instead of accessing the whole host computer address space (i.e. main memory physical addresses), the IOMMU - through per-device virtual address spaces - isolates each device from accessing other devices' buffers or other applications' memory which may lead to leaking of sensitive information or data corruption in memory.

As seen in Fig. 2.7, the IOMMU sits between the main memory and the Root Complex to translate per-device IOVAs to physical addresses. Each device is assigned a data structure called a *Page Table* in the IOMMU. A page table is a list of entries that contains mappings between a device's virtual address and its corresponding physical memory. While virtual addresses are typically contiguous from the peripheral device perspective, they can be mapped to scattered physical memory in the main memory. Therefore, IOVAs mask the memory access to the DMA buffers and provide a transparent mechanism to overcome scattered DMA buffers without relying on Scatter-Gather DMA. Each peripheral device has the same virtual address space as the other devices, but they have different mappings in the IOMMU as each page table belongs one and only one device. For example, address `0x01` in 2 different peripheral devices is mapped to 2 different physical addresses, each mapping uses the corresponding page table of the corresponding peripheral device. This isolates the peripheral device from other devices by limiting its address space to that of IOVAs and prevent memory accesses to physical addresses that they should not have privileges to and, thus, prevent leaking sensitive information from other applications' memory.

The security concern is a highly important concern in computer virtualization scenarios like is the case of DFV. Without IOMMU, a PCIe device belonging to one virtual machine can freely access the memory of another virtual machine because the device is configured to access the address space of physical memory. This is especially important for DFV in order to build secure data acquisition functions in campus computing facilities where diverse applications from different backgrounds and users should run *securely* together.

2.2.3 Networking Stacks

Networking stacks in OS kernels is a complex software system that provide data transfer services to user applications through user-space APIs and by interacting with the NIC driver and hardware. In this section, we will discuss their architecture and internals. We focus only on the data receiving flow as it is the only relevant data path for DFV.

2.2.3.1 Architecture and Data Flow

For security reasons, an operating system is divided into 2 realms of different privileges: kernel-space and user-space. Kernel-space is a privileged environment where software, e.g. NIC drivers, have direct access to hardware and hardware can perform DMA to memory without constraints. User-space is a restricted environment where software is constrained from directly accessing hardware and protected memory regions. The interaction between user-space and kernel-space is performed through a *system call*, a programming routine to perform OS-related tasks, e.g. reading and writing to network. With this architecture, all operations that require exploiting hardware, e.g. NIC, has to go through the kernel-space where the hardware and its memory are managed. Moving memory from kernel-space to user-space requires copying the memory data from kernel-space memory to user-space memory.

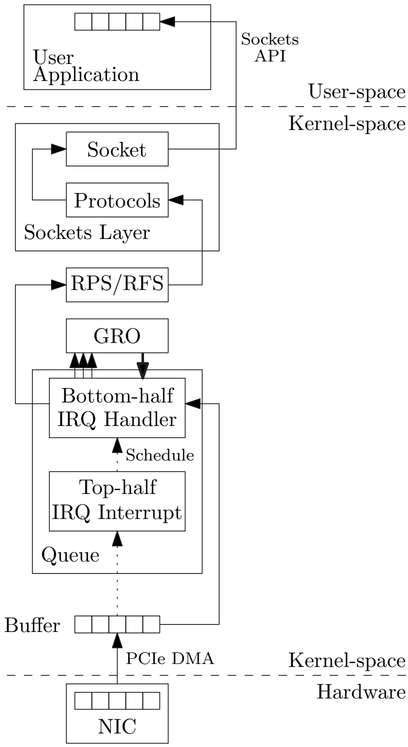


Figure 2.8: Architecture and Receiving Data Flow of OS Networking Stacks

Similarly, network operations go through the kernel where the NIC driver, data transfer protocols, and other security and memory management mechanisms are implemented. The NIC driver defines the method to interact with a NIC hardware. When the driver is initialized, it assigns a physical memory or buffer as a channel for communication between the OS and the NIC hardware. The driver also provides information to the NIC hardware on how to perform DMA to this buffer. This information includes the addresses of the memory chunks and their availability. Each memory chunk have a data structure, called *descriptors*, that represent its information. Descriptors are organized in one or more NIC queues where each queue contains a unique set of DMA addresses or descriptors. Multiple queues linearly scale the performance of the NIC.

When data arrives at a NIC, it goes in the embedded memory buffers of the NIC hardware. The NIC uses the provided descriptors in NIC queues to pick an available memory chunk and then write the received data to the chosen memory chunk using DMA. The OS is notified of new data availability through a mechanism called *interrupts*. Interrupts are notifications generated by peripheral hardware, e.g. NIC, to tell the computer processor that there is an event that needs handling. Once a NIC has new data it generates an interrupt request (IRQ) to be handled. A NIC driver provides the OS with 2 IRQ handlers for each NIC queue. The first IRQ handler is called the top-half IRQ handler, it is dispatched immediately by the corresponding NIC queue once new data is received to acknowledge the event. While running top-half IRQ handlers, the CPU masks other IRQs. Therefore, they should not block the NIC from issuing new IRQs, and thus they must be very fast. Typically, top-half IRQ handlers only acknowledge an IRQ and grab minimal info like memory address. Heavy work like processing the received data through data transfer protocols is deferred to the bottom-half IRQ handler which gets queued in the top-half IRQ handler. Bottom-half IRQ handler perform heavy work like protocol processing and passing the data to the user application. In Linux systems, bottom-half IRQ handlers are known as *SoftIRQ*. The bottom-half IRQ handler allocates a data structure called *socket buffer* which prepares the received data for further processing until it reaches the user application. Socket buffers act as a buffer abstraction layer that unifies memory management for different NICs from different vendors. The networking stack can reduce the number of allocated socket buffers by merging related ones to reduce their processing overhead through a mechanism called the *Generic Receive Offload (GRO)*. Next, processing data transfer protocols, like IP and UDP, is scheduled on one of CPU cores using 2 mechanisms: *Receive Packet Steering (RPS)* and *Receive Flow Steering (RFS)*. The goal of both mechanisms is to increase the NIC performance: RPS is the mechanism of using multiple NIC queues for received data, while RFS is the mechanism that ensures the data is received on the same CPU core as the application to decrease data movements from the NIC to the CPU or even inside the CPU itself. The sockets layer, powered by the *socket buffer* abstraction layer, executes the OS implementations of the data transfer protocols on each socket buffer. The OS organizes data flow between the OS and the application through data structures called sockets, after the protocol processing the received is assigned to its corresponding socket data structure. The application can fetch received data from sockets using the sockets API which are OS routines that move data from the socket data structure in the kernel-space to the application in the user-space.

It is important to note that modern NICs have hardware-based implementations of GRO, RPS and RFS that are known as *Large Receive Offload (LRO)*, *Receive Side Steering (RSS)* and *Accelerated Receive Flow Steering (aRFS)* respectively. Modern NIC moves some other components from software to hardware, like protocol processing and checksum calculations. All of these hardware features are known as NIC hardware offloads.

2.2.3.2 OS Networking Stack Performance

Different studies have discussed the performance overhead of OS networking stacks. Cai, Chaudhary, Vuppalapati, *et al.* provide a deep performance analysis of OS networking stacks in their paper [26]. This study shows that the maximum performance of OS networking stack is $\sim 42\,\mathrm{GB/s}$ even with the full support of NIC offloads. The study breaks the sources of performance bottlenecks: memory copying between kernel-space and user-space, protocol processing, memory management in the sockets layer are among the most contributing factors to performance degradation [26]. Other works have shown that these performance bottlenecks are starting to limit modern applications like databases [27], [28] and mobile telecommunication services [29], [30].

2.2.3.3 Extended Berkeley Packet Filter

Due to the performance bottlenecks resulting from copying memory from kernel-space to user-space, OS developers have designed a new mechanism to run computers programs inside OS without actual modifications to OS source code. The mechanism is known as Extended Berkeley Packet Filter (eBPF). It started in 2014 in the Linux kernel, and it is expanding to other OSs like Microsoft Windows [31].

An eBPF program is a hook to an event inside the OS kernel, e.g. received data, a call to OS API, etc. It runs in kernel-space through a Just-In-Time compilation virtual machine which ensures software security and prevents access to memory it should not have access to. A user application can load an eBPF program to the kernel through an OS routine. The program will be verified by the virtual machine to ensure its instructions do not violate its security constraints. The verifier makes sure: the instructions of the eBPF program do not access memory it should not, the program can finish fast and does not block other kernel components, and finally the program size does not exceed $4096\,\mathrm{B}$. The program will be rejected if the virtual machine verifier decided that it violates its security constraints. Communication between an eBPF program and a user-space application can be performed using special data structures called eBPF maps.

The eBPF programs that take care of computer networking are known as Express Data Path (XDP) programs and were introduced in 2018 [32]. For best performance, they run directly in NIC drivers, the closest to NIC hardware. Some enterprise-grade programs have already started using eBPF programs, e.g. the Cilium software switch used for networking in cloud computing networking. However, since XDP like all eBPF programs are limited in size (e.g. only up to $4096\,\mathrm{B}$-programs) and functionality (e.g. constraints on using loops), they are not suitable for heavy applications like data acquisition functions.

2.3 Introduction to Computer Virtualization

Computer virtualization is a foundational component in computing facilities. It plays a major role in managing and securing access to IT infrastructure through resource pooling and abstraction. However, computer virtualization has a performance overhead caused by the abstraction layers used to virtualize computing resources [33]. State-of-the-art computer virtualization are thus diverse technologies of different performance and software security levels. This section explains the different kinds of computer virtualization technologies and their characteristics. We specifically discuss virtual machines and containers.

Throughout the rest of this thesis, we refer to the virtualized user application as the guest and the actual computer system that is actually running the system as the host.

2.3.1 Virtual Machines

A virtual machine (VM) emulates a real-world computer system and provide an identical environment to that of a physical computer system. A virtual machine monitor (VMM), also called a hypervisor, is a software system that is responsible to provide emulated environments for multiple guest virtual machines by abstracting hardware and computer resources on a host computer system. A VM runs a fully-fledged OS with its own kernel and device drivers which provides a high-level of isolation that makes them a good virtualization candidate for computing facilities that require confidentiality, e.g. public cloud platforms.

Fig. 2.9 shows the difference between hypervisor types. Type-1 Hypervisors interacts with physical hardware directly or through a very thin software layer while Type-2 Hypervisors interact with hardware through a host OS kernel. Thus, Type-1 hypervisors have lower latencies because guest operations have to pass through fewer layers of hardware abstraction layers. Meanwhile, a Type 2 hypervisor reduce guest VM efficiency by inducing higher latencies through the host

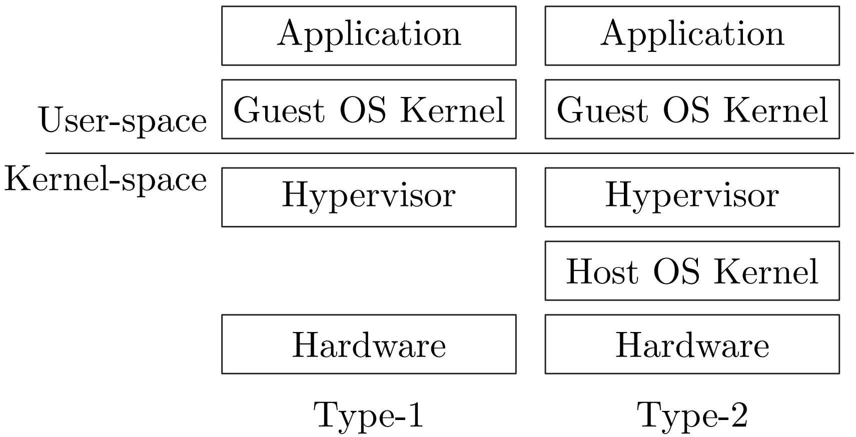


Figure 2.9: Hypervisor Types

OS kernel. Type-1 hypervisors are used in enterprise environments, but Type-2 hypervisors simplify deployment for smaller-scale or personal use cases.

There are 3 main types of virtualization paradigms for VMs: Full Virtualization, Para-virtualization, and Hardware-Assisted Virtualization. Full virtualization emulates the whole system by directly translating the guest VM CPU instructions in to the host CPU instructions. This approach runs unmodified guest OSs rendering them completely unaware of virtualization. In general-purpose computer systems, full virtualization hypervisors impact the performance of applications because of hardware emulation rather than executing guest VM on real hardware. Full virtualization is typically used in Type-2 hypervisors in personal computers and test environments. On the other hand, para-virtualization hypervisors are Type-1 hypervisors. The main idea of para-virtualization is to reduce the hypervisor overhead through a thin hardware abstraction layer. The hypervisor overhead is reduced through a mechanism called *hypercalls*. When a privileged operation is executed on the guest OS, the hypervisor intercepts it through a special hypercall and execute the operation on behalf of the hypervisor before returning the result back to the guest VM. This mechanism requires customizing the guest

OS to enable hypercalls. Without customization, guest OS cannot run a para-virtualized hypervisor. Thus, para-virtualization sacrifices flexibility (ability to run any guest OS) for performance. Hardware-Assisted virtualization includes both flexibility and performance through special CPU instructions extension for virtualization requiring no needs to customize guest OS while achieving good performance. Examples of CPU instructions extension for virtualization is the virtual machine extensions (VMX) of Intel x86_64 architecture. In this thesis, we are most interested in hardware-assisted virtualization as they are widespread in computing facilities. Examples of hardware-assisted virtualization are: VMWare ESXi, the Kernel-based Virtual Machine (KVM) that converts Linux to a Type-1 Hypervisor, etc.

2.3.1.1 Intel Virtual Machine Extensions

Virtual Machine Extensions, or VMX in short, are the x86_64 way to perform hardware-assisted virtualization on Intel CPUs. The extensions provide a set of CPU instructions that alternate the execution between the hypervisor and the guest. In VMX, CPU operations are divided into: root operations (executed by the hypervisor) and non-root operations (can be executed by the guest VM). A VMX-based hypervisor then manage the transitions between root and non-root operations. These transitions are called VMX transitions.

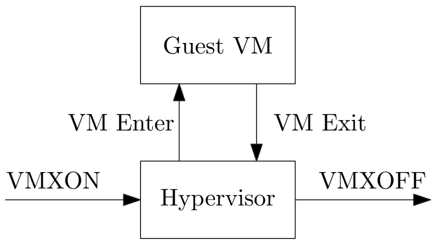


Figure 2.10: VMX Initial Transitions

As shown in Fig. 2.10, a hypervisor enters VMX execution mode through a `VMXON` instruction and similarly exits using a `VMXOFF` instruction. A VMX hypervisor allocates a Virtual Machine Control Structure (VMCS) for every launched VM. The VMCS contains: a copy of the guest state (e.g. registers, control registers, etc.), a copy of the host state, control fields to specify how transitions behave, and execution control flags. A typical VMX transition involve 2 major steps: VM Entering and VM Exiting. The idea of VM Entering and Exiting is to switch the CPU registers according the values in the guest VMCS. On VM Entering, the hypervisor stores the host CPU registers to the VMCS and loads the guest values from the VMCS into the CPU registers. VM Entering is managed through the VMX instructions: `VMLAUNCH` to launch the VM for the first time, and `VMRESUME` to restore the VM after it has been launched and exited. On VM Exiting, the same switch happens but in the opposite direction. All data accesses to a VMCS happen through VMX instructions to prevent incorrect modifications to its data. Examples of VMCS manipulation instructions are: `VMCLEAR`, `VMPTRLD`, `VMREAD`, and `VMWRITE`.

2.3.2 Containers

Containers are another computer virtualization technology that emulate a physical execution environment through isolation features in the host OS. In place of a hypervisor, containers use a container runtime. The unit of isolation in containers is an OS process instead of a complete system (hardware and OS) like in virtual machines. The container runtime exploit the host OS to isolate the guest by preventing unrestricted access to CPU, main memory, and file system resources.

For example, container runtimes in Linux are built using 3 main OS features to isolate a guest process in a host machine: *process namespaces*, *control groups*, and *file system namespaces*. Process namespaces isolate the guest application's processes by emulating an additional process tree (a namespace) in an OS rendering them unaware of system and other applications processes. This feature fortifies the isolation between diverse processes running on the host OS and strengthens the constraints that prevent them from accessing each other memory. Control groups (cgroups) is a feature that limits CPU and main memory resources accessible by the container prevent it from taking all over the available computing resources in a host system. On the other hand, file system namespaces are similar to process namespaces, but instead they isolate file-based operations e.g. access to storage and network devices.

In containers, the guest container still have access to the host OS through its APIs (or system calls). Although this lead to better performance than virtual machines, but they can lead to a security breach like breaking out of the container isolation and accessing sensitive data on the host system. Therefore, container runtimes usually employ additional security layers that reduce the surface of attack inside the container by limiting the container privileges and capabilities and by restricting the access to the host system calls. Examples of such security layers in Linux are: Secure Computing (SECCOMP), Security-Enhanced Linux (SELinux), etc.

Containers are usually hosted in an orchestrator environment. The job of the orchestrator is to scale, manage, and automatically configure containers on a set of computer systems. In other words, the orchestrator manages and virtualizes the resources of a computer cluster. When in need, the orchestrator can perform some privileged operations on behalf of the guest container. Examples of container orchestrators are: Kubernetes, Docker Swarm, etc.

2.4 Related Work

This section dedicated to construct a contrast between our work and related work. We specifically look at computer virtualization and software-based high-performance data acquisition functions in data acquisition systems in scientific research infrastructure and experiments.

2.4.1 Computer Virtualization in DAQ Systems

Computer virtualization in research infrastructures have attracted significant interest. However, it is only used for DFs that does not require high-performance like slow control systems or for offline data analysis services. Examples of DAQ platforms based on computer virtualization in scientific infrastructure are:

(1) At the European Organization for Nuclear Research (CERN): CERN operates its large-scale infrastructure, the large hadron collider (LHC), using the Worldwide LHC Computing Grid (WLCG), an international cooperative project incorporates 170 computing centers in 42 networks over a grid-based network infrastructure [34], [35]. The WLCG, however, operated the computing infrastructure of the LHC and offline distributed data analysis of already saved data, but it does not operate the DAQ systems of the LHC where high bandwidth data from the LHC detectors need direct and online processing [35]. Instead, CERN operates online software-based DFs in a local computer cluster for every scientific experiment, e.g. CMS and ATLAS HLTs [8], [9].

    In 2017, CERN outlined the computing infrastructure challenges to operate the DAQ systems of the LHC experiments in a virtualized computing platform [35]. Later in 2019, CERN published a roadmap to tackle these challenges and computer virtualization was a substantial part of it [36]. In the same year, the scientists of the ATLAS experiment at the LHC published a paper evaluating Kubernetes, a computer virtualization technology, as an orchestrator of the event filter computing resources of the DAQ systems [37], [38]. The ALICE experiment at CERN has also started to adopt virtualized computing technologies in its infrastructure that are mainly for offline distributed data processing of already saved data [39].

(2) At the Karlsruhe Institute of Technology (KIT) for the Karlsruhe Tritium Neutrino (KATRIN) experiment: The KATRIN has operated a virtualized computing infrastructure called KaaS for its data management services and parts of the slow control system [40]. KaaS runs a container-based virtualization solution on only 3 computer servers, and it provides distributed data management services for KATRIN's slow control system. KaaS mainly focuses on data storage and offline data analysis and visualization.

(3) At the Jiangmen Underground Neutrino Observatory (JUNO) in China: Similar to KaaS, the JUNO experiment is operating a service-oriented architecture for its slow control system using a Kubernetes-based computing platform that involves several distributed applications and middleware to guarantee the scalability and the availability of the slow control system of the experiment [41].

2.4.2 High-Performance Software-based DAQ

High-performance software-based DAQ is essential in every research infrastructure. Previous works have discussed high-performance DAQ in the age of $100+\,\mathrm{Gbit/s}$ scientific experiments. The papers in [42], [43] discuss resource-efficient implementations of UDP and Transmission Control Protocol (TCP) protocols on FPGAs; however, it does not discuss how can the UDP or TCP DAQ streams be processed on a computer system. Another solution is presented in [44] which uses the POSIX sockets to perform DAQ up to $10\,\mathrm{Gbit/s}$ but no more. Other solutions rely on RDMA to perform DAQ. The authors of [45] integrate RDMA into slow control systems through a dedicated channel over Infiniband that can be initiated through the control system. The authors of [14]–[16], [46] propose different implementations of RDMA on FPGAs that can take a considerable percentage of the FPGA logic elements (11% to 50%). The DUNE DAQ system proposes a new architecture that is built on top of a high-performance networking technology called the Data Plane Development Kit (DPDK) [47].

However, these studies have not included computer virtualization in their work and have deployed their solutions on a computer cluster local to a research infrastructure. They do not exploit shared computing resources of campus computing facilities. In this thesis, we study the feasibility of running similar software-based DAQ functions in virtualized campus computing facilities.

3 High-Performance Computer Networking Technologies for DFV

DFV relies on computer networking to perform data transfer from detector electronics on the scientific experiment site to the software-based DAQ functions on the virtualized computing premises of campus computing facilities. Therefore, it is highly important to employ a networking technology and protocols that are able to handle all the scientific detector's throughput without data loss and that can compensate for the overhead of distribution of the different DAQ functions.

This chapter is dedicated to compare and analyze different networking technologies that can construct DFV foundations. This chapter is planned as follows: the first part discusses the qualitative criteria of networking technologies that are suitable for DFV, the second part studies 3 categories of networking technologies: hardware-assisted, user-space drivers, and AF_XDP, and then discusses their relevance to DFV based on these criteria, and the third part dives into quantitative analysis of the selected networking technology to study its readiness and robustness.

Publication

Parts of this chapter are published and won the Best Paper Award in the 9th IEEE NFV-SDN Conference:

J. Mostafa et al., "Are Kernel Drivers Ready For Accelerated Packet Processing Using AF_XDP?," 2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Dresden, Germany, 2023, pp. 117-122, doi: 10.1109/NFV-SDN59219.2023.10329590.

3.1 Criteria of Networking Technologies for DFV

In order to know which networking technologies are suitable for DFV, we set 3 constraints inspired by the definition of DFV: *suitability for virtualization*, *resource efficiency on detector electronics*, and *high performance*. By these 3 constraints, we cover the DFs execution environment, the limitations on detectors' electronics, and the throughput requirements of the detector.

3.1.1 Suitability for Virtualization

DFV relies on computer virtualization available in campus computing facilities to run isolated DFs. However, computer virtualization constraints the virtual environment of its guests through abstraction layers on top of the computer host hardware. DFV can only have the virtualization merits if its foundational networking technology respects the constraints of computer virtualization. Such constraints include:

(1) *No Specific Hardware Requirements*: services in virtual environments, like computer networking, are provided through software abstractions of the supporting physical hardware that can be literally anything from commercial off-the-shelf solutions to custom ones. Therefore, campus computing facilities expects their customers, e.g. DFV programs, to treat their virtual environments as black boxes with no special hardware requirements or execution environments that can break their abstraction layers. Thus, a networking technology for DFV should not require a special execution environment.

(2) *Zero-Trust Security Model*: campus computing facilities usually expects *any* application to be deployed on their premises, and therefore they employ a zero-trust security model to protect themselves against potential threats that may abuse a provided service. This security model refuses to

provide the customers with administrative or special privileges as a restrictive contributing factor. Similarly, any choice of a networking technology for DFV should respect this security model.

(3) *Compatibility with Other Applications*: campus computing facilities expects diverse software programs of diverse technologies to work on the same computing premises without obstructing each others. The usage of a networking technology for DFV that obstructs the execution of other applications, e.g. deprive them of networking resources, goes against the principles of computer virtualization. Therefore, a networking technology should also respect compatibility with other applications in order to realize DFV.

3.1.2 Resource Efficiency on Detectors' Electronics

DFV relies on the participation of the detectors' electronics, e.g. FPGA-based systems, in the data transfer between the detector and the software-based DFs. This participation requires that data transfer protocols consume some resources on detectors' electronics. However, resources of detectors' electronics are considered to be scarce and expensive [1] as they are usually used for different components that are responsible for the control and data acquisition in the scientific infrastructure. Hence, any added component, like protocols implementation for data transfer in DFV, should be resource-efficient and only consume resources as low as possible. A networking technology for DFV consequently should not require a data transfer protocol that is resource-demanding on detectors' electronics.

3.1.3 High Performance

Although data filtering and reduction is done on different levels in the DAQ process, software-based DFs are required to process data streams up to hundreds of gigabits per second of throughput. Therefore, a networking technology for DFV should be characterized by high-performance in order to process all the detectors' electronics output throughput without data loss. As discussed in Section 2.1.1 and in order to achieve zero loss of detector's data, a networking technology should not impose a high dispatching latency in software-based DFs that would impact their throughput. A suitable networking technology for DFV has high throughput that would meet the requirements of software-based DFs.

3.2 High-Performance Networking Technologies

To realize DFV, we conduct a survey of available high-performance networking technologies in the industry and the market. The technologies are categorized in three groups: (1) hardware-assisted technologies: these technologies rely on NIC features to achieve high-performance; (2) user-space drivers: ones that rely on providing more efficient implementations of NIC drivers; (3) and AF_XDP: an emerging technology with a novel approach in the world of high performance networking. We study their relevance to DFV according to the criteria discussed in Section 3.1. Since standard OS-based networking is considered inefficient as discussed in Section 2.2.3, we do not consider it in this survey.

3.2.1 Hardware-Assisted Technologies

Hardware-assisted technologies rely on some hardware features in the NIC to assist the data-intensive user program in achieving high-performance networking and consequently overall improved performance. A famous example of such a technology is Remote Direct Memory Access (RDMA) which was initiated by Mellanox Technologies in 2011 as a way to overcome the high latency of data transfer in MPI-based clusters. The main concept behind RDMA was to allow network peers to do DMA to the memory of a targeted system as if they are its peripherals. It is motivated by the fact that DMA minimizes the CPU participation in fetching network data, and, thus, bypasses the kernel networking stack and reduces the memory latency overhead caused by moving data back and forth from main memory to all CPU caches levels. While RDMA relies on the Infiniband interconnect protocol, several extensions has been proposed to cover Ethernet networks which lead for wider adoption as a high-performance networking technology.

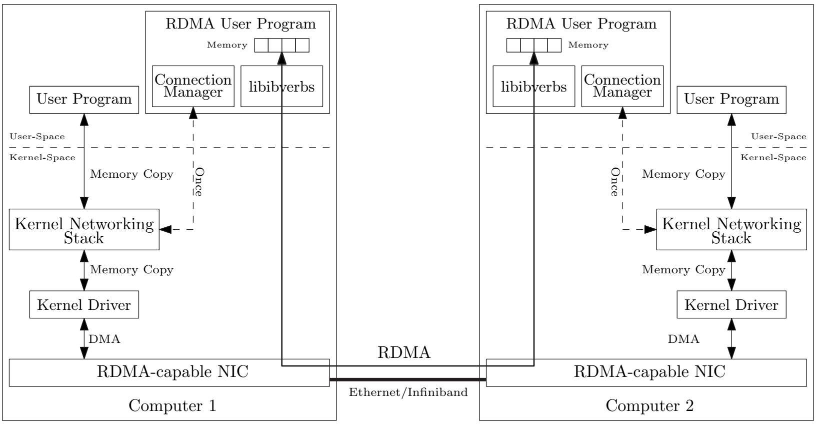


Figure 3.1: RDMA Architecture Compared to that of Kernel-space Drivers

Fig. 3.1 shows how the architecture of RDMA compared to kernel networking stack. While the kernel networking stack is inefficient due to multiple memory copies, RDMA aims to bypass the OS kernel and put data directly in the buffers of the user program and thus yield a significant latency and throughput improvements. To set up an RDMA-based connection between 2 computers (let us call them *Computer 1* and *Computer 2*), each computer has to allocate a memory buffer in its program that is specifically assigned/pinned for RDMA operations. Pinning memory buffers for RDMA operations prevents swapping memory pages out to storage disk and ensures its virtual address is not remapped to another physical address which is essential to perform DMA operations. Through a software library called *libibverbs*, the user inform the NIC of this pinned buffer and create corresponding queues pairs (for both request submission and completion) to manage descriptors of this buffer. To complete the setup of the RDMA session, both computers have to perform a handshake on a separate channel (e.g. on the existing kernel networking stack). This handshake is done only once by a connection manager, but it's required to exchange necessary information to set the RDMA connection up, e.g. number of available queue pairs, security keys, maximum transfer unit, etc. Once the RDMA connection is ready, the user can perform RDMA-based writing or reading to remote memory using the APIs available in the *libibverbs* library. These operations are completely independent of the kernel networking stack, and, thus, performed in less than 10 µs [6], [45], [48].

Supported by the NIC, RDMA achieves ultra-low latencies and has been adopted in several industries including scientific instrumentation. RDMA is heavily used in data center and super-computing infrastructure [13], [48], [49] and in telecommunication networks [50], [51]. RDMA has also gained adoption in scientific infrastructure and instrumentations, e.g. Dritschler proposes the concept of Software-Defined Data Acquisition that aims to replace custom DAQ hardware by commercial off-the-shelf components where RDMA plays a major role to realize this concept. Other authors implement the RDMA protocol on FPGA-based systems to send detectors' data to computer systems [15], [16].

RDMA Extensions for Ethernet As Ethernet bandwidth has gained a huge boost reaching up to hundreds of gigabits per second, RDMA has been extended to work on Ethernet networks. This extension is called RDMA over Converged Ethernet, in short RoCE. RoCE has 2 variants: *RoCE version 1 (RoCEv1)* and *RoCE version 2 (RoCEv2)*. RoCEv1 encapsulates Infiniband frames in Ethernet frames in the NIC hardware, so RDMA operations can be transmitted on a local Ethernet network. Upon arrival of an RoCEv1 frame, the remote RoCE NIC extracts the Infiniband RDMA operation from the frame and processes it as if it was connected to an Infiniband network. Since RoCEv1 use Ethernet frames to transmit Infiniband RDMA operations, RoCEv1 relies on layer 2 MAC addresses to forward RDMA operations between RDMA peers and thus RoCEv1 only works in local networks. To solve this limitation, RoCEv2 was proposed which is quite similar to RoCEv1 but encapsulates the RDMA operation in the Ethernet-based layer 4 UDP and layer 3 IP protocols before transmitting on an Ethernet network.

Another RDMA extension for Ethernet is the *iWARP* protocol which is proposed by the Internet Engineering Task Force (IETF) in RFC 5040 [52] and is similar to RoCEv2 but uses the reliable layer 4 TCP protocol instead of the unreliable UDP protocol. Since these extensions add an overhead to the original protocol, they have a performance penalty [53].

Limitations RDMA and its extensions rely on hardware support to achieve ultra-low latencies. Thus, to use RDMA, a special execution environment consisting of the RDMA-capable NIC and the required software libraries has to be deployed to achieve high-performance data transfer.

3.2.2 User-space Drivers

User-Space drivers are drivers that interact directly with the data-intensive user programs instead of the operating system kernel to overcome the inefficiencies of the operating system kernel. This new perspective simplifies the development of device drivers compared to the development of kernel-space drivers, but most importantly increases the performance of I/O in user programs. User-space drivers exploit some operating systems features, such as the User-space I/O (UIO) subsystem and the Virtual Function I/O (VFIO) subsystem in Linux, which allow user programs to map device memory into program's address space and directly access the hardware device, e.g. NIC, without intermediate buffering [11].

High performance networking employ user-space drivers to bypass the OS kernel's networking stack resulting in reduced latency and increased throughput of distributed user programs like software-based DAQ systems. Fig. 3.2 shows the architecture of solutions based on user-space drivers compared to solutions based on kernel-space drivers. In kernel-space solutions, the NIC driver and the networking stack are implemented as parts of the OS resulting in several memory copies and leading to performance bottlenecks as discussed in Section 2.2.3. User-space drivers are identical to their kernel counterparts in their purpose and the used protocols; however, in the user-space drivers approach, the OS is no

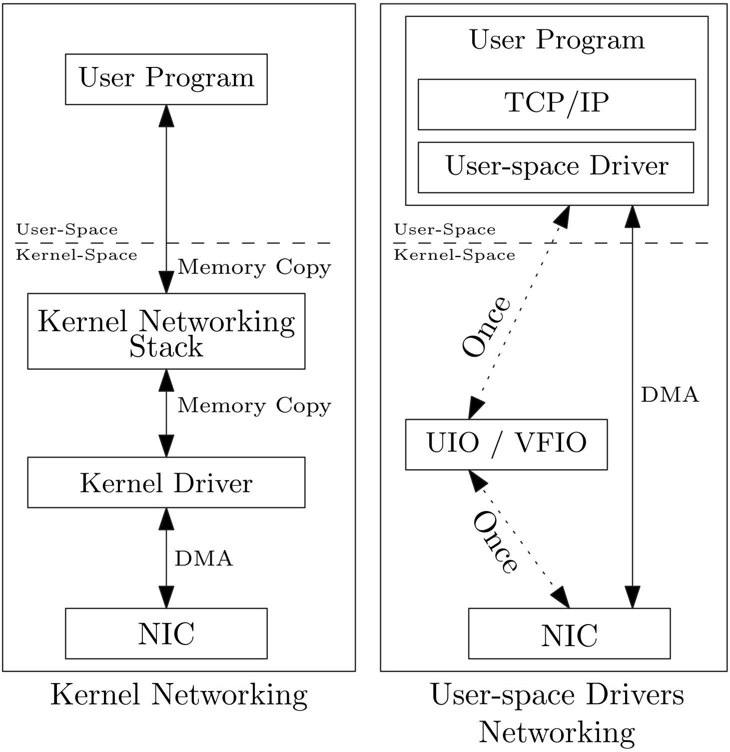


Figure 3.2: User-space Drivers Architecture Compared to that of Kernel-space Drivers

longer responsible for operating the NIC, but the user runs the NIC driver as part of his program in user-space. In this case, the NIC can directly access the memory of the user program without intermediate buffering in the OS which significantly improves the dispatching latency of software-based DFs. This is possible by exploiting OS features like Linux's UIO or VFIO to map the NIC memory to the program's memory address space once as the user-space driver first run. Apparently, the NIC drivers has to be completely reimplemented from scratch in the user-space program without the ability to re-use the source code of the original driver available in the OS kernel [11]. Therefore, user-space drivers are usually provided in software frameworks like the Data Plane Development

Kit (DPDK) that was initiated by Intel in 2010 [54] and Snabb [55] to simplify designing and running user programs exploiting them.

High Performance of User-space Drivers Frameworks Frameworks like DPDK do not only use user-space drivers to achieve the best possible performance. In order to achieve a latency smaller than $10\,\mu s$ (similar to that of hardware-assisted technologies) [48], [2], [56], these technologies employ a set of computer optimizations that require a special execution environment with unrestricted administrative access. For example, the users of DPDK (the most widely adopted user-space drivers framework) often require setting up the BIOS settings to a very low latency profile that disable CPU power management features or reduces System Management Interrupts (SMI) that may interrupt the execution of user-space drivers [11], [57]. In addition, user-space drivers frameworks employ the concept of "Poll-Mode Drivers" where the user-space drivers poll the NIC memory to check for new incoming data instead of using hardware interrupts that notify the driver logic of the new data. For this purpose, DPDK and similar frameworks require setting up dedicated CPU cores that are infinitely polling the NIC buffers without interruptions from the operating system or other running tasks and thus the performance of DPDK is always measured as function of the CPU cores the user has configured to run these frameworks on. Being implemented in user-space, user-space drivers provide a larger set of configurations and optimizations than their kernel counterparts which provides the users with the power to tailor the user-space for their specific requirements and achieve better performance. An example of such optimizations that is easily available in user-space is the *Burst* parameter which allows the user-space drivers to control how much Ethernet frames can be processed in a batch. Powered by such optimizations, user-space drivers frameworks can achieve round-trip latency that is as low as $10\,\mu s$ and can easily process high throughput network links with a few CPU cores, e.g. DPDK can process $100\,\mathrm{Gbit/s}$ of network data using only 1 CPU core [19], [48], [2], [56].

The significant performance benefits of user-space drivers frameworks allowed its adoption in multiple software systems in IT and telecommunication industries.

For example, DPDK accelerates User Plane Functions (UPF) in 5G and 6G networks [29], [30] or in IT infrastructure to accelerate key-value stores like Redis and Memcached [28]. Some software-based DAQ have also adopted DPDK for high-throughput online data reduction like the DUNE experiment [19] and the Elettra 2.0 particle accelerator [58].

Limitations User-space drivers deliver excellent performance but have some limitations. Using user-space drivers means replacing the kernel driver available in the OS kernel, and thus it becomes the job of the user program running the user-space driver to manage and run the NIC. This basically indicates that user program will take over the NIC and prevents other programs from using it. User-space drivers have a completely different foundation than their kernel counterparts, thus a reimplementation of the driver of the same NIC has to take place in the user-space. Although there exists software frameworks that provide a set of ready-to-use user-space drivers, these frameworks only have a limited number of these drivers, and NIC vendors usually refrain from supporting user-space drivers because it enforces major additional work on their staff. In addition to the required BIOS and computer optimizations, these frameworks can only work in a specific execution environment that is only available on specific systems. Other limitations of user-space drivers are related to the software engineering aspects in the user program like their short Long-Time-Support (LTS) time (e.g. DPDK has only 2 years of LTS time) and the need to design and write the program source code specifically for their custom APIs without relying on standard APIs which demands active and heavy software maintenance of the user program. Additionally, programs using user-space drivers have to process protocol headers like UDP and TCP headers as they are no longer processed by the OS. However, processing protocols' headers is usually provided as part of user-space drivers' frameworks.

3.2.3 AF_XDP: The Express Data Path Sockets

An emerging new approach is the new sockets address family AF_XDP (or the eXpress Data Path Sockets) whose foundations were introduced in Linux kernel 4.18 [59]. AF_XDP accelerates packet processing by allowing the developer to configure the kernel driver to use user-allocated memory buffers for packet I/O instead of kernel-space memory buffers and thus bypassing the kernel networking stack. While user-space networking stacks rely on user-space drivers and are considered demanding to fit in a virtualized environment, the approach of using kernel drivers enables the developer to deploy his application on any system and does not restrict the application to special execution environments. This approach can efficiently decrease the needed work to operate an AF_XDP-based function and to migrate from one system to another while achieving good performance.

The concept behind AF_XDP architecture is to let the kernel driver expose its memory model to the user-space in order to bypass the Linux kernel networking stack. For this purpose, the driver is assigned an eBPF/XDP program that redirects packets from the NIC driver in the kernel-space directly to the AF_XDP sockets (XSK) in the user-space.

An application that uses XSKs allocates a memory buffer in the user-space called `umem` using `malloc`, `mmap`, or using huge pages, then registers it with the driver's hardware queue using `setsockopt`. The low-latency packet processing is then achieved by alternating the ownership of the `umem` chunks between the kernel and the application through four in-memory rings that are assigned by other calls to `setsockopt`: one `tx` and one `rx` rings are assigned per XSK, one or more `fill` ring per `umem` to pass chunk addresses from the user-space to be processed in the driver, and one or more `comp` ring per `umem` to get a completion notification in the user-space about chunk addresses where the network operation has finished. The final activation of an XSK happens using a `bind` call with activation options as its arguments to specify the AF_XDP mode: (1) Native Zero-Copy where drivers with AF_XDP support does not allocate any memory buffers but uses the `umem` directly as its own buffer; (2) Native Copy where the driver with AF_XDP support allocates its own private buffers but bypass the other components of the

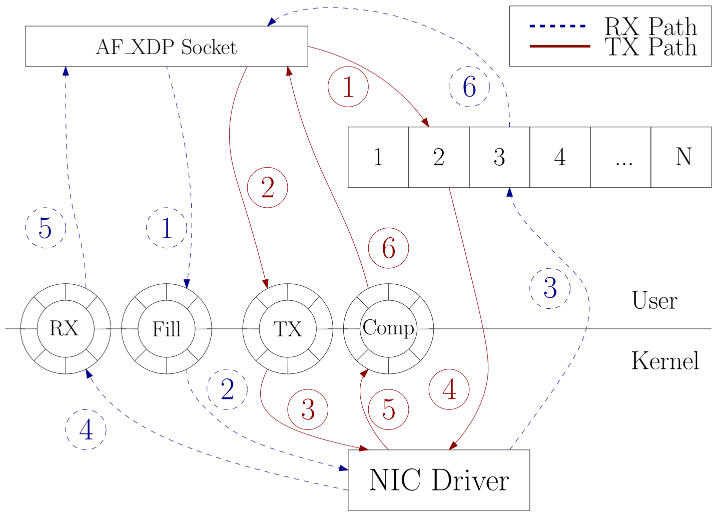


Figure 3.3: AF_XDP architecture and the paths for receiving and transmission.

networking stack by copying the driver buffers directly to `umem`; (3) and Generic mode: it was added for compatibility reasons [60] as it does not need any driver support. Fig. 3.3 shows the architecture of AF_XDP and the transmission (TX) and receiving (RX) paths. The RX path can be summarized as: (1) The XSK application produces `umem` addresses that the driver can fill up with RX data into the `fill` ring; (2) The driver consumes these addresses from the `fill` ring to know where it can put the RX data; (3) The driver puts the RX data in the `umem` at the specified addresses; (4) The driver puts the addresses filled with RX data in the XSK's `rx` ring; (5) The user application checks for new addresses in the `rx` ring; And finally (6) the application fetches the data from `umem` using the addresses in `rx` ring. However, the TX path can be described as follows: (1) The application puts data at a specific index in the `umem`; (2) The application produces `umem` addresses in the `tx` ring for data in the `umem`; (3) The driver consumes addresses from the `tx` ring to know where the TX data is; (4) The driver gets and sends the data in the `umem` of these addresses; (5) When the TX process completes, the driver produces into the `comp` ring the corresponding `umem` addresses of the TX data to notify the application of the addresses that can be re-used; (6) The application get notifications of transmitted data by consuming addresses from the `comp` ring.

The Need Wake-up Flag The developers of AF_XDP applications can add the flag `XDP_USE_NEED_WAKEUP` to the arguments of the `bind` system call. The flag that was added in Linux 5.4 optimizes the data path in case the NIC queue interrupt handler and the application were running on the same CPU core. If the interrupt handler and the application are running on the same core without this flag, the application has little chance to refill the `fill` ring with new descriptors while the corresponding interrupt handler would be continuously busy-spinning to read new ones from the ring [61]. This flag optimizes this behavior by letting the interrupt handler stop spinning and only wake-up when the `fill` ring has been replenished by the developer through an explicit system call. When the flag is enabled on the socket and the interrupt handler has no addresses to fill the `umem` with, the interrupt handler decides to sleep and sets on the `fill` ring an indication that it needs to be woken-up by the application instead of busy-spinning. This will give the application the chance to process the RX packets and refill the `fill` ring. The application then wake up the interrupt handler again using a system call to complete processing the newly added `fill` ring entries. The flag works similarly on the `tx` ring to optimize the TX path: if the interrupt handler has sent all the packets and issued all their completion interrupts, it sets an indication on the `tx` ring that it needs to be woken up to continue sending more packets. This will give the application time to process the completion notifications in the `comp` ring, and fill the `tx` ring with new entries. The application then can wake up the interrupt handler to process the newly inserted entries in the `tx` ring which will also reduce the system calls to read new entries from the `tx` ring in comparison to busy-spinning the ring.

Running with Multiple Queues AF_XDP developers can launch several XSKs that are each bound to a different queue. The `umem` can be organized in two ways: (1) private `umem` per XSK: where each socket has its own `umem` and is completely separated from the other sockets; (2) one shared `umem` for all XSKs:

all sockets share the same `umem` but each has its own `fill` and `comp` rings to communicate with the `umem` (this feature was added in Linux 5.10). In case the developer chose a shared `umem`, he creates the first XSK with the desired `bind` flags, i.e. zero-copy or need wakeup flags, as usual, but not to the other sockets. Instead, he *only* passes the flag `XDP_SHARED_UMEM` for the other XSKs.

Limitations Bypassing the kernel networking stack means that processing protocol headers like UDP and TCP headers and is no longer the job of the OS and should be provided by the user program. AF_XDP does not support all hardware offloading capabilities yet that can hugely improve its performance, but there are some initiatives to solve this problem [62]. As each AF_XDP take over 1 NIC queue, the developer can create AF_XDP sockets no more than the number of the targeted NIC queues.

3.2.4 High-Performance Networking Technologies for DFV

To select a networking technology for DFV, we have to study the criteria discussed in Section 3.1 for the above networking technologies. While high performance is a common characteristic for these networking technologies, there is discrepancy among them in the other characteristics.

Hardware-assisted technologies and RDMA require an implementation of their own protocols. RDMA, specifically, relies on the Infiniband protocol and on a stateful secure connection between the RDMA network peers, and thus, it is considered a heavyweight protocol whose implementations consume a lot of resources on detector electronics. For example, state-of-the-art RDMA implementations on detector electronics use 11% to 50% of FPGA resources [14]–[16], [46]. RDMA also require a special execution environment that demands direct access to the physical RDMA-capable NIC breaking the abstractions of computer virtualization in campus computing facilities. Being resource-demanding on detector electronics, we consider RDMA to be out of the race of networking technologies for DFV.

Similarly, user-space drivers require a special execution environment by directly accessing physical NIC through an independent driver implementation in the user-space. User-space drivers consequently break the abstractions of computer virtualization in campus computing facilities and are considered demanding to fit in a virtualized environment. Fortunately, they employ lightweight TCP/IP protocols to perform data transfer whose implementations are considered resource-efficient on detectors' electronics [1], [42], [43]. Due to their unsuitability for virtualization, we rule user-space drivers out of the race for DFV, but we consider it as a comparison baseline for high-performance networking technologies especially for their resource efficiency in detector electronics.

The emerging technology AF_XDP promises high-performance data transfer and uses the resource-efficient TCP/IP protocols. AF_XDP is also considered suitable for virtualization. It is totally a software solution in the OS that does not require a special hardware or execution environment to work and does not violate the zero-trust security model in campus computing facilities. Since AF_XDP uses the same kernel driver to construct 2 parallel paths: traditional and fast, it does not violate the compatibility constraints of computer virtualization. Thus, we consider AF_XDP to be a good match for networking technologies for DFV. However, AF_XDP is an emerging technology that needs a quantitative performance evaluation to be considered for critical tasks like scientific data acquisition.

3.3 A Top-Down Analysis of AF_XDP Performance

This section targets understanding AF_XDP in deep details and in comparison to other high-performance networking technologies to build robust DFs that do not cause detector data loss and ultimately realize DFV. For this purpose, it is important to study and analyze AF_XDP performance, how design decisions of a kernel driver can impact its performance, and how AF_XDP-based kernel drivers are different from user-space drivers. We propose a top-down analysis of AF_XDP performance based on RFC 2544 "Benchmarking Methodology for Network Interconnect Devices" [63] where the top is an application based on one of the studied networking technologies and the down is the internals of its corresponding networking technology. As hardware-assisted approaches, like RDMA, use heavy protocols and consume a lot of resources to be implemented on DAQ electronics, we do not consider them as candidates for DFV and consequently for this comparative evaluation.

3.3.1 Experimental Setup

As proposed by RFC 2544, we employ a forwarding application for each of the networking technologies: AF_XDP, DPDK, and POSIX sockets. The forwarding application receives data from network using the targeted networking technology, swaps the network addresses, and send it back to the sender. The sender studies the round trip time (RTT) latency in µs and the throughput in millions frames per second (MFPS). We use a binary search algorithm to tune the throughput, then we try to confirm the resulting value for $60\,\mathrm{sec}$.

We run our experiments on two systems: the device under testing (DuT), and the tester machine. Both systems are of an identical hardware setup: 2 NUMA nodes, each has an Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz of $24\,\mathrm{MB}$ L3 cache, and $64\,\mathrm{GB}$ of DDR4 RAM (total RAM per system $128\,\mathrm{GB}$). The systems are connected in peer-to-peer mode using $100\,\mathrm{Gbit/s}$ NVIDIA-Mellanox ConnectX-6 Ethernet NIC over PCIe 4.0 x16 bus. On both systems, we set hardware queue sizes to 1024 and the size of PCIe maximum read request (MaxReadReq) to $1024\,\mathrm{B}$ as recommended by Mellanox performance reports for $100\,\mathrm{Gbit/s}$ interfaces [64]. The tester machine uses Ubuntu 18.04 on Linux kernel 4.15, and runs Cisco `trex` traffic generator that is capable of saturating the link and has proven to be capable of accurate μ-second resolutions in latency studies [65]. The forwarding agent runs on the DuT on top of Ubuntu 23.04 and Linux kernel 6.2. We enable eBPF Just-In-Time compilation for efficient execution of XDP code that redirects packets to user-space [66]. The processor's C-State and P-State are set to *C0* and *P0* respectively, Intel Turbo Boost and Hyper-threading technologies are disabled, and the `irqbalance` daemon is stopped, so we can manually assign the affinity of the interrupt handlers to ensure the process of redistributing the interrupt handlers does not interfere with the application performance. We enable *Spectre v2* mitigation using *retpolines* even if they may degrade AF_XDP performance [59], but they are very important for secure code execution. For AF_XDP, we develop the benchmarking tool *afxdp_perfeval*[1] to act as the AF_XDP forwarding agent using 1 or more NIC queues. For our DPDK measurements, we use the *testpmd* application in *macswap* mode to act as the forwarding agent. We set the compression of completion queue entry (CQE) in the driver to *aggressive* which allows us to save the maximum possible bandwidth on PCIe under high load pressure. PCIe bandwidth is collected using the Intel PCM utility *pcm-pcie*.

The *mlx5* Driver We consider the *mlx5* driver from NVIDIA-Mellanox as an example of an enterprise-grade driver that supports AF_XDP and has another user-space implementation in DPDK. It supports copy and zero-copy AF_XDP sockets by implementing all AF_XDP features. The maximum supported MTU by *mlx5* for XDP is 3498 B [67]. *mlx5* employs Multi-Packet Work Queue Entry (MPWQE), a mechanism to save PCIe bandwidth for RX and TX by assigning one descriptor, or Work Queue Entry (WQE), for multiple packets instead of sending one WQE per packet.

[1] https://github.com/jalalmostafa/afxdp_perfeval

3.3.2 Results

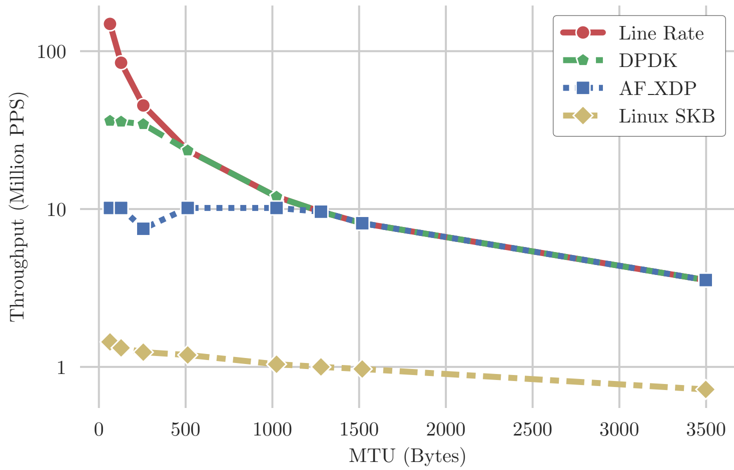
We assign two physical CPU cores per queue for better performance as recommended by [59], one for each of: the AF_XDP application and the SoftIRQ interrupt handler.

Fig. 3.4a shows the achieved frame rate of one NIC queue for DPDK, AF_XDP, and Linux UDP sockets (Linux SKB) as a function of maximum transfer unit (MTU). For jumbo frames, we only consider the largest MTU supported by *mlx5* for AF_XDP: 3498 B. AF_XDP performs better than Linux SKB by achieving 10.16 MFPS in most cases before saturating the link at frame size 1280 B while Linux SKB can't achieve more than 1.44 MFPS at maximum. Contrarily, the highly tuned and optimized DPDK is capable of achieving 36.19 MFPS at maximum in our setup and of saturating the link sooner at 512 B.

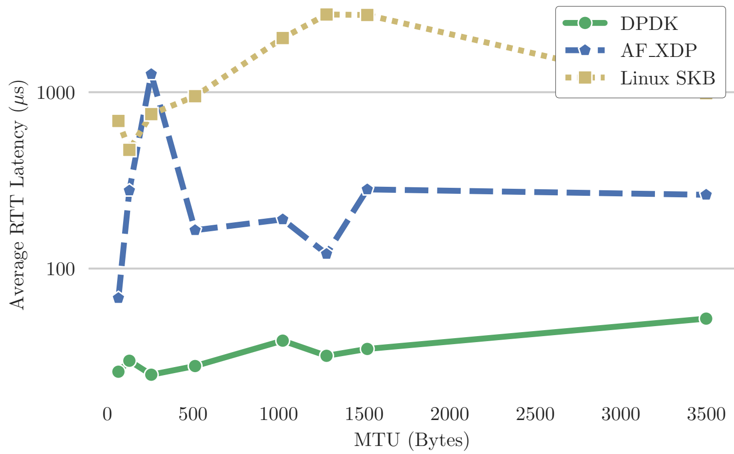
Fig. 3.4b shows the average latency in micro-seconds for all tested MTUs. DPDK also achieves lower latencies than AF_XDP but AF_XDP still provides much lower latencies than Linux SKB e.g. for frame size 64 B DPDK's latency is 9 µs and AF_XDP's is 22 µs but Linux SKB's latency is much higher at minimum 326 µs. The latency values then increase as the frame size increases until AF_XDP reaches 219 µs and DPDK hit the 51 µs at frame size 3498 B.

However, we notice that at frame size 256 B the AF_XDP throughput drops to 7.5 MFPS from 10.16 MFPS as seen in Fig. 3.4a. The drop is also associated with a high average latency value of 1247 µs as seen in Fig. 3.4b. We look at the driver statistics using `ethtool` to find the cause of the problem. We notice that the performance degradation seen in figures 3.4a and 3.4b is associated with a huge number of inlined MPWQE operations of 145 millions for frame size 256 B while it is exactly zero for frames strictly greater than 256 B.

We investigate the *mlx5* MPWQE algorithm source code to understand why the performance degradation is associated with inlined MPWQE. Algo. 1 illustrates the MPWQE algorithm psuedocode. In order to prevent saturating the PCIe bandwidth, `mlx5` uses the MPWQE mechanism to batch multiple frames in one WQE before sending it to the hardware over PCIe. The MPWQE descriptor



(a) Throughput in MFPS



(b) Average RTT Latency

Figure 3.4: AF_XDP and DPDK 1 NIC queue performance

represents each frame by a pointer to the actual data in the memory as seen on line 18 of the psuedocode. On the other hand, when the *mlx5* driver detects that the outstanding TX frames are more than 128 on the NIC, it considers a PCIe congestion has occurred and kicks off the inlining process for small frames (64 B to $256\,\mathrm{B}$) within the WQE by *copying* their data to the descriptor buffer instead of only pointing to them to save even more PCIe bandwidth in favor of more CPU cycles caused by an extra `memcpy` call for each frame (line 15). Once started, the inlining process does not stop until there are less than 10 outstanding frames to send which might cause an *inlining loop* where the outstanding frames were larger than 128 frames, e.g. 129, for a slight period then they dropped below 128 frames but at some values strictly larger than 10, e.g. 11 outstanding frames, which is not enough to deactivate the inlining algorithm. An *inlining loop* has higher chances to occur on larger frames e.g. at $256\,\mathrm{B}$ as they have higher latencies to be transmitted as shown in Fig. 3.4b and consequently stay in outstanding state for more time than smaller ones which can trigger MPWQE inlining. As a result, the extra `memcpy` of the MPWQE inlining algorithm impacts AF_XDP performance by occupying the CPU for more time.

3.3.3 MPWQE Inlining Algorithm Analysis

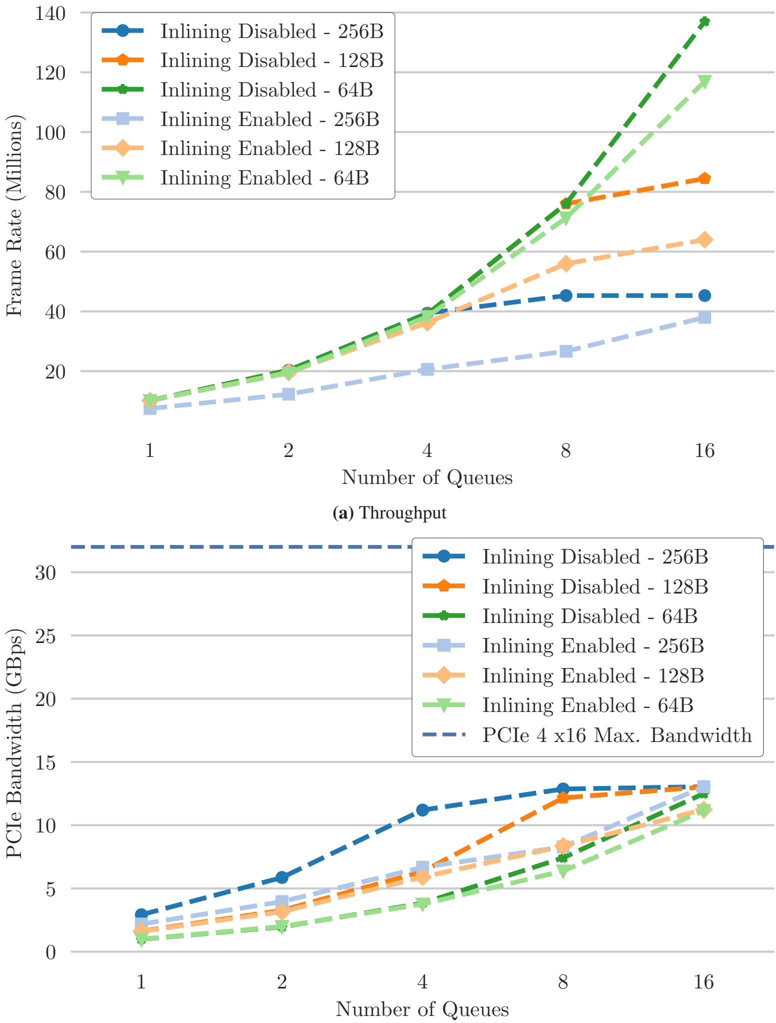
To test the validity of the MPWQE inlining thresholds, we disable MPWQE inlining and check AF_XDP throughput and PCIe bandwidth. To disable the MPWQE inlining algorithm, we had to modify the kernel driver source code by always setting the $inline$ flag in Algo. 1 to $false$ as *mlx5* does not provide user knobs to control the algorithm. Fig. 3.5 shows the throughput and corresponding PCIe bandwidth for AF_XDP using traditional MPWQE inlining algorithm and when MPWQE inlining is disabled in the driver source code. While disabling MPWQE inlining gives AF_XDP the ability to saturate the link using 8 queues (16 CPU cores - 2 cores per queue) for $256\,\mathrm{B}$ frames, it cannot saturate the link for the same size even with 16 queues (32 CPU cores) when inlining is enabled. The consumed PCIe traffic in all cases does not saturate the PCIe 4 x16 bandwidth of $32\,\mathrm{GB/s}$ as seen in Fig. 3.5b. As we increase the number of queues to 4 queues, MPWQE inlining starts to impact the performance of AF_XDP not only at frame size $256\,\mathrm{B}$ but also at frame sizes $128\,\mathrm{B}$ and $64\,\mathrm{B}$ even if the PCIe is not saturated which is a result of MPWQE inlining loops. AF_XDP performs best when the traditional MPWQE inlining algorithm is disabled.

Algorithm 1: Simplified MPWQE Algorithm

```
inline ← false;
while true do
    outstanding_tx ← produced − sent;
    if not inline and outstanding_tx ≥ 128 then
        inline ← true;
    else if inline and outstanding_tx ≤ 10 then
        inline ← false;
    else
        inline ← inline;
    end
    frame ← get_frame();
    wqe ← get_mpwqe_descriptor();
    if inline then
        length ← wqe.length;
        memcpy(wqe.buffer, frame, length);
        wqe.buffer ← wqe.buffer + len;
    else
        // WQE gets a pointer
        wqe.frames[i] ← frame;
    end
end
```

3.3.4 Inlined MPWQE in Mellanox DPDK User-space Driver

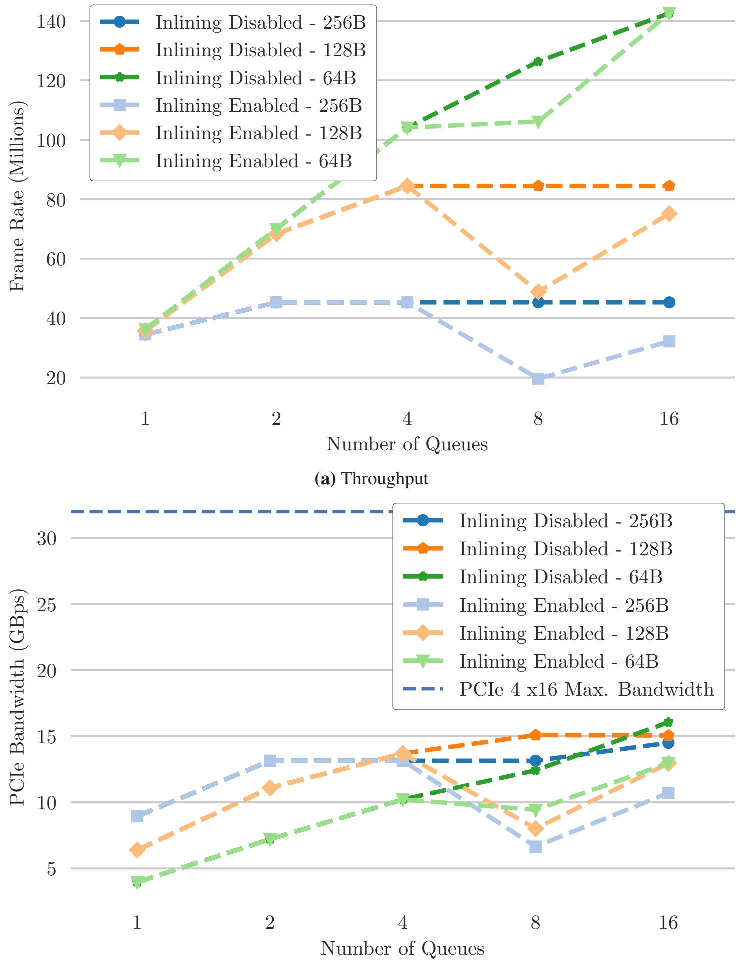
While *dpdk-mlx5* (the DPDK variant of *mlx5*) also uses MPWQE inlining, it employs a different algorithm to trigger the inlining process that is based on the available CPU resources where it only starts MPWQE inlining if the number of the TX queues reaches a user-defined minimum. According to DPDK Queues-to-Cores mapping (one core per CPU), this also means that there is enough CPU cores to handle the overhead of the extra `memcpy` induced by inlining. By default, the minimum number of TX queues to start MPWQE is set to 8 queues, i.e. 8 cores, in *dpdk-mlx5*. Fig. 3.6 shows the throughput and corresponding PCIe



(a) Throughput

(b) PCIe Bandwidth

Figure 3.5: Multiqueue AF_XDP Throughput and PCIe Bandwidth Considering MPWQE Inlining State



(a) Throughput

(b) PCIe Bandwidth

Figure 3.6: Multiqueue DPDK Throughput and PCIe Bandwidth Considering MPWQE Inlining State

bandwidth for DPDK when MPWQE inlining is completely disabled (no inlining at all costs) and when the number of TX queues is assigned to the default value of 8 queues. The throughput of *dpdk-mlx5* scales linearly until it drops when inlining is activated at 8 queues for all frame sizes. When we disable MPWQE inlining using *dpdk-mlx5* driver options, the throughput scales linearly in all cases on the expense of more PCIe bandwidth as seen in Fig. 3.6b and Fig. 3.6a. However, we notice the importance of MPWQE inlining feature in *dpdk-mlx5* when we employ 16 queues and consequently 16 CPU cores where both configurations score the same throughput at 142.59 MFPS but with 3 GB/s less PCIe traffic for when MPWQE inlining is used.

3.4 Discussion

As seen in Fig. 3.6, the MPWQE inlining is helpful in certain setups when PCIe is congested or when there are enough CPU resources to compensate for the extra CPU cycles needed by the `memcpy` to save PCIe bandwidth. Therefore, any driver employing the MPWQE inlining feature must provide the users with control knobs to tune its performance per setup. For this reason, *dpdk-mlx5* provides 2 user-accessible driver knobs to control MPWQE inlining: (1) `txqs_min_inline` to control the minimum number of TX queues to start MPWQE inlining (default value: 8); and (2) `txq_inline_mpw` to specify the maximum frame size to be inlined (default value: 268). In comparison, the kernel *mlx5* driver MPWQE inlining algorithm is hard-coded and can lead to inlining loops when MPWQE inlining is not needed. The kernel *mlx5* also does not provide any user knob to control the MPWQE inlining feature. Adding these knobs can contribute to AF_XDP performance on the *mlx5* kernel driver and increase its operational maturity.

While these results show some limitations in *mlx5* driver implementation for AF_XDP, they do not affect software-based DFs or DFV. This limitation only affect the data transmission path which is not usually used in software-based DFs. DFV and software-based DFs are designed to work on top of the data reception path which does not have this limitation. Moreover, this limitation only happen at small packet sizes (256 B or below) but not at larger ones that are usually used in software-based DFs and in scientific infrastructure. On the other hand, AF_XDP is an emerging technology that is evolving every day.

3.4.1 Evolution of AF_XDP

While there might be some minor limitations in AF_XDP, its development process is active and is lead by large IT companies like Intel, NVIDIA/Mellanox, and Red Hat. This process is powered by the community AF_XDP has succeeded to build in industry, academia and through individual open-source contributions. For example, we count 39 NIC kernel drivers in the Linux kernel from 23 different IT companies and projects that has already supported AF_XDP. The complete support details are available in table in the Appendix A.1.

Figure 3.7: AF_XDP Throughput: Bug vs Our Patch

Another example of continuous improvements of AF_XDP is our contributed fix to its engine in the Linux kernel. During our evaluation of AF_XDP, we have discovered that AF_XDP performance degrades when multiple XSKs share the same `umem`. We found that the configurations to reproduce this bug are: (1) shared `umem` for multiple XSKs is being used; (2) each XSK is configured to run on the same core as its interrupt handler; and (3) the need wake-up flag is activated. After further investigations, we found that the need wake-up flag is not being passed by the AF_XDP engine to the other XSKs when a shared `umem` is being used. As a consequence, XSKs without the need wake-up flag fight with their corresponding interrupt handlers over refilling the descriptor rings as described in 3.2.3. Without enough descriptors, the interrupt handler cannot fill the incoming data in the `umem`. Fig. 3.7 shows AF_XDP throughput in millions Ethernet FPS before our fix and after the fix. Before our fix, only the first XSK performs well and the other XSKs have a negligible contribution to the overall performance. On the contrary, our fix results in similar high throughput in all XSKs. The fix has been contributed to and merged in the Linux kernel [2].

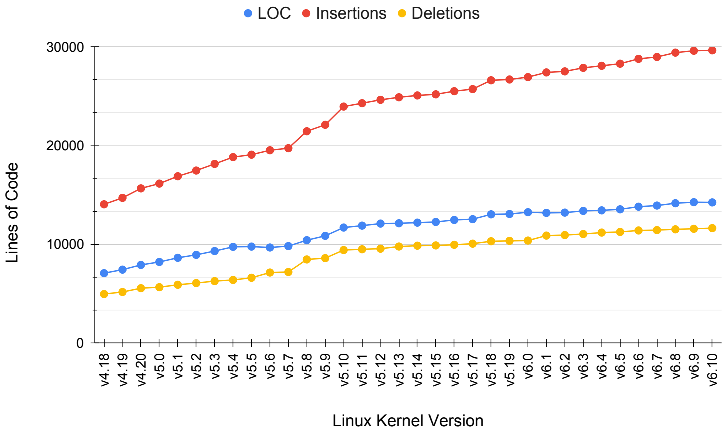


Figure 3.8: AF_XDP Lines of Code in Each Kernel Version Starting v4.18

Software engineering metrics, like AF_XDP lines of code over time and Linux releases, provide a good understanding of the software evolution of AF_XDP and gives insights about its continuous improvement and maintenance. Therefore, we also look into these metrics of the AF_XDP source code and its drivers' support available in the Linux kernel source code. We develop a small tool [3] to find these software engineering metrics. The tool mines the Linux kernel source code and searches for AF_XDP semantics and knobs then for each kernel release calculates: the total lines of code, the inserted and the deleted lines of code, and the total number of commits. Fig. 3.8 shows the total lines of code, the inserted lines of code, and the deleted lines of code for each kernel version starting the v4.18 (the initial kernel release with AF_XDP support). The figure shows that the development and the improvement of AF_XDP is active with thousands of

[2] https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=60240bc26114
[3] https://github.com/jalalmostafa/mining-afxdp

line additions and deletions in the latest kernel release v6.10. The total lines of code is steadily increasing with time which signifies that there is maintenance and development activity.

As a result, AF_XDP might have some minor issues that are still under development and maintenance by a large community that includes large IT companies.

3.5 Conclusion

A high-performance networking technology forms one of the cornerstones to realize DFV. A networking technology for DFV should be suitable for virtualization and the implementation of its data transfer protocols should not be resource-demanding in order to make the integration of DFV into existing scientific detector electronics as seamless as possible. This chapter studied and analyzed DPDK, RDMA, and AF_XDP as candidates for DFV. While other networking technologies are capable of handling the high throughput of software-based DFs but with poor support for DFV, AF_XDP provides good throughput with very good support for virtualization and depending on the lightweight TCP/IP transfer protocols that do not consume many resources on detector electronics. Throughout the rest of thesis, we build our solutions to realize DFV using AF_XDP.

4 Cache-Aware Framework for High Performance Data Acquisition Functions

Data transfer of scientific data is a crucial process that must be robust to prevent any loss of valuable scientific data. Any disruption during the data transfer process can increase the dispatching latency of the queuing-based networking system that performs data transfer and consequently leads to potential scientific data loss. Therefore, it is crucial to understand the reasons causing these disruptions and control them with the appropriate means.

This chapter studies high dispatching latencies caused by the computer architecture in general-purpose computer systems. In response to this problem, we propose the *Data Acquisition Development Kit* (DQDK), a software framework to enable DFV and ensure reliable data transfer of scientific detectors' data. DQDK exploits best programming practices and off-the-shelf features of general-purpose processors to mitigate memory issues in general-purpose computers.

Publication

Parts of this chapter are published in IEEE Transactions on Nuclear Science and 24th IEEE Real-Time Conference:

J. Mostafa, D. Tcherniakhovski, S. Chilingaryan, M. Balzer, A. Kopmann and J. Becker, "100 Gbit/s UDP Data Acquisition on Linux Using AF_XDP: The TRISTAN Detector," in IEEE Transactions on Nuclear Science, doi: 10.1109/TNS.2024.3452469.

4.1 Sources of High Dispatching Latency in an AF_XDP Application

Zero-copy data transfer using AF_XDP offers a great advantage over other networking technologies to realize DFV. However, as a software solution, AF_XDP in general-purpose computer systems is prone to the inefficiencies and bottlenecks of general-purpose computer systems especially in campus computing facilities where the architecture of computer systems are complex and diverse applications are competing for resources. These inefficiencies, if not handled adequately, can cause high dispatching latencies impacting the overall performance of the queuing-based DF. As discussed in Section 2.1.1, a failure in the queuing system can lead to potential loss of scientific data.

4.1.1 Memory Hierarchy & The Principle of Locality

In 1994, computer scientists realized that the rate of improvement in computer CPUs is exceeding that of Dynamic Random Access Memory (DRAM) causing the DRAM to be a performance bottleneck in computer programs [68]. The cause of these inefficiencies in general-purpose computer systems is the required latency to access a computer's main memory e.g. access to a Dual Inline Memory Module (DIMM) DRAM takes up to $100\,\mathrm{ns}$ while CPU registers only take up to $0.5\,\mathrm{ns}$ [69]. This problem became to be known as the *Memory Wall Problem*. In other words, when accessing data in main memory, a CPU has to wait until the required bytes has been fetched from slower main memory to the faster CPU registers in order to continue executing the necessary computations. For this purpose, computer architects augment the CPU with a set of small memories of varying access speeds and sizes to the CPU called *CPU caches*. These caches in addition to the CPU registers and the main memory are organized in a memory hierarchy whose model is shown in Fig. 4.1. The model organizes the trade-off between memory speed and size as faster memory is expensive to design and produce. Consequently, closer memory to the CPU, e.g. Level 1 caches, are smaller but have lower access latency (faster) and distant memory is larger but have higher access latencies (slower). Through this hierarchy, CPUs try to keep frequently accessed data closer to the CPU. i.e. when a CPU wants to process some data, it first searches the Level 1 cache, if the data is not found, it searches the higher level cache until finally fetching it from the main memory if not found in the caches at all. Consequently, this architecture minimizes the *average* latency that is imposed by frequent accesses to the slow main memory and thus leads to faster execution of the target program. This is called the *Principle of Locality* where a program is expected to run faster when accessing memory closer to the CPU.

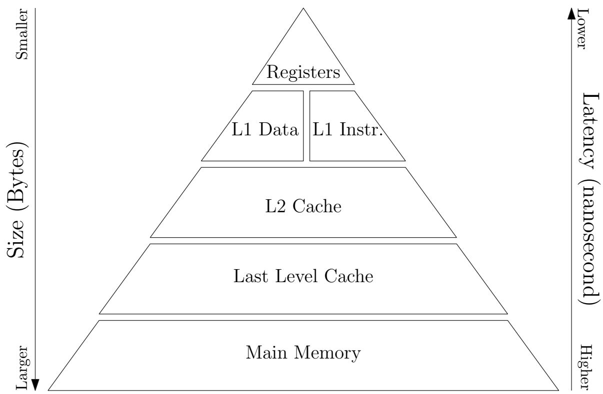


Figure 4.1: The Trade-Off between Size and Access Latency in General-Purpose Computer Memories

The Memory Wall Problem impacts data-intensive applications like DFs especially because they require a lot of memory to store and perform computations on data. This is notably true, when the memory access pattern does not acknowledge the memory architecture and frequently access distant slow memory instead of close fast memory. Thus, poor memory access patterns can violate the Principle of Locality, trigger the Memory Wall Problem, and as a consequence cause significant degradation in overall performance of the DF. Major factors that violate the Principle of Locality are: interference from the operating system and unoptimized memory accesses without tuning for the underlying memory architecture.

4.1.2 The Memory Wall in Modern Computer Systems

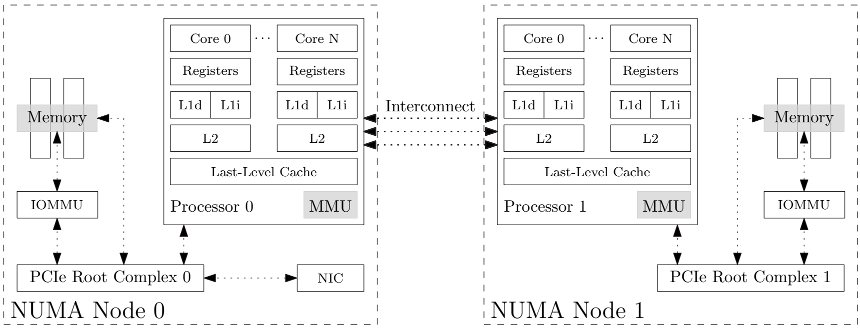


Figure 4.2: General-Purpose Computer and Memory Architecture used in Data Centers

To understand the causes of poor memory access patterns, let us understand a state-of-the-art computer architecture used in data centers. Fig. 4.2 shows an example of such a computer and memory architecture. In a state-of-the-art data center, computers use 1 or more memory nodes, each adopting the memory hierarchy shown in Fig. 4.2. Each memory node has a complex architecture that is designed for parallel job execution augmented with multi-level caches and a dedicated channel to its DRAM main memory.

A modern CPU consists of multiple CPU cores that can work independently and in parallel, hence this architecture's name is Symmetric Multi-Processing (SMP). As shown in Fig 4.2, each CPU core has a dedicated Level-1 (L1) and Level-2 (L2) caches, but they share the Last-Level Cache (LLC). A CPU cache is a Static Random Access Memory (SRAM) that keeps a copy of the memory chunk as physically close as possible to the CPU. A closer cache means a faster memory access according to the Principle of Locality. If the data is found in a CPU cache, this event is called a *cache hit* while the opposite is called a *cache miss*. Since CPU caches are of limited small capacity, the LLC is usually considered a good compromise between proximity (speed) and capacity. Unlike main memory, cache memory is allocated statically i.e. memory has to be allocated in constant predefined chunks called *cache lines*, e.g. the *x86_64* architecture defines the cache line size to be 64 B. Since CPU caches are limited in capacity, CPU manufacturers implement algorithms to decide when a cache line is no longer needed by the CPU, so it can be evicted from the cache and its space can be freed up. Examples of such eviction algorithms are, but not limited to: First In First Out (FIFO), Least Recently Used (LRU), etc. Since data available in multiple CPU caches (e.g. L2 caches for 2 different CPU cores) can be accessed and modified individually and in parallel, the multiple cache copies may differ at a given time and may lead to data corruption and undesired behaviors. To ensure the correctness and the coherence of a chunk of data whose copies are available in multiple CPU caches at the same time, CPU manufacturers implement the hardware cache coherence protocols, e.g. the MESI and the MOESI protocols, that synchronize and validates the correctness and the coherence of the multiple private copies across the CPU caches. For example, when a cache line has been changed in one CPU cache, all other cache copies (or other corresponding cache lines) gets invalidated requiring the CPU to mark these cache lines as invalid and their consequent accesses should result in a cache miss, so the CPU can fetch them again from the updated main memory chunk.

To expand the resources beyond one memory node, several memory nodes can be used. Each memory node has a dedicated set of main memory that is connected to its SMP CPU through a dedicated memory channels and a dedicated PCIe Root Complex. The dedication of resources and their expansion on multiple nodes ease the memory wall problem by distributing the available memory resources and channels on multiple tasks and thus lowering the competition on these resources. Accessing memory from one memory node to another is possible through a proprietary interconnect that links 2 or more memory nodes, e.g. Intel's Ultra Path Interconnect (UPI) or AMD's Infinity Fabric. Such memory accesses are called *Non-Uniform Memory Accesses* (NUMA), and, similarly, the memory nodes are called *NUMA nodes*. Every NUMA node has its dedicated PCIe Root Complex and consequently its own IOMMU. When connecting a peripheral device, e.g. a NIC, to a computer over PCIe, it is connected directly to a specific NUMA node through its dedicated PCIe Root Complex and thus the OS binds this device to this NUMA node by allocating memory for its driver and interrupt handlers only from its NUMA node's main memory. Consequently, all memory accesses of the driver and the interrupt handlers are local to the NIC NUMA node. However, this does not guarantee that the networking application, e.g. a DF, will run on the same NUMA node as the NIC driver and interrupt handlers, on the contrary, the OS is free to assign the application to whatever CPU core and whatever NUMA node he thinks is best based on its process scheduling algorithm.

4.1.3 NUMA and the Principle of Locality

While a NUMA architecture provides additional memory resources (channels, memory size, IOMMU, etc), remote memory accesses in a NUMA architecture impacts the overall performance of a low-latency application like DFs. They violate the Principle of Locality because they impose additional latency for 2 additional processing components: the interconnect between the NUMA nodes, the cache-coherency protocols. To access a remote memory in a NUMA architecture: (1) the requester CPU sends the NUMA request to the CPU owning the required memory; (2) the owner CPU loads the data from main memory to its local caches; (3) the owner CPU writes the data to the interconnect connecting it to the requester CPU; (4) the requester CPU reads from the interconnect and loads it to its local cache. However, having 2 different copies of the same memory in the requester and the owner CPU caches may lead to cache incoherence problems. Therefore, the CPUs use the cache coherency protocols to synchronize the data between the 2 CPUs. This is referred to cache coherent NUMA (ccNUMA). However, the cache coherence overhead of ccNUMA architecture is an extra latency added by the cache coherence protocols to validate and synchronize the cache copies again. As a result, NUMA and ccNUMA are expensive and imposes an additional latency that affect the overall performance of a low-latency application, e.g. using Intel UPI, the added latency can reach up to $150\,\mathrm{ns}$ per memory access [70].

Operating systems plays a major role in the NUMA behavior of an application through its memory pages and the NIC interrupts. As seen in Fig. 4.2, NICs are physically bound to a NUMA node when installed in a computer. To privilege maximum possible performance, operating systems usually run the NIC top-half and bottom-half interrupt handlers on the same NUMA node that the NIC was physically bound to, which prevents unnecessary remote memory accesses. However, there is no guarantee that the low-latency networking application will be scheduled to run on the same NUMA or even if its memory pages will be allocated on the same NUMA. These decisions are left for the operating system's process scheduler and memory manager which may decide at runtime to run the low-latency application on a different NUMA node than that of the NIC. For example, the operating system may decide to balance the memory load on multiple NUMA nodes by migrating an application's memory pages from the NIC's NUMA node to remote one. This may lead to remote memory accesses while moving the data from the memory buffers of the drivers to the memory buffers of the application, and to increased data transfer latencies as a consequence. This is especially relevant in general-purpose operating systems in data centers because they are not usually tuned for specific workloads like low-latency workloads. Therefore, an application that does not manage its memory efficiently on a NUMA architecture may face performance degradation.

4.1.4 SMP and the Principle of Locality

The Principle of Locality in SMP relies on the behavior of the application and its efficiency in utilizing this memory hierarchy. A low-latency application in a SMP architecture efficiently utilizes the CPU caches to prevent high latencies resulting from frequent main memory accesses. Thus, it is in favor of a low-latency application to also decrease the number of cache misses, especially that of LLC, to minimize the number of main memory accesses and preserve the Principle of Locality. However, some common memory access patterns violate the Principle of Locality in SMP architectures. Examples of such memory access patterns are:

Cache Contention & Thrashing Inefficient algorithms implementations in low-latency applications can lead to cache contention and thrashing. Cache contention is the phenomena where shared caches, like LLC, are overwhelmed by its subscribers. In a cache contention scenario, the shared cache is typically full, and the CPUs are competing to allocate cache memory leading to very frequent cache evictions. The result of frequent cache evictions is frequent cache misses: a CPU core processing some data has to pause because the data was evicted from the cache, encounter a cache miss, and wait until the required data is available in the cache again. Similarly, cache thrashing causes frequent cache misses because the required data size does not fit in the cache, and it has to trigger the cache eviction algorithm frequently to free up spaces of still required data in order to fetch others from main memory. Freeing up and evicting required data will cause a cache miss later when the CPU needs this data again to continue processing. Cache misses resulting from both cache contention and thrashing violate the Principle of Locality and lead to a significant overhead and overall performance degradation.

Inefficient Thread Synchronization In order to avoid data corruption, parallel execution on multicore CPUs require synchronized accesses to chunks of shared memory through synchronization techniques like e.g. locks, mutual exclusions, and semaphores. These synchronization techniques store their states in a shared memory chunk. In case this state is updated frequently, so will its shared memory chunk. An example of this situation is when the threads are acquiring and releasing a lock, on each lock acquisition or release, the state of the lock will change. When the lock's state change, all representative private memory copies available in all CPU caches should be updated using cache coherency protocols. Frequent updates to the cache copies will trigger the cache coherency often to mark all cache lines available in all corresponding CPUs as invalid and consequently resulting in cache misses when trying to access the lock's state. This phenomenon is known as the *Cache Line Ping-Pong*, and it violates the Principle of Locality due to frequent cache misses leading to an overall performance degradation.

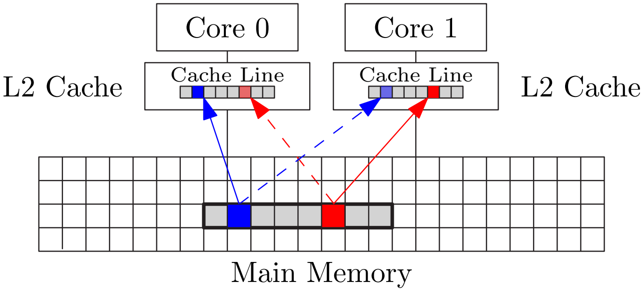


Figure 4.3: An Illustration of False Sharing in L2 Caches of 2 CPU Cores

False Sharing Using locks and their potential performance degradation can be mitigated by avoiding shared data at all. However, this does not mean there is no potential performance degradation. An example of such a scenario is the *False Sharing* phenomenon where the software was not designed to share memory between multiple parallel cores, but sharing is enforced by the memory architecture leading to triggering cache coherency protocols and consequent cache misses. Here is an example. Let us assume the parallel software is running on 2 CPU cores: Core 0 and Core 1, as shown in Fig. 4.3. Core 0 only processes the blue data chunk, similarly, Core 1 processes the red data chunk. Both memory chunks (red and blue) are independent, i.e. processing one of them does not require thread synchronization, but they are located close together in main memory and can fit in one cache line. When these memory chunks are loaded to CPU, SRAM caches do not only load the required blue or red box but load a whole cache line (the smallest possible SRAM cache allocation) from the main memory. The loaded cache line consequently can contain required and unrequired memory chunks. For example, both Core 0 and Core 1 load the cache line that contains both the blue and the red chunks. When any of the members of this cache line change, e.g. Core 1 changes the red chunk, Core 0 which also owns a copy of the cache line containing the red chunk have to preserve the coherence of its copy through cache coherency protocols, invalidate its cache line, and fetch it again causing a cache miss. The cache invalidation process occurs even if Core 0 does not process the red chunk. Thus, the red memory chunk is said to be *falsely shared* because it was not intended to be shared, and the same could be said about the blue chunk as marked by the dashed lines in Fig. 4.3. Frequent cache invalidation due to false sharing will cause lots of cache misses, violate the Principle of Locality, and consequently lead to an overall performance degradation.

OS Interference Other factors that contribute to the violation of the Principle of Locality are caused by the shared execution environment the low-latency application is running in. General-purpose OSs are designed to fairly share the CPU cores among hundreds of applications running on the computer system. Hence, they implement a subsystem called the *process scheduler* which is responsible to distribute CPU resources among applications. The scheduler usually assigns a time quantum for each application to run on CPU resources before it gets suspended to give the CPU resources to another process. This transition of processes on a CPU core is called a *context switch* where the CPU evict all cache lines of the suspended application to free up space for the new application. When the scheduler resumes the application again by giving it another time quantum and CPU resources, the application has to load its memory back again to the CPU caches resulting in cache misses. Sometimes, the scheduler decides to migrate an application from a CPU core to another which will also force the application to reload its memory to the caches of the new CPU core causing a cache miss. Frequent suspensions and CPU core migrations will lead to frequent cache misses which will violate the Principle of Locality and lead to overall performance degradation.

4.1.5 DFV and Memory Wall

As a class of low-latency applications, DFV needs to mitigate the Memory Wall Problem. Triggering the Memory Wall Problem in DFV could lead to high dispatching latencies in the queuing system and probable loss of scientific data as DFV might not be able to process incoming network data in time. Optimizing the data transfer process by mitigating the Memory Wall Problem would minimize the overhead imposed by the computer and memory architecture leading to significantly lower dispatching latencies and consequently the ability to process the incoming detector data in time with zero loss.

4.2 The Data Acquisition Development Kit

To mitigate the Memory Wall Problem in DFV, we propose the Data acquisition Development Kit (DQDK) software framework. The DQDK framework exploits the Principle of Locality by adopting a cache-aware design to reduce expensive cache misses while transferring detector data from detector electronics to the campus computing facilities or data centers. The framework considers the campus data center to be a black-box, and rather than requiring a special memory architecture, it adapts its execution to the available memory architecture for maximum performance.

4.2.1 Design Principles

The DQDK framework adopts several design principles to leverage the Principle of Locality in SMP and NUMA memory architectures.

Efficient Scalability A universal framework that can work for most scientific detectors should be scalable in order to handle increasing detector throughput. We aim to design DQDK framework to be scalable using multiprocessing in a SMP architecture. To prevent scalability bottlenecks in SMP architecture like False Sharing, Lock Contention, etc., the framework adopts a lock-free shared nothing design for its multiprocessing infrastructure.

Cache-Awareness Whether it is NUMA, False Sharing, etc., the overall performance degradation is caused by failing to access the data in a close memory to the CPU. Hence, the Principle of Locality is best realized through cache-awareness, i.e. by relying on the CPU caches especially the LLC which offers a good compromise between cache capacity and speed. The term cache-awareness compiles the necessary measurements for both NUMA and SMP architectures as all inefficiencies of both memory architectures are mostly caused by cache misses. So if we minimize cache misses, we mitigate the Memory Wall Problem.

Seamless Integration with Available Execution Environment Compatibility with the available hardware and software in the campus computing facilities and data centers is important to integrate DFV in their execution environment and get the benefits of computer virtualization. Campus computing facilities can have any memory architecture, and from DQDK's perspective, it is an unpredictable black box that it should be compatible with. Similarly, compatibility with other software without obstructing their execution is also taken by consideration by the DQDK framework.

Ease to Use The DQDK framework aims to hide all technical details like mitigating the Memory Wall Problem from the user. It provides opinionated easy-to-use APIs that abstract technical details from scientists. The framework aims to keep user setup efforts as minimal as possible, and instead auto-detects hardware configuration and autoconfigures the system for optimal performance.

4.2.2 Architecture

The DQDK framework is an easy-to-use lock-free multithreaded software framework optimized for scalability, cache awareness and compatibility. It is built on top of AF_XDP and eBPF/XDP and employs the best practices mitigating the Memory Wall Problem. The DQDK framework is implemented in $\sim$3200 lines of code, and is written in C and eBPF and utilizes the `libbpf` and `libxdp` libraries to run and manage AF_XDP sockets, the `libnuma` library for NUMA-aware processing, and the `libpthread` library for lock-free multithreaded processing.

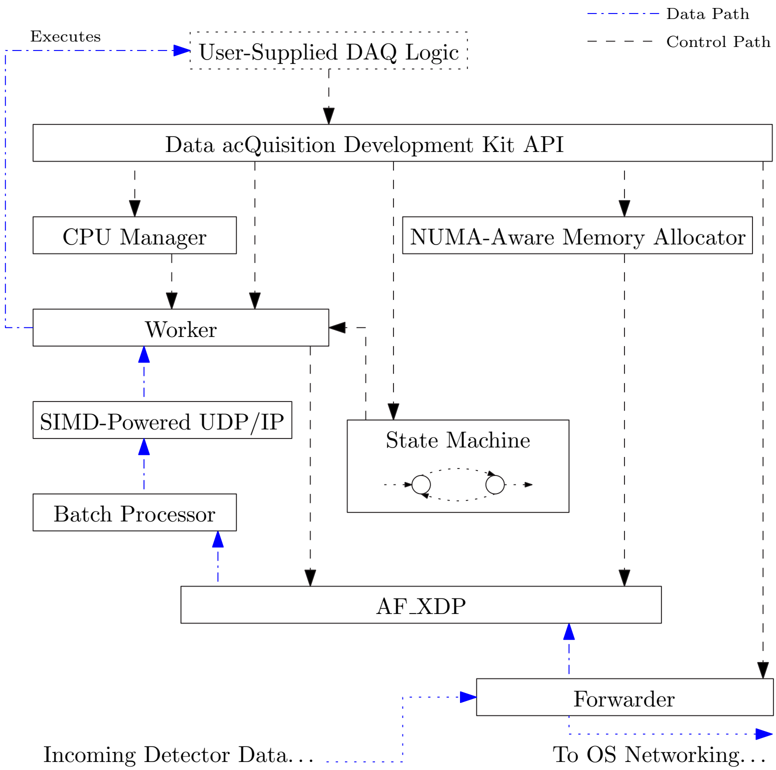


Figure 4.4: The Data Acquisition Development Kit Software Framework Architecture

Fig. 4.4 shows the architecture of the DQDK framework. The DQDK framework employs threads, hereby called workers, where each worker is running a AF_XDP socket for a unique NIC queue. A worker receives data through the AF_XDP socket and then executes a user-supplied DAQ logic over the received data. The workers are initialized and operated over 2 phases or paths: *The Data Path* and *The Control Path*. The data path is responsible for high-performance delivery of detector data from the network link to the DF, and the control path is responsible for configuring and setting up the worker and binding it to the data path. The control path is managed by an application programming interface (API) that initializes and passes the execution parameters to the internal components, e.g.

number of workers. It has 2 main responsibilities: allocation of computing resources necessary for the data path, and loading the data path components to the system to start the DAQ process. The memory and CPU resources are allocated through the NUMA-Aware Memory Allocator and the CPU Manager respectively. The NUMA-Aware Memory Allocator allocates `umem` buffer memory for each worker in the data path considering the NUMA architecture of the computer system. Elaborately, the allocator automatically detects the NUMA node owning the NIC, and then allocates a *locked* `umem` buffer from the local *huge pages* of the owner NUMA node for each DQDK worker. The memory allocator has 3 advantages: (1) NUMA-Aware Allocations: it allocates local memory and reduces cache misses resulting from allocating remote memory; (2) Huge Pages-based allocations: it prevents high latencies resulting from frequent translations of virtual addresses into physical memory addresses by allocating less but larger pages and thus efficiently utilizing the MMU; (3) Locked Memory: it prevents the operating system from swapping memory from main memory to far storage disks and thus cause performance degradation by violating the Principle of Locality. On the other hand, the CPU Manager is responsible for allocating and pinning CPU resources for the DQDK workers. It auto-detects the hardware configuration of the owner NUMA node and its CPU cores, it allocates for each worker 2 CPU cores (one for the application and another for the bottom-half interrupt handler of the NIC queue). The CPU cores allocation process is realized by isolating these CPU cores and pinning the application and the bottom-half interrupt handler thread to these cores in order to prevent sharing of CPU resources with the OS processes and other tasks. The performance advantages of the CPU manager are: (1) NUMA-aware CPU cores allocation that prevent cache misses due to remote memory accesses by allocating CPU and memory on the owner NUMA node; (2) it prevents interference from other tasks or the OS processes by isolating and dedicating the CPU core to the DQDK framework; (3) it protects memory locality by preventing frequent process context switching by configuring the OS's process scheduler to pin the DQDK framework threads only to the chosen dedicated CPU cores. After allocating resources, the DQDK API starts to load the data plane by creating the workers, the AF_XDP sockets and other components.

The data plane in DQDK consists of 4 main components: Workers, SIMD-powered UDP/IP, Batch Processor, AF_XDP Sockets, and the Forwarder. Because AF_XDP takes ownership of some NIC queues, it prevents all data arriving at those queues from reaching the targeted application. To solve this problem, we install an eBPF/XDP hook program that operates before AF_XDP called the forwarder. The forwarder preserves compatibility with other software applications by filtering detector data from other network data, e.g. data that belongs to other applications like slow control software in scientific infrastructure. The filtering decision is based on the network addresses of the detector electronics and that should be configured by the DQDK user. If received network data is detector data, it is directly forwarded to the AF_XDP sockets in the user-space as seen in Fig. 4.4. Otherwise, the forwarder returns the ownership of this network data to the OS which ensures it is received by its target application. The sockets are polled infinitely for data using the Batch Processor which takes care of managing the AF_XDP descriptors as described in Section 3.2.3 and passing data for further processing in the network protocol processing component: the SIMD-powered UDP/IP. The DQDK data plane uses the User Datagram Protocol (UDP) and the Internet Protocol version 4 (IPv4) protocols for data transfer. The choice of the UDP/IP protocol stack is based on 2 factors: being standards in computing facilities and data centers based on the Internet Engineering Task Force (IETF) Request for Comments (RFC) 768 [71] and 791 [72] respectively, and their lightweight implementations on detector electronics [42], [43], [73]. Unlike UDP, there exists network protocols that can detect data loss and compensate it through data retransmission like the Transmission Control Protocol (TCP) or RDMA. However, these protocols are considered resource-demanding because they require a lot of resources on the detector electronics. Moreover, this does not mean UDP does not provide any mechanisms to detect data corruption in a distributed infrastructure like DFV. The UDP protocol employs a checksum of its payload in its header, once a system receives a UDP packet, the system recalculates the checksum and compares it to the one available in the packet's header. If both checksums are equal, the data is considered correct, otherwise corrupt. However, recalculating the checksum takes time to finish and increases dispatching latency which causes data loss in high throughput use cases like scientific infrastructure.

To accelerate this issue, we provide a parallelized implementation that calculates the UDP checksum using Single Instruction Multiple Data (SIMD) instructions that are widely available in CPUs of data centers and decreases the time of heavy computations by applying one instruction on a vector of data rather than the traditional approach of applying one instruction on a scalar value. After protocol processing, the packets reach the worker which was initialized by the control plane. In the last step, the worker executes the DAQ logic supplied by the user via DQDK on the received data.

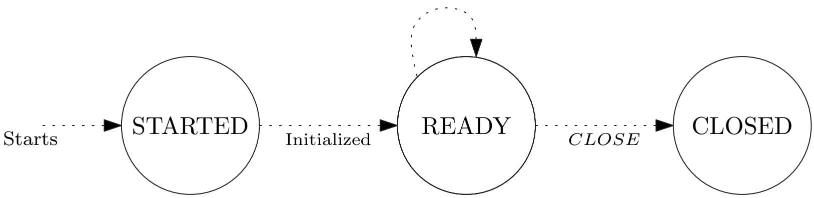


Figure 4.5: DQDK State Machine

The transition from the control path to the data path is maintained through a state machine. The job of this state machine is to ensure the data path is ready for processing detector data before the detector electronics starts sending them. This is achieved through a minimal protocol that synchronizes the slow control system and DQDK. The protocol should be implemented in the slow control system that is responsible for commanding detector electronics to start the scientific experiment and sending data. Fig. 4.5 shows the states and the transitions of DQDK's state machine. When the state machine starts, it has the state `STARTED` which means that DQDK is currently in the control path trying to allocate resources and to start the data path components. Once the initialization has been finished, DQDK automatically transitions to the `READY` state which means the data path initialization has been fully completed, and it is ready for online processing of detector data. The slow control system can send 2 commands to the DQDK state machine: `QUERY` and `CLOSE`. The `QUERY` command will return the current state of the state machine. The slow control system should only start the scientific experiment, when it receives this command returns the `READY` state. On the other hand, the `CLOSE` command allows the slow control system to finish the scientific experiment. The slow control system should wait for the state to change to `CLOSED` to ensure successful closing of the DQDK data path.

4.2.3 DQDK Program Example

```c
#include "dqdk.h"

dqdk_controller_t* controller
        = dqdk_controller_start();
dqdk_ctx_t* ctx = dqdk_ctx_init(...);

dqdk_for_ports_range(ctx, start_port, end_port);

for (u32 i = 0; i < ...; i++) {
    dqdk_worker_init(...);
}

dqdk_start(ctx);
dqdk_controller_wait(controller, ctx);
dqdk_waitall(ctx);

dqdk_ctx_fini(ctx);
```

Listing 4.1: DQDK Example

The DQDK main API only consists of 8 functions in the C programming language whose flow is shown in Listin 4.1. The program flow starts by initializing the DQDK state machine using the function `dqdk_controller_start`. The function will return a pointer to a `struct` of type `dqdk_controller_t` representing the state machine. This function is also responsible for ensuring the synchronization between the slow control system and DQDK is valid by waiting for the slow control system to connect. To initialize the control plane, the user should call the function `dqdk_ctx_init` and it will return a pointer to a `struct` of type `dqdk_ctx_t` that represents the DQDK control plane. Additional configuration to the control plane is given through the functions `dqdk_for_ports_range` and `dqdk_worker_init`. The function `dqdk_for_ports_range` adds the UDP ports range that the forwarder should use to filter detector data from other data. The function `dqdk_worker_init` initializes the DQDK workers, the user can initialize as many as the NIC has queues. The transition to the data plane (from `STARTED` state to `READY` state) happens with the function call `dqdk_start`. The user should call this function only after he has completed initialization of the data plane. The function `dqdk_controller_wait` listens on the network for commands (`QUERY` and `CLOSE`)from the slow control system. The function only exits when the command `CLOSE` is received. After receiving the `CLOSE` command, the user should call the function `dqdk_ctx_fini` to clean up all resource allocations done while in the control plane.

4.2.4 Design Limitations

The DQDK framework has some limitations related to its design and the techniques it employs. For example, DQDK requires the Ethernet protocol and does not support a diverse set of data transfer protocols (e.g. TCP, IPv6) other than UDP and IPv4, but it is open to extension. The current IPv4 implementation also does not support packet fragmentation which requires manually increasing the Ethernet MTU when receiving data that is larger than the usual MTU (i.e. $1514\,\mathrm{B}$). It does not support sending Ethernet frames larger than $4\,\mathrm{kB}$. The DQDK framework uses CPU cores (2 cores per NIC queue) to achieve zero-loss DAQ, and consequently requiring higher throughput may need using more than 1 NIC queue which also increases the number of the needed CPU cores.

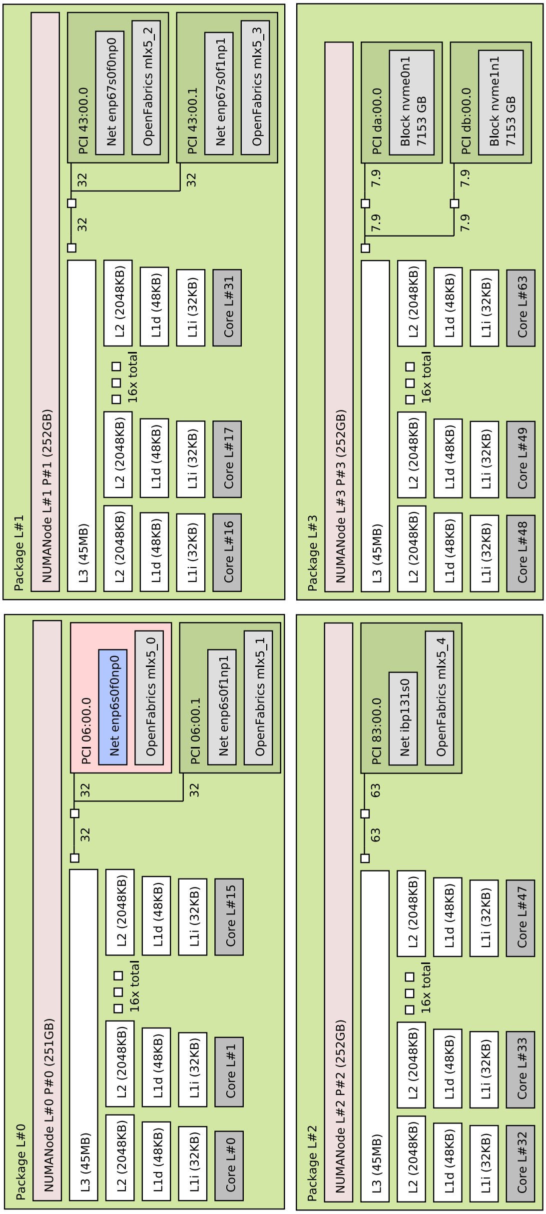


Figure 4.6: The Memory Architecture of the DuT machine. The used NIC is in blue.

4.3 Performance Evaluation

This section is dedicated to study the capabilities of the DQDK framework and its raw data transfer performance. Understanding the raw data transfer performance will help us know the performance capabilities of the framework without the added latency of the user-supplied logic. Thus, this section does not aim to provide a real-world scenario of DQDK.

4.3.1 Experimental Setup

To study the performance of the DQDK framework, we set up an experimental setup consisting of 2 server machines: a packet generator machine and the Device under Test (DuT). Both servers are connected using a $100\,\mathrm{Gbit/s}$ link, and run Ubuntu 24.04 LTS with Linux kernel 6.8.

The packet generator machine is a computer workstation that simulates a scientific detector by generating UDP packets. As this stage of evaluation focuses only on data transfer not on online processing, the payload of the generated UDP packets is not important, and thus, we assign it to random data. It uses a 4-cores Intel(R) Xeon(R) CPU E5-1630 v3 @ 3.70GHz with L3 cache of $10\,\mathrm{MB}$ and has $128\,\mathrm{GB}$ of main memory. For reproducible research results, we assign the CPU C-state that controls sleeping behavior to C0, and we disable the CPU P-State control which reduces power usage by the CPU. The machine runs a $100\,\mathrm{Gbit/s}$ NVIDIA-Mellanox ConnectX-6 NIC over PCIe 3.0 x16.

The DuT is a computer system that runs the DQDK framework with empty user-supplied logic. Its memory architecture is shown in Fig. 4.6. The system has $1\,\mathrm{TB}$ of DRAM main memory balanced over 4 NUMA nodes. The NUMA nodes are connected in a peer-to-peer mode and each has a 16-core Intel(R) Xeon(R) Platinum 8444H CPU @ 2.9GHz with Level 1 Data (L1d) Cache of $3\,\mathrm{MB}$ and Level 1 Instruction (L1i) cache of $2\,\mathrm{MB}$ and Level 2 (L2) cache of $128\,\mathrm{MB}$ per core, in addition to a shared Level 3 (L3) cache of $180\,\mathrm{MB}$. The system has a $100\,\mathrm{Gbit/s}$ NVIDIA-Mellanox ConnectX-6 Dx NIC connected over PCIe 4.0 x16 to the PCIe Root Complex of NUMA node 0.

4.3.2 Investigated Metrics

The metrics we look at to understand the performance of the DQDK framework are throughput in Gbps, dispatching latency, and cache miss rate. The throughput is calculated both in Gbps and in packets per second (PPS). The PPS throughput is important to understand how dispatching latency added by processing packets headers impact overall performance. The dispatching latency is calculated by calculating the difference between the moment the packet reaches the user-supplied logic and the moment it reaches the NIC hardware using the NIC hardware timestamps. The cache miss rate provides insights on the impact of memory hierarchy on performance and the memory wall problem and how efficiently a program is being executed in the CPU. More cache misses lead to more accesses to DRAM memory and degraded performance due to the memory wall problem. The cache miss rate is measured using the *perf* Linux utility.

4.3.3 Benchmarking DQDK

Fig. 4.7 shows the throughput in gigabits per second (Gbps) of fully-optimized DQDK in comparison to unoptimized DQDK as function of the MTU. We specifically target disabling the CPU manager and the memory allocator optimizations in DQDK to study the impact of the absence of these optimizations on performance. Disabling the CPU manager means that the OS and other programs can interfere with the execution of the DQDK framework. We measure DQDK's throughput in this case by running a system update utility along with DQDK. Disabling the memory allocator allows DQDK to allocate memory without considering the NUMA architecture of the system, and consequently allocating remote memory. We compare the results to the line rate of the $100\,\mathrm{Gbit/s}$ link (maximum bandwidth) and to that of traditional OS networking which is commonly used for data transfer in DAQ in scientific infrastructures. For DQDK measurements, we only consider up to the maximum supported MTU by AF_XDP.

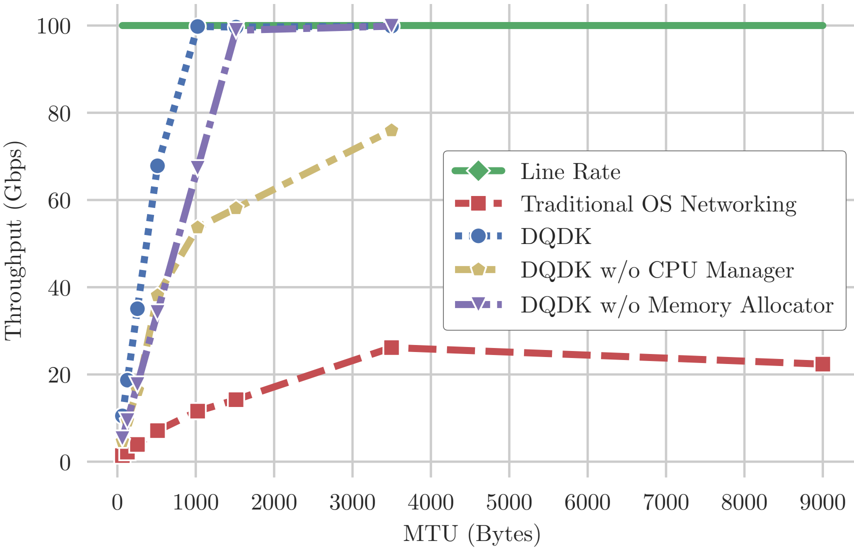


Figure 4.7: DQDK vs Traditional OS Networking Throughput in Gbps

The DQDK framework achieves the best throughput by saturating the 100 Gbit/s at MTU 1024 B only while traditional OS networking cannot saturate the link even at 9000 B the largest possible MTU in the Ethernet protocol. The importance of DQDK's optimizations can be seen by looking at the throughput of DQDK while the CPU Manager or the memory allocator are disabled. While the CPU manager is disabled, DQDK's throughput degrades to a maximum of $\sim$80 Gbit/s losing the ability to saturate the link even at large MTUs, e.g. at 3498 B. While the memory allocator is disabled, DQDK's overall performance is degraded, but it can still saturate the link at a slightly larger MTU of 1514 B compared to 1024 B of a fully optimized DQDK. This is an important measurement to understand how DQDK will behave in case remote memory was needed. An example of such a case in DAQ is the need to allocate memory larger than the available capacity of one NUMA node, in such a case, DQDK can allocate remote memory from remote NUMA nodes to expand its memory buffers with a slight performance overhead.

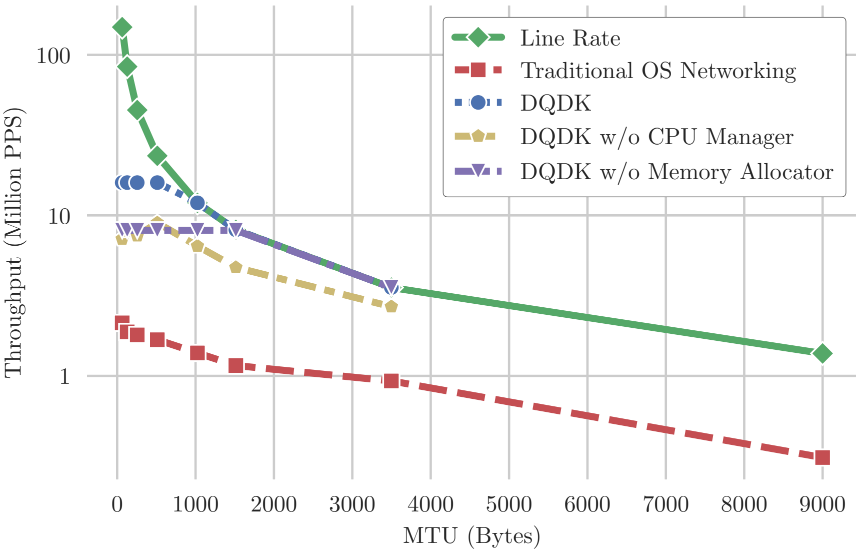


Figure 4.8: DQDK vs Traditional OS Networking Throughput in Millions Packets per Second on Log-scale

A different perspective on DQDK throughput can be seen in Fig. 4.8 where we measure throughput in millions packets per second (MPPS) as function of MTU in comparison to the maximum packets per second that is theoretically possible for an MTU. The DQDK framework throughput is constant at 16 MPPS for MTUs ranging between 64 B and 512 B, and it only drops when it saturates the 100 Gbit/s link at 1024 B or larger. The invariance of throughput at 16 MPPS between 64 B and 512 B signifies that the DQDK framework is bounded by 16 MPPS regardless of the MTU size. Similarly, the throughput of DQDK without the memory allocator optimization is constant at 8 MPPS (half of that of fully-optimized DQDK) for MTUs ranging between 64 B and 1514 B, but it is enough to saturate the 100 Gbit/s link at 1514 B or larger. In contrast, DQDK without the CPU manager and traditional OS networking cannot saturate the link at all

MTUs and can only achieve a maximum throughput of 9 MPPS and 2.14 MPPS respectively.

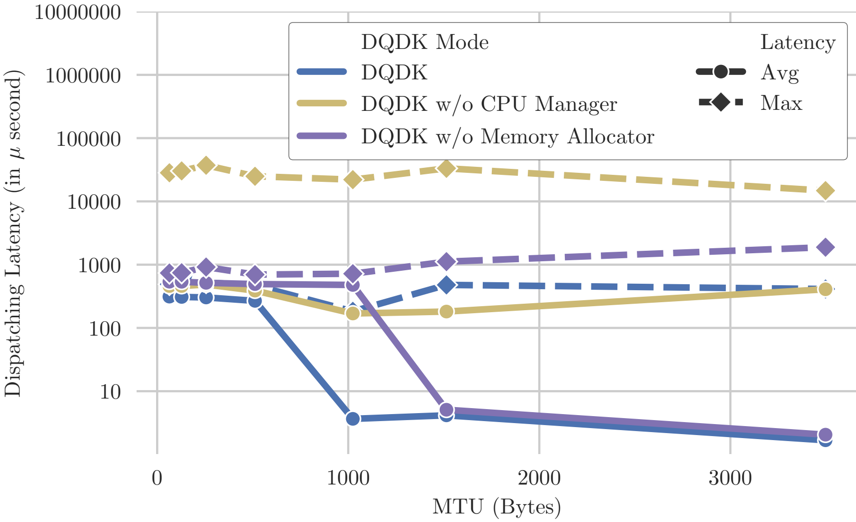


Figure 4.9: DQDK vs Traditional OS Networking Dispatching Latency in µs on Log-scale

To understand the performance degradation while disabling DQDK optimizations, we measure the dispatching latency and the cache miss rate. Fig. 4.9 shows the log-scale of the average (solid lines with circle markers) and maximum (dashed lines with diamond markers) dispatching latency (in μs) of the fully optimized DQDK in comparison to DQDK while disabling the CPU manager or the memory allocator as function of the MTU. At small MTUs and where there is many small packets per second, the fully optimized DQDK achieves the lowest dispatching latency in average and maximum values compared to the DQDK framework without the CPU manager or the memory allocator. For example, at 64 B MTU, DQDK achieves an average latency of 312 μs and a maximum latency of (489 μs) in comparison to an average of 459 μs and a maximum of 28 ms for DQDK without the CPU manager, and to an average of 537 μs and a maximum of 738 μs for DQDK without the memory allocator. The same behavior is also common in larger MTUs. The largest maximum latency distribution is achieved by DQDK without the CPU manager. On the contrary, the largest average latency distribution is achieved by DQDK without the memory allocator between 64 B and 1024 B MTUs.

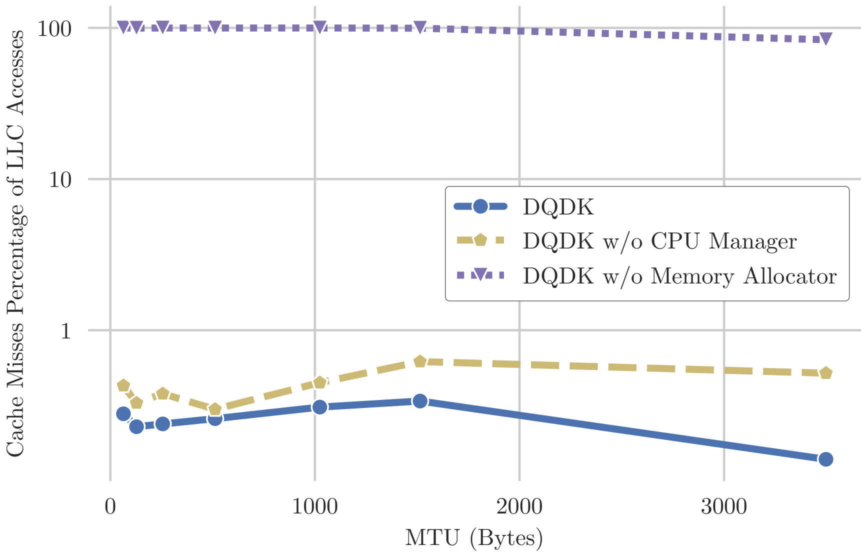


Figure 4.10: DQDK's Cache Misses Percentage for Different Optimizations

To understand the latency statistics, we measure the percentage of cache misses from all LLC accesses as function of MTU. The lowest cache miss rate is achieved by the fully optimized DQDK. The DQDK framework without the memory allocator produces the highest cache miss rate that monotonically declines from 99.78% at 64 B to 83.51% at 3498 B. These high number of cache misses results from accesses to remote memory when the NIC is owned by one NUMA node and DQDK runs on a remote NUMA node which forces the system to move data from the owner CPU node to the remote NUMA node through the CPU interconnect resulting in cache coherency problems and consequently many cache misses. Although the DQDK framework without the CPU manager has a slightly larger cache miss rate than fully optimized DQDK because the OS process scheduler migrates

DQDK from one CPU core to another, it is not proportionate to the throughput degradation or the latency increase. We attribute this performance drop to the fact the system update utility is competing with DQDK for CPU resources which yields less CPU time for DQDK and consequently lower performance.

4.3.4 Scaling DQDK

The DQDK framework aims to realize DFV for in most scientific infrastructure. The fact that DQDK works on top of the standard and the wide-spread Ethernet protocol reinforces this goal. However, scalability is an important factor to realize DFV on scientific detectors of all performance requirements. For this purpose, we target understanding the scalability of DQDK.

Since DQDK's throughput is bounded by the number of packets per second, we choose to study its scalability at MTU 64 B where we can study the throughput up to 148.8 MPPS. As we have proven in the previous sections, DQDK's throughput is independent of the received MTU size. Consequently, the scalability studies on 64 B are identical for larger MTUs.

Fig. 4.11 shows the throughput of DQDK in MPPS as function of used NIC hardware queues and compared to 148.8 MPPS, the maximum possible throughput for 64 B MTUs. By DQDK's design, every NIC hardware queue consumes 2 dedicated CPU cores (one for bottom-half interrupt handler, and another for the DQDK framework thread). For example, 1 NIC hardware queue will consume 2 CPU cores, 8 CPU cores will consume 16 CPU cores. For 9 and 10 NIC queues, the framework will consume 18 to 20 CPU cores which are not available in a single NUMA node in our experimental setup. To prevent using remote NUMA nodes, we estimate the throughput value for only those 2 NIC queues based on the previously measured throughput on previous values of NIC queues.

As seen in Fig. 4.11, the throughput of DQDK scales linearly as we increase the number of NIC hardware queues (and subsequently the number of CPU cores). At 10 NIC queues, we reach 148.8 MPPS, the maximum throughput of the

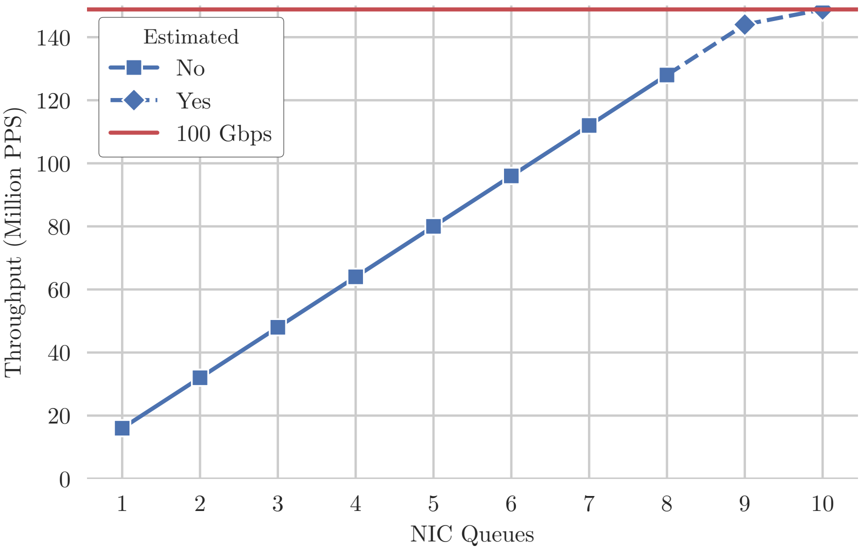


Figure 4.11: DQDK's Scalability: Throughput as function of NIC Queues

100 Gbit/s link. The parallelization of DQDK by increasing the NIC hardware queues supports the framework scalability. This is possible due to DQDK's shared-nothing design that prevents bottlenecks caused by thread synchronization to access shared data.

Although parallelization does not cause a bottleneck, the DQDK framework may face a memory bottleneck due to the bandwidth saturation of the memory channels connecting the CPU to the main memory. However, this can be easily mitigated through scaling to remote NUMA nodes which can provide additional memory bandwidth either by using the same NIC or using multiple NICs distributed over multiple NUMA nodes. Another way to scale memory bandwidth is to scale using multiple computer systems in the campus computing facility.

4.3.5 Performance Impact of Secure DMA

The IOMMU enhances the security of DMA through virtual memory addresses that are private to the NIC device. This section studies the impact of secure DMA access on dispatching latency and through in DQDK. We specifically study 3 different IOMMU setups: IOMMU fully configured with IOVA, IOMMU configured with pass-through, and by disabling IOMMU. The most secure of the setups is a fully configured IOMMU where the IOMMU assigns IOVA for the NIC and performs translations between IOVA and addresses of physical memory. IOMMU with Passthrough is a setup where an IOMMU is turned on but the IOVAs are not used. The third setup completely turns off the IOMMU and let the device use the physical memory addresses without any additional features.

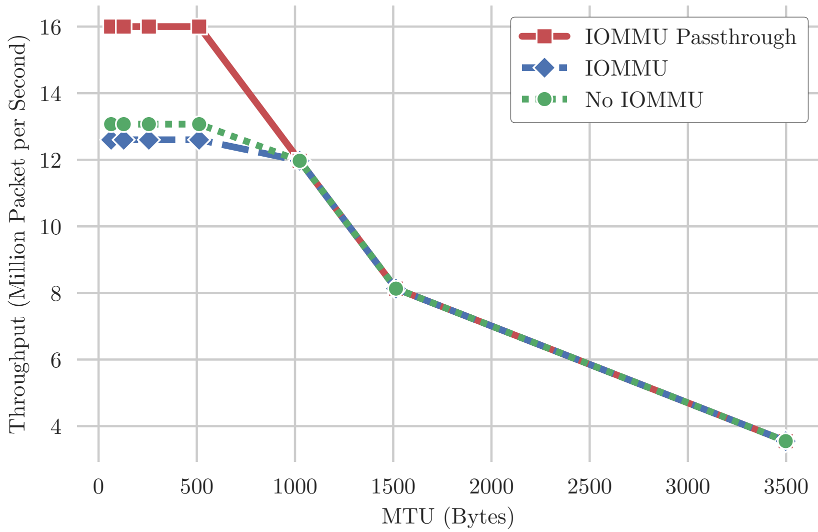


Figure 4.12: DQDK Throughput in Million Packet per Second for Different IOMMU Setups

Fig. 4.12 shows DQDK's throughput for the 3 IOMMU setups. We notice that when IOMMU is fully enabled, address translations from IOVAs to physical addresses impact performance and lower DQDK's throughput to a maximum of

12.6 MPPS. Disabling IOMMU completely has a performance advantage raising the maximum throughput to 13.07 MPPS. The best performance can be obtained by configuring the IOMMU to pass-through mode which enables IOMMU but does not use address translation between IOVA and physical addresses. When IOMMU is set to pass-through, the maximum throughput can reach up to 16 MPPS.

I/O MMU Mode	Min	Median	Mean	Std. Dev.	95%	99%	Max
Disabled	1.722	16.6	20.97	67.67	31.18	35.42	1102.95
Enabled	1.5	17.03	18.16	8.96	32.46	36.63	173.19
Pass through	2.16	15.03	19.68	67.26	28.07	30.78	1001.42

Table 4.1: Dispatching Latency Distribution (in µs) under Different IOMMU Setups

Throughput measurements aligns with latency measurements shown in Table. 4.1 and Fig. 4.13. Table. 4.1 shows the statistical distribution of dispatching latency under the 3 IOMMU setups. The table shows that when IOMMU is enabled, the distribution of the dispatching latency has low variance by ranging between 1.5 µs and 173.19 µs at maximum with mean value 18.16 µs and standard deviation 8.96 µs. However, its 95% and 99% percentiles are higher than when IOMMU is completely disabled or set to pass-through mode.

To understand how latency distribution cause a throughput drop, we look at the empirical cumulative distribution function (ECDF) between 0 µs and 50 µs where more than 99% of the latency samples are as shown in Fig. 4.13. The figure shows that when IOMMU is set to pass-through mode, its ECDF is above those of when IOMMU is completely disabled or completely enabled. Therefore and in terms of performance, we can say that IOMMU Pass-through is better than disabling IOMMU or completely enabling IOMMU. The reason of this phenomenon is that when IOMMU is completely disabled, there is an overhead imposed by the OS to mitigate performing DMA to non-contiguous memory which is usually handled by IOMMU. In the absence of an IOMMU, the OS does not perform DMA directly to the user-space buffers of DQDK but instead uses intermediary

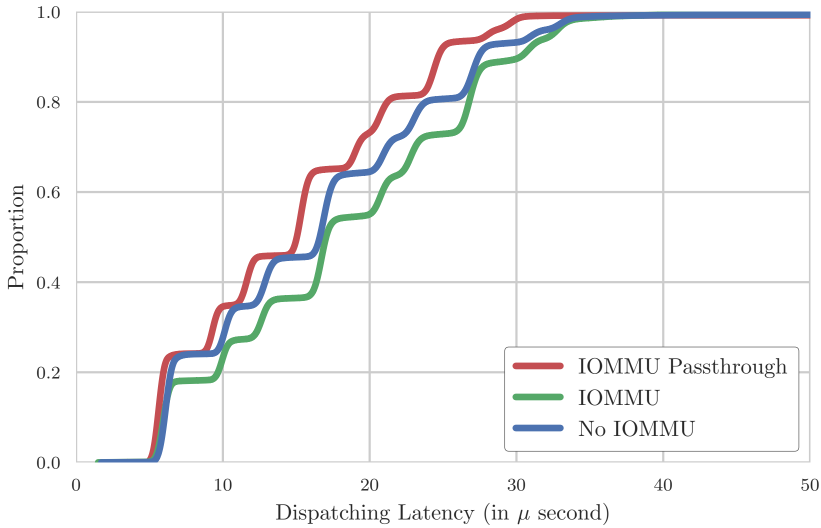


Figure 4.13: Partial Empirical Cumulative Distribution Function of Dispatching Latency (in µs) for Different IOMMU Modes limited from 0 µs to 50 µs

contiguous buffers, called "bounce buffers", which are considered safe for the actual DMA operation. After DMA to bounce buffers, the OS copies the data to the non-contiguous user-space buffers of DQDK. This eliminates the advantages of DQDK's zero-copy data transfer and leads to throughput degradation.

4.4 Conclusion

We presented the DQDK framework, a software library for high-performance and reliable DAQ. The DQDK's design is based on the Principle of Locality to mitigate the Memory Wall Problem. It exploits the computer and the memory architecture to employ the full power of the computer system without bottlenecks. It is available as on open-source project on GitHub: `https://github.com/kit-ipe/dqdk`.

The DQDK framework is bounded by the number of processed packets per second and its throughput is not affected by their sizes. As seen in the previous sections, the employed optimizations in DQDK are important to realize the full power of AF_XDP and to prevent data loss caused by interference in campus computing facilities. The framework is linearly scalable by using more CPU cores. We expect that it is also scalable by expanding the processing to remote NUMA nodes in addition to the owner NUMA node to overcome memory bandwidth bottlenecks, or by expanding to multiple computer systems in the campus computing facility. The user is responsible for the design and the performance of the logic supplied to the framework, we, however, recommend reusing the memory allocator of the DQDK framework through the provided APIs, i.e. `dqdk_malloc` and `dqdk_huge_malloc`, for better performance.

It is important to notice that different memory architectures may yield different performance results. For example, AMD CPUs divide the LLC on the CPU cores, and thus instead of having one big shared LLC, they have multiple small LLC distributed over the available CPU cores. Some CPU features may also yield different performance results like Intel's Data Direct Input/Output (DDIO) which allows NIC to do DMA directly to the CPU cache instead of DMA to the main memory. This feature is included in our experimental setup and can contribute to high performance. Alas, AMD CPUs do not have an equivalent for DDIO, and thus DMA is only possible to main memory in AMD setups.

5 Virtualized Data Acquisition Functions

Computing facilities isolate guest applications using computer virtualization technologies. Their end goal is to provide a highly available and scalable secure environment where all guest applications of diverse requirements can run together without impacting the host system or each others. However, availability, security and isolation of computer virtualization come at a cost.

The goal of this chapter is to quantify the performance impact on DFV induced by computer virtualization. We look at how different computer virtualization setups impact throughput of DQDK. In specific, we look at software-based and hardware-based virtualization of computer networks in virtual machines and containers.

5.1 Computer Networks Virtualization

Computer virtualization technologies like virtual machines and containers share all computing resources of the host with all available guests. If a guest completely takes over a computing resource, e.g. a NIC, other guests get deprived of these resources. Therefore, computer virtualization technologies employ software-based or hardware-based network virtualization mechanisms to share a NIC between multiple guests.

Software-based network virtualization build on top of host network stacks discussed in Section 2.2.3 by adding a virtual NIC device. The virtual NIC device is a piece of software that acts as a tunnel between the host and the targeted guests.

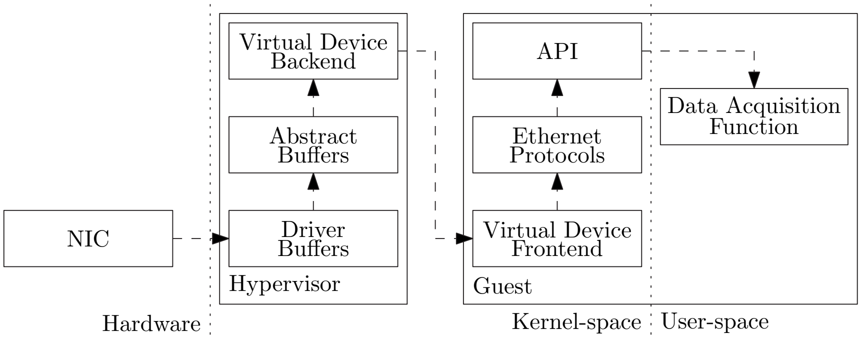


Figure 5.1: Abstract Model of Software-based Networking Devices in Computer Virtualization Environments

Thus, the virtual NIC device has 2 ends: the backend in the host, and the frontend in the guest. The backend is responsible for receiving guest data, deciding which guest it should go to, then forwarding the data to the appropriate target guest. The frontend of the virtual NIC device is linked to the guest and emulates a physical NIC to provide the guest application with networking resources. This way the host virtualizes the physical NIC card and multiple guests can use it.

On the contrary, hardware-based network virtualization relies on NIC hardware features to virtualize computer networking on the host. One prominent example is Single Root Input/Output Virtualization (SRIOV). With SRIOV, administrators of the host can configure the PCIe-based NIC to present itself as multiple virtual NICs, called virtual functions, while preserving the original physical function of the physical NIC. The physical NIC is responsible in this case to perform switching among multiple virtual functions and the physical function in order to direct network data to the correct target.

5.1.1 Virtual Machines

Virtual machines support both software-based and hardware-based network virtualization. For hardware-based technologies, it is enough to allocate a new virtual function using SRIOV and then transfer its ownership to the guest virtual machine using PCIe pass-through. The ability to run AF_XDP and consequently DQDK on SRIOV virtual functions is determined by the physical NIC driver. SRIOV does not impose any limitations on AF_XDP.

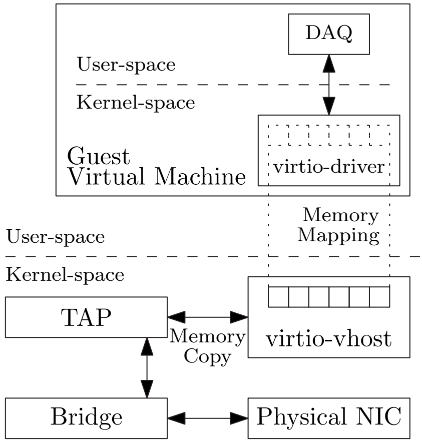


Figure 5.2: virtio-vhost Architecture

For software-based network virtualization, virtual machines use a special virtual NIC device called *virtio-vhost*. The architecture of virtio-vhost is shown in Fig. 5.2. The virtio-vhost is a software NIC that relies on the cooperation of the host and the guest OSs. The host creates the virtual device backend as a file on the host file system. When the OS in the guest OS starts, the virtio driver (or the virtual device frontend) maps the virtio-vhost file from the host to its guest memory. The job of the mapped file is to emulate a NIC in the guest by sharing memory between the host and the guest. Exchanging event, like interrupt and event handling, between host and guest in virtio-vhost is then performed by means of 2 file descriptors.

The virtio-host architecture creates only a protocol for the host and guest to communicate but does not provide the guest with network access, e.g. communication to external computers or the internet. For this purpose, the host has to maintain 2 additional components: the bridge, and the TAP interface. Both of which are virtual utility devices that link the physical NIC to the virtio-vhost virtual device. The bridge device acts as a switch between the physical NIC and the virtio-vhost virtual device. However, in order to link the virtio-vhost's file to the bridge device, the TAP interface is used which drives communication between the virtio-vhost file and the bridge. The virtio-vhost driver supports AF_XDP copy mode only.

5.1.2 Containers

Containers have different components to realize software-based and hardware-based network virtualization. Since containers are just host OS processes in isolation *namespaces*, they can also have restricted access to the host network. In this scenario, the guest container can use the physical NIC to send and receive data but its capabilities to configure it are extremely limited. In this scenario, DQDK can have access to some NIC queues while keeping others for other guest applications.

Containers can also have software-based network virtualization. It is known as virtual Ethernet (veth). Fig. 5.3 shows the architecture of veth. A veth is a software virtual NIC that has 2 ends: one is in the host OS, and the other, called *veth peer*, is assigned to the file system namespace of the guest container. Similar to virtio-vhost, veth does not provide the guest container with network access outside the host without a bridge between the physical NIC and the veth backend. The veth driver only supports AF_XDP copy mode.

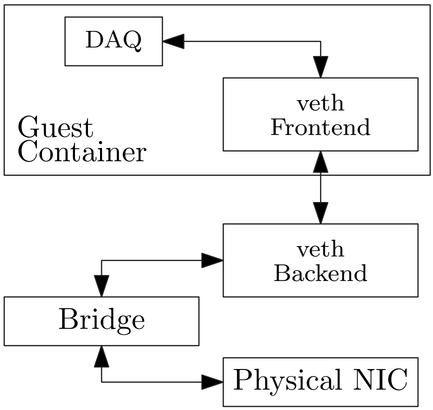


Figure 5.3: Virtual Ethernet Architecture

5.2 Porting DQDK to Virtual Environments

In order to run DQDK in a virtual environment, e.g. a virtual machine or a container, some adjustments to the environment configurations are required.

In a virtual machine, DQDK's CPU Manager and Memory Allocator are directly affected by virtualizing resources. When a virtual machine is launched, the hypervisor create virtual CPUs that can run on any CPU core on the host machine which means that the DQDK process running on a virtual CPU may be sharing a physical CPU core with another process. This breaks DQDK requirement to benefit from pinning and isolating the physical CPU core to enhance performance. Similarly, a virtual machine is unaware of the NUMA memory architecture on the host system, and thus DQDK's memory allocations may lose their NUMA-awareness property. This also may lead to performance degradation. However, there exists mitigations to prevent such behaviors: the administrator can configure the hypervisor to run a DQDK virtual machine on specific physical CPU cores and on a specific NUMA node of the host system. This is enough to get the full throughput in DQDK. These configurations are easily available through `libvirt`, a common utility to launch virtual machines on Linux-based hypervisor systems. In case of using SRIOV with a virtual machine, the virtual function has to be created beforehand. Some physical NICs require a system reboot to enable SRIOV or even to increase the number of possible virtual functions.

In a container, the constraints are lower than a virtual machine. In a container environment, DQDK still have access to read most of the system configurations like detecting the NUMA node of a physical NIC or even the ability to transparently detect the CPU core architecture. Certain system capabilities that provide the privilege to configure specific host parameters or to run AF_XDP may be needed. Examples of such capabilities on Linux host systems are: `CAP_NET_ADMIN` and `CAP_NET_RAW` that are needed to run AF_XDP. However, these capabilities can be mitigated in a virtual environment by offloading them to the container orchestrator as explained in Section 2.3.2. For example, the requirement of the

`CAP_NET_ADMIN` and `CAP_NET_RAW` capabilities can be mitigated by offloading creating the AF_XDP socket to Kubernetes through its AF_XDP plugin [74].

5.3 Performance Evaluation

This section studies the performance impact resulting from using software-based and hardware-based network virtualization technologies. We study the performance impact in both: virtual machines and containers.

5.3.1 Experimental Setup

To study the performance of the DQDK framework, we use the same experimental setup used to evaluate DQDK in Chapter 4 consisting of 2 server machines: a packet generator machine and the Device under Test (DuT). The DuT, however, runs the KVM hypervisor on Linux as a renowned hypervisor used in computing facilities for virtual machines. For container tests, we use *podman* container runtime. We test the DQDK framework with empty user-supplied logic inside the container and inside the virtual machine, each seperately.

For virtualization, we use *libvirt* to configure the KVM VM. We manually set the mapping between virtual CPUs inside the VM and the physical CPU cores in the host. We assign the VM to run on the NUMA node of the NIC. For network virtualization, we use virtio-vhost for software-based network virtualization and SRIOV with PCIe Passthrough for hardware-based network virtualization.

The configuration in containers is transmitted through the command line options of *podman*. For simplicity, we provide the required capabilities in the configuration. We use *veth* as software-based network virtualization and 1 NIC queue for hardware-based network access. Other guests can still have up to 127 NIC queues of the same interface in our setup.

We compare all setups to bare-metal performance i.e. DQDK without computer virtualization running directly on host OS.

5.3.2 Results

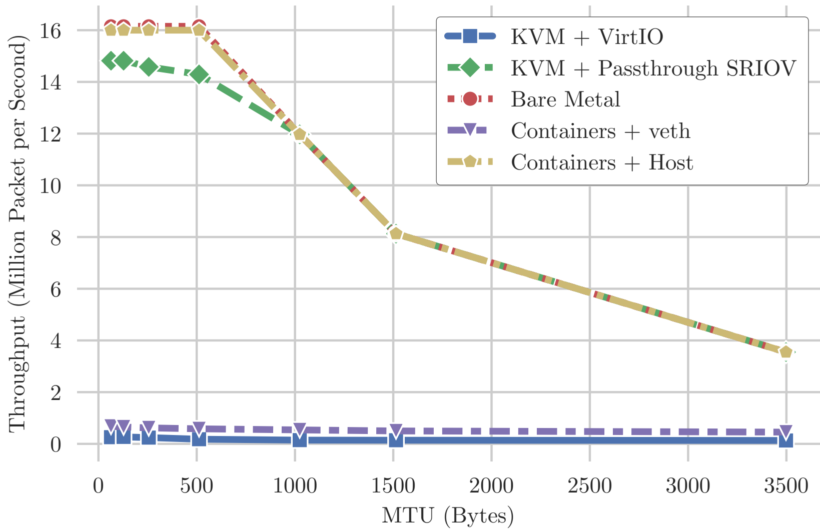


Figure 5.4: DQDK Throughput in Millions Packets per Second for Different Virtualization Setups

Fig. 5.4 shows DQDK's throughput in Millions Packets per Second for different computer virtualization setups in comparison to its performance on bare metal. The setup *Containers + veth* shows DQDK throughput running in a podman container while using veth for software-based network virtualization. Similarly, we run DQDK while exposing host network in a container, this setup is denoted by *Containers + Host*. For virtual machines, we denote the setup of a KVM VM with software-based virtio-vhost network virtualization by *KVM + VirtIO* and the setup of a KVM VM with a passthrough-SRIOV virtual function by *KVM + Passthrough SRIOV*.

For software-based network virtualization, we notice that the maximum throughput is achieved by *Containers + veth*. It starts at 0.67 MPPS at MTU $64\,\mathrm{B}$ and then starts to degrade to 0.45 MPPS at MTU $3498\,\mathrm{B}$ due the lack of zero-copy support for AF_XDP in veth. *KVM + VirtIO* has throughput with 0.25 MPPS at $64\,\mathrm{B}$ and degrades to 0.13 MPPS at MTU $3498\,\mathrm{B}$ due to lack of AF_XDP zero-copy in virtio-vhost. In comparison to *Containers + veth*, *KVM + VirtIO* performs lower due to the complexity of its virtualization setup. Packets in *KVM + VirtIO* have to traverse a longer way to reach the DF: they have to pass through 2 OS kernels (host + guest kernels compared to only the host kernel in veth case) and they have to pass through more network virtualization components (Bridge - TAP - virtio-vhost compared to only Bridge - veth in containers). Even in best cases of software-based network virtualization, a maximum throughput of $12.78\,\mathrm{Gbit/s}$ (0.45 MPPS at MTU $3498\,\mathrm{B}$) is not enough to perform $100\,\mathrm{Gbit/s}$ DAQ.

Hardware-based network virtualization performance is comparable to that of bare metal outperforming software-based network virtualization. Exposing the host network to the container environment enables DQDK to have identical performance to that of bare metal. Both *Bare Metal* and *Containers + Host* scores a constant 16 MPPS until the $100\,\mathrm{Gbit/s}$ link gets saturated at MTU $1024\,\mathrm{B}$. The *KVM + Passthrough SRIOV* setup shows a slight performance drop to 14.82 MPPS at MTU $64\,\mathrm{B}$, but it is stable to saturate the $100\,\mathrm{Gbit/s}$ link at MTU $1024\,\mathrm{B}$. This is a result of the overhead of CPU and network virtualization. CPU virtualization leads to increasing context switches between host and guest (i.e. VM Entering and Exiting), depriving the guest from some CPU resources. This is cannot be avoided because the guest VM might require non-virtualized resources that are only accessible by the hypervisor especially those related to interrupt handling like the `CLI` and the `STI` instructions in x86 systems that are responsible for handling top-half interrupts [75].

5.4 Conclusion

In this chapter, we show how DQDK performs under different computer virtualization setups. We survey possible software-based and hardware-based network virtualization technologies for both KVM-based virtual machines and containers. We also provide insights about porting DQDK to such environments and what changes are required.

Containers in addition to host networking provide the highest throughput that is identical to DQDK's performance on bare metal. Some campus computing facilities may only support virtual machines, for such infrastructures, we recommend SRIOV-based network virtualization with PCIe Passthrough.

If the required throughput is around $10\,\mathrm{Gbit/s}$, DQDK in containers and using veth can be also used.

Future works may evaluate DQDK in overlay networks that use software switches and are frequently used in computing facilities. Examples of such networks are: OpenvSwitch (used for both virtual machines and containers), Cilium (mainly for containers), etc.

6 Use-case: The TRISTAN Upgrade at KATRIN

To prove its applicability, we apply DFV on a use-case from a real-world DAQ system of an operating scientific infrastructure. A good example of an operating scientific infrastructure is the KATRIN infrastructure at KIT which is operated by the KATRIN collaboration from 18 institutes in 7 countries in Europe and Northern America. The KATRIN infrastructure investigates the mass of neutrino and is powered by a tritium beta-decay source. An upcoming upgrade to the KATRIN infrastructure is the new high data rate TRISTAN detector with the goal to prove the existence of sterile neutrino empirically.

In this chapter, we discuss the operational challenges of KATRIN. We argue that DFV is a valuable addition to the KATRIN infrastructure by taking the TRISTAN upgrade as an example. We study the systematic requirements of the TRISTAN upgrade. Finally, we provide insights on the challenges and the results of adopting DFV in the TRISTAN upgrade using the DQDK framework.

Publication

Parts of this chapter are published in IEEE Transactions on Nuclear Science and 24th IEEE Real-Time Conference:

J. Mostafa, D. Tcherniakhovski, S. Chilingaryan, M. Balzer, A. Kopmann and J. Becker, "100 Gbit/s UDP Data Acquisition on Linux Using AF_XDP: The TRISTAN Detector," in IEEE Transactions on Nuclear Science, doi: 10.1109/TNS.2024.3452469.

6.1 The KATRIN Infrastructure

The KATRIN infrastructure is a scientific instrument at the Karlsruhe Institute of Technology (KIT), Germany to study the mass of neutrino particles via ultrahigh precision measurements (sensitivity of $0.2\,\mathrm{eV}/C^2$ [76]) of the kinematics of electrons from beta-decay. It exploits the law of energy conservation where the amount of energy in a beta decay is constant i.e. the transition energy that is shared by the released electron, the neutrino, and the nuclear recoil of a beta decay. It then calculates the neutrino mass by comparing the maximum energies of resulting electrons in addition to the mass of the known nuclear recoil to the transition energy of the beta decay. The first results of KATRIN were published in 2019, and its latest results concluded that a neutrino mass is smaller than $0.45\,\mathrm{eV}/C^2$ [77].

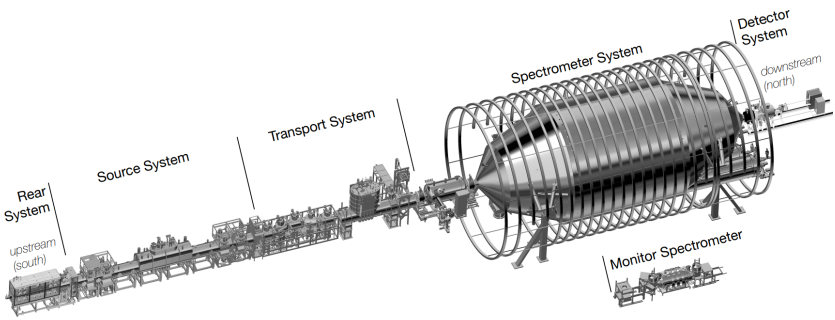


Figure 6.1: The Huge KATRIN's Experimental Setup consists of 6 main systems.
Source: The Design, Construction, and Commissioning of the KATRIN Experiment [40]

As shown in Fig. 6.1, the KATRIN infrastructure is composed of 6 main subsystems that are responsible for adequate transportation and separation of electrons from the source subsystem to the detector system through the transport, rear, spectrometer, and monitor spectrometer subsystems. The experiment's data of the KATRIN infrastructure is collected by the detector subsystem and its DAQ system. The length of this experimental setup exceeds $70\,\mathrm{m}$ and is visually dominated by the spectrometer system with a diameter of $10\,\mathrm{m}$ and length of $23.3\,\mathrm{m}$. This infrastructure, in its turn, imposed operational challenges that can be summarized as follows:

Numerous Components Developed by an International Team As shown in Fig. 6.1, KATRIN has a medium to large experimental setup of 6 main systems and spreads over multiple buildings in the North Campus of KIT in Eggenstein-Leopaldshafen, Germany. Four different control and DAQ systems control and collect data from around 10000 sensors: ZEUS [78], Siemens WinCC systems, custom National Instruments LabVIEW-based systems, and an Orca DAQ system [79]. To handle the data of these systems, scientists and engineers use 3 different storage mediums: block storage to store the detector's data, and Microsoft SQL Server and MySQL database systems to store slow control systems data. The experiment's data should be accessible by scientists in all participating countries to perform data analysis and extract scientific results and conclusions. As a result, KATRIN architecture is complex and accommodates diverse software components imposing the data management challenge of heterogeneous data sources and storage destinations [40].

Highly Dynamic Infrastructure KATRIN has been running for 20 years. As the experiment progresses, some overlooked details in the experimental setup are recognized and corresponding components are added or updated. Another reason is the failure of some legacy components that cannot be replaced by exact duplicates due to their outdated technology. For example, some components were developed 20 years ago, and consequently, they impose compatibility and security challenges.

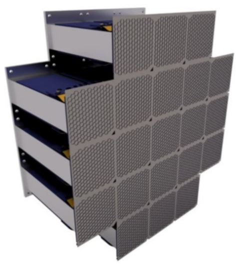
Low Manpower KATRIN like many large-scale experiments has a limited number of scientists and engineers that are responsible for its operation and development.

High Data Rate Detectors KATRIN's flexible and modular design allows reusing the infrastructure with the necessary upgrades. An example is the detector upgrade of KATRIN. The KATRIN experiment started with the focal plane detector (FPD) but as of the time of writing of this thesis, KATRIN is preparing to install a new detector and DAQ system as part of the new Tritium Sterile Anti-Neutrino (TRISTAN) upgrade [73] that utilizes the same tritium source to investigate the existence of sterile neutrinos empirically. Its data rates are expected to reach hundreds of gigabytes per second and has high memory requirements reaching up to 1 TB.

6.2 The TRISTAN Experiment

The TRISTAN upgrade wants to explore the hypothetical existence of the so-called sterile neutrinos experimentally which are considered to have a potential contribution to dark matter. The upgrade will replace the current focal plane detector with a new high-rate detector exploiting the tritium source of the KATRIN experiment [73].

6.2.1 Structure and Operational Modes



(a) Full 21-Modules Detector

(b) A TRISTAN Detector Module

Figure 6.2: The TRISTAN Experiment's Detector and Its Modules. Source: Max Planck Institute for Physics [80].

The TRISTAN detector is formed of independent modules called tiles. Each tile has 166 Silicon Drift Detector (SDD) hexagonal cells (also called pixels or channels). The detector will be deployed at the KATRIN infrastructure in 2 phases: the first phase employs 9 tiles (1494 pixels in total), the second employs 21 tiles (3486 pixels in total). Fig. 6.2a shows the focal plane of a 21-tile TRISTAN detector and Fig. 6.2b shows a 3D model of one detector tile with the 166 hexagonal shown at top.

The tiles will collaboratively work in 4 modes: Waveform, List-wave, List-mode, and Histogram. The Waveform Mode is a high throughput mode where the detector issues a raw and unfiltered data stream of the detector signal for a certain duration of all available tiles. The waveform will be sampled at $62.5\,\mathrm{MHz}$ with a sample size of $16\,\mathrm{bit}$ and producing $125\,\mathrm{Mbit/s}$ per channel. The rate of the detector in this mode is expected to reach $200\,\mathrm{Gbit/s}$ which should be collected *without loss* for at least $100\,\mathrm{ms}$ and requires at least $2.5\,\mathrm{GB}$ of memory. Longer runs can reach up to $1\,\mathrm{TB}$ of memory. The List-Wave is similar to the Waveform Mode includes raw data, but it is filtered data associated with timing and energy determination information. The filtering consists of offset removal, exponential deconvolution, timing filter, energy filter, and event building. Similarly, the Histogram and the List-Mode modes use the same filtering mechanisms but does not include raw data. The difference between the Histogram and List-Mode modes is the size of data in an event. All modes except Histogram Mode are used for diagnostics and calibration purposes.

The Histogram Mode is a low latency mode where a single detector tile is expected to trigger 16.6 million energy events in a second that should be collected, processed, and summarized in 4 to 10 histograms per detector channel before being saved to disk. Each histogram has 32768 bins of $4\,\mathrm{B}$ integers. Calculating the total required memory to build the histogram in the memory would result in: $21\,Tiles \times 166\,Channels \times 4\,Histograms \times 32768\,Bins \times 4\,Bytes\ = 1.7\,\mathrm{GB}$ at minimum, and similarly $4.25\,\mathrm{GB}$ at maximum.

6.2.2 The Hybrid Data Acquisition System

The high data rates of the TRISTAN detector impose high memory requirements that cannot be met without large segments of main memory, e.g. to receive the high throughput UDP stream of the waveform mode or to build the energy histogram in the histogram mode. Therefore, we propose a hybrid hardware-software DAQ system for the TRISTAN detector in collaboration with the scientists and the engineers at the TRISTAN experiment. Event building and filtering in this DAQ system is performed on detector electronics, but building histograms and saving detector data to storage disks for all modes are performed in a software subsystem running on a computer system.

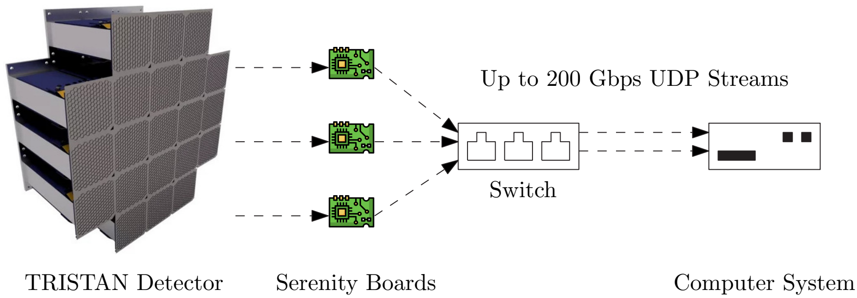


Figure 6.3: TRISTAN Proposed Architecture & Setup in Phase 1

Figure 6.3 shows the proposed setup of the TRISTAN DAQ system for 9 tiles in phase 1. The 9 tiles will be managed by Serenity boards that acquire data from the detector channels and send them to a computer system where software-based DAQ is performed. Each Serenity board is connected to 3 tiles and is responsible for the event building and filtering of 498 channels. Detector data from all Serenity boards is aggregated in 2x 100 Gbit/s uplink *Ethernet* links to the computer system using an Ethernet switch. To prevent the need of buffering on the Serenity boards and consequently the need of large memory buffers, the no-buffering lightweight User Datagram Protocol (UDP) will be used.

The computer system is an enterprise-grade computer with enough main memory capacity to handle the maximum requirements of the TRISTAN detector. Any failure in handling the detector data rates and the memory requirements to buffer them would result in loss of scientific data.

As discussed earlier, this hybrid DAQ architecture imposes several additional operational efforts on KATRIN infrastructure. The current hybrid DAQ architecture will need procuring a new computer system which needs additional proper system maintenance and administration. This also means that an engineer from the already low manpower KATRIN environment needs to dedicate part of his time to perform these tasks. Scaling the TRISTAN detector from 9 tiles in phase 1 to 21 tiles in phase 2 will increase the throughput demands and will likely needs to procure additional computer systems that would increase the operational efforts of the TRISTAN detector.

6.3 DFV for TRISTAN

The TRISTAN experiment adds another layer of complexity to the operational efforts in the KATRIN infrastructure due to the hybrid hardware-software DAQ system and due to the new detector's high data rates and memory requirements. DFV can simplify the operational efforts of the TRISTAN experiment by relying on campus computing facilities.

The TRISTAN experiment can exploit the automated flexible diversity-friendly environment of computer virtualization in DFV to run the software-based DAQ system in a campus computing facilities. Such a shift eliminates the need to procure new computer systems that are only used for the TRISTAN DAQ process and lowers the need for human intervention to maintain computer systems for DAQ by using the shared computer systems available in campus computing facilities. Instead of dedicating a whole computer cluster as in the traditional DAQ architecture, DFV shares the same computer cluster with other tasks (e.g. software DF from other experiments) while ensuring compatibility with other applications using computer virtualization and the DQDK framework. DFV also provide an easy method to scale TRISTAN DAQ from the throughput of 9 tiles in phase 1 to 21 tiles in phase 2. Instead of procuring additional computer systems to scale TRISTAN DAQ, DFV can exploit the readily available systems in campus computing facilities.

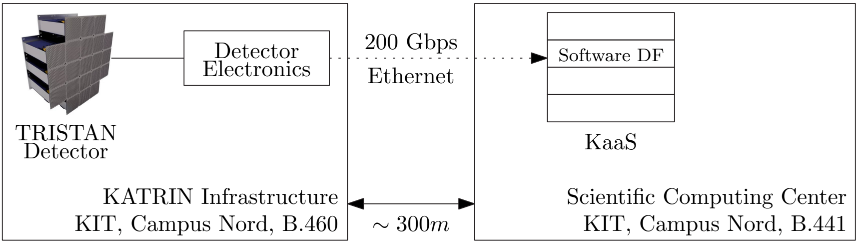


Figure 6.4: DFV Setup for TRISTAN

Scientists at KATRIN have already realized a computer virtualization solution for slow control and data management systems through the KATRIN as a Service (KaaS) private cloud [40]. We plan to exploit the KaaS platform to perform high throughput software-based DAQ for TRISTAN there. The KaaS platform is formed of 14 enterprise-grade computer nodes and is located at the Scientific Computing Center (SCC) in the north campus of KIT. The platform runs Kubernetes and containers to orchestrate and isolate the services of the slow control and data management systems on the 14-node cluster. Fig. 6.4 shows the proposed DAQ setup for TRISTAN using DFV. Detector electronics will send TRISTAN's detector data to the software-based DF running on the KaaS platform at SCC. We do not need major changes on the KaaS platform to enable DFV except for a $200\,\mathrm{Gbit/s}$ link that directly links the TRISTAN detector electronics at Building 460 in KIT Campus Nord and the KaaS platform at SCC Building 441 in KIT Campus Nord which are approximately $300\,\mathrm{m}$ apart. However and to the time of writing of this thesis, there exists no $100+\,\mathrm{Gbit/s}$ link between KaaS and the TRISTAN detector electronics, but it is only logistics and does not impact the DFV concept. Therefore, we deploy DFV for TRISTAN in an identical environment next to the TRISTAN detector for conceptual evaluation reasons.

6.4 Implementation

We adopt the DQDK framework for TRISTAN DAQ. At the time of the writing of this thesis only Waveform and Histogram modes are available to test in TRISTAN's detector electronics. Therefore, we implement the user logic for both modes in order to test TRISTAN DAQ.

The user logic in DQDK for the waveform is simple: all received data should be stored on permanent storage. The detector electronics send UDP packets with a $3392\,\mathrm{B}$ payload. The implementation should receive all packets with zero loss, process the UDP and IPv4 headers, and then store the content of the packets in an intermediary buffer before writing them to storage disks at the end of the session. The main challenge of this mode is managing memory accesses to the intermediary buffer: (1) allocating up to $1\,\mathrm{TB}$ of memory uses around 250 million $4\,\mathrm{kB}$-pages, which can increase latency due to TLB thrashing, (2) large memory allocations may push the OS to move some pages to the swap space to free up some physical memory, and finally (3) in case of concurrent access, unsynchronized access to the buffer may cause data corruption. In order to mitigate these challenges, we extend the DQDK memory allocator that already handles (1) and (2). The extension is a lock-free memory allocator that reserves memory from the pre-allocated intermediary buffer by incrementing an offset that points to the next free memory chunk using fetch-and-add atomic operations. Atomic operations, especially while using relaxed memory ordering, have a very lightweight overhead when compared to other synchronization primitives like *mutex* and *spin locks*.

For the Histogram Mode, the supplied user logic allocates the histogram structure in physical memory using the DQDK memory allocator. Similar to the Waveform Mode, the data electronics send UDP packets with a $3392\,\mathrm{B}$ payload containing 212 $16\,\mathrm{B}$-events representing energy values and other information e.g. timestamp, the corresponding histogram number, and the corresponding channel number. The implementation should receive all packets with zero loss, process the UDP and IPv4 headers, and then execute the histogram logic for each of the $16\,\mathrm{B}$-events in a packet's payload. The histogram logic identifies and increments the histogram bin using the channel number, the histogram number, and the energy value available in the event. Similarly, histogram increments are relaxed atomic operations to mitigate synchronization of parallel data accesses.

6.5 Evaluation

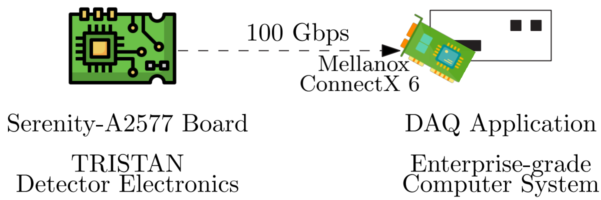


Figure 6.5: TRISTAN Simulation Test Setup

We test our software-based DAQ function for the TRISTAN detector by employing a simulation setup consisting of a Serenity-A2577 board equipped with a Xilinx VU9P FPGA & an enterprise-grade computer system with 16-core Intel(R) Xeon(R) Platinum 8444H CPU @ 2.9GHz, 256 GB main memory, and a 100 Gbit/s Mellanox ConnectX 6 NIC. Figure 6.5 shows the simulation setup. A Serenity-A2577 board implements a UDP client core including minimal IPv4 & ARP implementations over MAC & PMA/PCS Xilinx IP core. The UDP implementation is resource-efficient as it uses no block RAMs and only consume 1048 CLBs of the available 67200 CLBs (roughly 1.5%). The UDP server on the board generates UDP traffic with 3392 B of packet size simulating the TRISTAN detector. The UDP data is then transmitted over a 100 Gbit/s link to the enterprise-grade computer system.

On the DAQ computer system, we run the software-based DAQ function of TRISTAN for the Waveform and the Histogram modes. We provide 2 implementations for the software-based DAQ function: one that is based on DQDK as discussed in Section 6.4 and another that is based on state-of-the-art traditional OS networking

(POSIX Sockets). Since we plan to deploy the software-based DAQ function on the container-based KaaS platform in the future, we perform our evaluation using containers with host networking and bare metal.

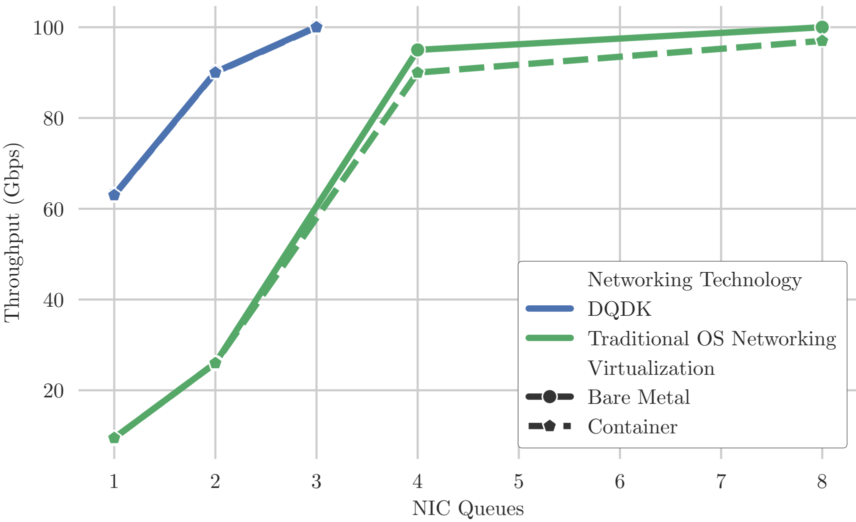


Figure 6.6: Zero-loss Throughput for TRISTAN Waveform Mode as function of NIC Queues

In the simulation of the Waveform Mode, Figure 6.6 shows the throughput (in Gbps) without loss using DQDK as function of NIC queues (each NIC queue requires 2 CPU cores for best performance). The plot estimates the needed CPU cores to process all the 100 Gbit/s traffic of the Waveform Mode with zero-loss. Using 1 NIC queue and our DQDK implementation, the maximum possible throughput with zero loss is only 63% of the 100 Gbit/s while it is only 9.5% for the traditional OS networking implementation. Increasing the throughput beyond 63% would push the NIC to drop detector data as there are not enough computing resources to process all received packets. Using 2 NIC queues, the throughput of the DQDK implementation is increased to 90% of the 100 Gbit/s link and only 26% for the traditional OS networking implementation. It is enough to employ 3 NIC queues (6 CPU cores) to saturate the link using the DQDK implementation, but we need up to 8 NIC (16 CPU cores) to do the same if we used traditional

OS networking. Based on our results in Chapters 4 and 5, we expect our DQDK implementation to scale linearly in order to process the 200 Gbit/s throughput of 9 tiles in Phase 1 requiring 12 CPU cores. We notice that virtualization does not impact our DQDK implementation (2 coinciding blue plots) which also confirms the results of Chapter 5. On the other hand, we see that using virtualization for traditional OS networking impact their performance: even using the maximum number of CPU cores (16 CPU cores) at 8 NIC queues cannot process all the NIC bandwidth achieving only 97%.

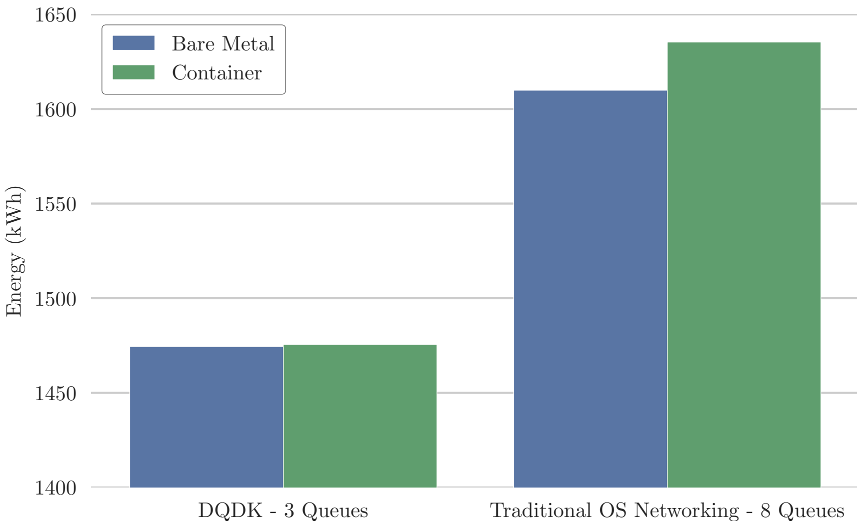


Figure 6.7: Consumed Energy for Zero-loss 100 Gbit/s Setups

We also compare the consumed energy by both setups that can process all the 100 Gbit/s link: DQDK implementation with 3 NIC queues and traditional OS networking implementation with 8 queues. We notice that our DQDK implementation consumes 1474 kWh and the traditional OS networking implementation consumes 1609 kWh. Using containers will consume slightly more energy in the traditional OS networking implementation (~26 kWh). The energy savings are up to 161 kWh (around 10% less energy).

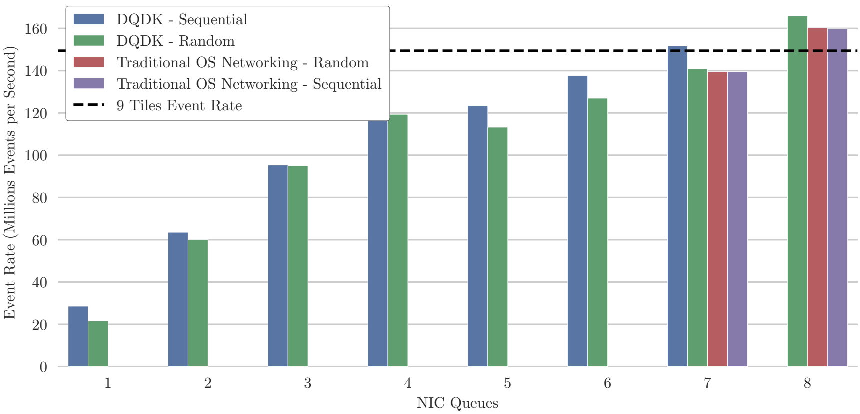


Figure 6.8: DQDK Histogram Event Processing Rates as function of NIC Queues

In the Histogram Mode and to simulate a trigger system on the Serenity board, we implement a simulator for TRISTAN Tiles (166 channels generating 16.6 million events per second for each tile) where each channel generates random numbers representing energy values. The goal is to calculate what are the needed resources to process the energy events of 9 Tiles in Phase 1 of the TRISTAN deployment. We run the software-based DF inside a container. Similarly to the Waveform Mode, the Histogram Mode is using 3392 B of 212 16 B-events each. The required processing rate for all 9 tiles would be 149.4 million events per second resulting in around 705000 3392 B-packet per second.

The histogram in the software-based DF is allocated as a multidimensional array. Incrementing the histogram bins based on the received events requires memory accesses to this array and yielding cache misses in some cases. Cache misses will increase the dispatching latency and the processing latency, and consequently harm the DF performance to process more events per second. Since the multiple histogram bins (2 B per bin) can fit in one cache line (64 B), the memory access pattern might affect the number of cache misses. For example, if the detector electronics are sending randomly ordered events, every bin increment may yield a cache miss. On the contrary, if the detector electronics are sending sequentially ordered events by the channel number, then some bin increments will land in the same cache line limiting the number of cache misses. For this purpose, we perform our evaluations for 2 memory access patterns: *sequential* access where the events in the UDP packet are ordered by the channel number, and *random* access where is no specific order for the events.

Figure 6.8 shows the event processing rate of both the DQDK and the Traditional OS Networking implementations as function of NIC queues. For sequential memory access pattern, it is enough to employ 7 NIC queues (CPU cores) to process the events of 9 TRISTAN tiles. However, we would need 8 NIC queues (16 CPU cores) to perform the same task. For random memory access, 8 NIC queues are needed for all setups. However, the DQDK implementation can process up to 165.88 million events per second which approximately corresponds to 10 TRISTAN tiles, while the Traditional OS Networking implementation can process up to 160.19 million events per second for random memory accesses and up to 159.77 million events per second for sequential memory accesses.

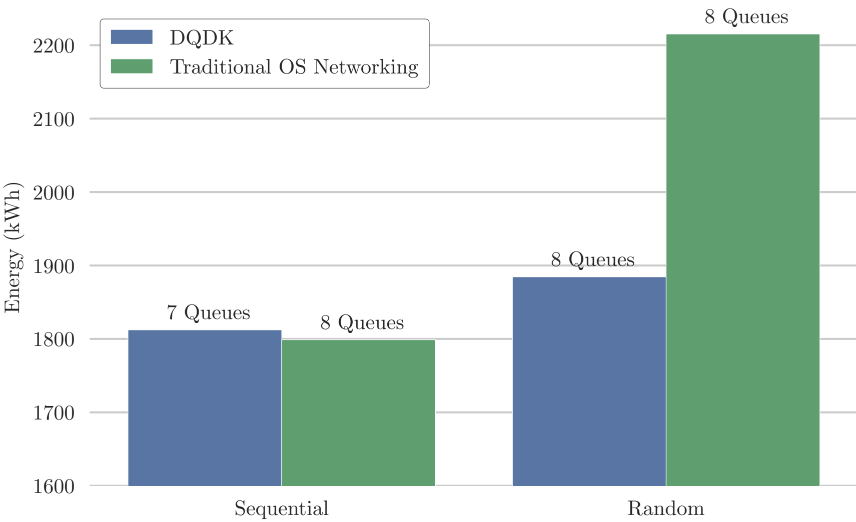


Figure 6.9: Consumed Energy for 9-Tiles Histogram Construction

Fig. 6.9 shows the consumed energy for configurations that can process all events of the 9 tiles or more. For sequential memory access, we record the consumed energy of both: DQDK with the required 7 NIC queues and Traditional OS Networking with 8 queues. Both configurations consume relatively the same amount of energy, but with our DQDK configuration is considered better because it can do the same with less CPU cores. For random memory access, 8 NIC queues are required to process the events of 9 tiles both in DQDK and Traditional OS Networking and thus both configurations require 16 CPU cores. However, our DQDK implementation consume $330.7\,\mathrm{kWh}$ less ($1884.55\,\mathrm{kWh}$ in total consumption) than Traditional OS Networking ($2215.25\,\mathrm{kWh}$ in total consumption).

These results show that our DQDK implementation is superior to Traditional OS Networking in terms of performance and energy. The largest performance advantage of DQDK in TRISTAN appears when the processing latency of the software-based DF is relatively low, e.g. Waveform Mode. Even when the processing latency is higher and DQDK performance is close to Traditional OS Networking, e.g. in some cases in the Histogram Mode, the DQDK framework still saves up to 15% in energy.

6.6 Conclusion

We presented a use-case for DFV using the TRISTAN detector, a real detector to be deployed soon in KATRIN's infrastructure. Using the DQDK framework, scientists can run software-based data acquisition functions with lower computing resources and energy. Our experiments show that we only require 6 CPU cores to process a $100\,\mathrm{Gbit/s}$ stream of TRISTAN detector in Waveform Mode and up to 16 cores for building a histogram in the Histogram Mode. The energy reductions are 10% in minimum and up to 15%.

The performed experiments are executed in a virtualized environment using containers. This will allow the scientists to realize DFV and unleash its advantages for the TRISTAN detector. Using DFV, scientists can deploy software-based DF on campus computing facilities eliminating the efforts to run, manage, and maintain computer clusters on the TRISTAN experiment's site. Since the required hardware is not available yet, we plan to extend our evaluations to $200\,\mathrm{Gbit\,s^{-1}}$ in the future to cover the whole throughput of the 9 TRISTAN tiles in Waveform Mode.

7 A Scalable Monitoring System in Scientific Infrastructure

Alongside a detector in a scientific infrastructure, a monitoring system is employed to ensure the robust operation of its DAQ system. A monitoring system collects and manages process data from hundreds to thousands sensors, e.g. temperature or magnetic fields sensors. As the scientific infrastructure gets more complex or increase in size, the operational efforts to store and access process data increase in complexity.

In this chapter, we study the attributes and the operational challenges of detector monitoring systems. We propose a new data-driven system design that is scalable and that reduces the operational efforts to run and maintain a monitoring system. The new design is built on top of time-series databases as a storage and analytics system. We argue that time-series databases can lower operational efforts by storing more and accessing process data faster than the traditional systems.

Publication

Parts of this chapter are published in 2 papers in the proceedings of the 34th International Conference on Scientific and Statistical Database Management and in 18th International Conference on Accelerator and Large Experimental Physics Control Systems:

J. Mostafa, S. Wehbi, S. Chilingaryan, and A. Kopmann. 2022. SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things. In Proceedings of the 34th International Conference on Scientific and Statistical Database Management (SSDBM '22). Association for Computing Machinery, New York, NY, USA, Article 12, 1–11.

J. Mostafa, A. Beglarian, S. Chilingaryan, and A. Kopmann. "Interfacing EPICS and LabVIEW Using OPC UA for Slow Control Systems," in JACoW, vol. ICALEPCS2021, pp. TUPV011, 2022.

7.1 Introduction

Thousands of sensors are deployed in a scientific infrastructure to measure different physical metrics related to the DAQ system or the environment of the scientific infrastructure e.g. electric current and voltage, magnetic field, temperature, etc. Collecting these measurements contributes to the scientific value of the scientific infrastructure, for example, to ensure the robust operation of the DAQ system or to complement the detector data to derive meaningful scientific conclusions. These measurements are called *process data*.

Process data are often time-series data that should be stored on permanent storage and gets integrated into the offline data analysis process of the scientific infrastructure on a later stage [81], [82]. We can summarize their data flow in 3 stages: (1) Transfer Stage: the slow control subsystem samples process data from the field-level devices, i.e. the sensors, according to a predefined sampling rate and then digitizes the sampled value in preparation for permanent storage; (2) Storage Phase: after sampling and digitizing process data, it is saved permanently on a computer storage volume that can handle the throughput requirements to write the process data measurements and to read them back; and finally (3) Offline Analysis Phase: process data is usually integrated in the offline data analysis process of a scientific infrastructure, storage volumes sometimes supports offline process data analysis through efficient indexing for fast retrieval or through some built-in basic data analysis functions like average and sum calculations.

As a scientific infrastructure gets larger or more complex, so does the operational efforts of the data flow of process data. The increasing number of sensors and their increasing sampling rates in a scientific infrastructure stresses their permanent storage in both of the Storage Phase to store the process data and the Offline Analysis Stage to retrieve and analyze it. For example, the KATRIN infrastructure has around $10{,}000$ sensors whose sampling rates range between $10\,\mathrm{Hz}$ and $0.1\,\mathrm{Hz}$ yielding around $16\,\mathrm{MB/s}$ [40]. With new upgrades like the TRISTAN detector among other upgrades, the number of sensors is expected to grow by an order of magnitude.

The storage volume is the central part in the data flow of process data to store and retrieve. It must be able to handle manage hundreds to thousands of gigabytes of process data while remaining available. However, an unreliable and unscalable storage volume can harm the availability of the data. Therefore, a reliable scalable storage volume is required to ensure the availability of process data. This chapter follows a benchmark-guided approach to design a reliable and scalable storage volume for process data.

7.2 Process Data in Databases

Database management systems, databases or DBMSs in short, are storage systems provide efficient data management and integrity by indexing the stored data and providing an easy data retrieval interface called the query language. For these reasons, scientists have adopted databases to manage time-series process data in several scientific infrastructure and experiments. In this section, we review two classes of DBMS: ACID and Time-Series Databases, and we study the suitability of both classes to manage process data.

7.2.1 ACID DBMS

ACID DBMSs are databases that implement the ACID principles which ensure the integrity and reliability of data management and prevent potential failures. The smallest execution unit in an ACID-based DBMS is a transaction with the following properties:

- **A**tomicity: a transaction is an indivisible unit of execution, if it fails while execution, all of its changes are rolled back restoring the data state exactly as it was before the transaction;
- **C**onsistency: a transaction cannot corrupt data, either it succeeds with a consistent state or it rolls back to the last state before executing the transaction;

- **I**solation: a transaction is independent of other transactions preventing interference among them;
- **D**urability: all changes made by a transaction are permanent and survive any subsequent failure.

The robust implementation of ACID DBMS made them popular as general-purpose data stores. Examples of such use-cases are financial transactions, E-commerce, etc. ACID databases are also used extensively to store process data in scientific infrastructure. For example, the KATRIN infrastructure uses MySQL and Microsoft SQL Server, two ACID DBMS to manage KATRIN's process data [40]. Another example is the CMS experiment that uses Oracle Database to store process data [83]. However, in big data scenarios like process data where data keep growing inside the database limitlessly, the ACID principles become a performance and a scalability bottleneck favoring consistency over availability and performance [84]. Thus, we are looking to replace ACID databases in scientific infrastructure with other DBMS technologies that can handle enormous amounts of process data.

7.2.2 Time-Series Databases

The evolution of time-series applications like process data as big data applications allowed the emergence of Time-Series Databases (TSDB). TSDBs are motivated by the special characteristics of time-series data in comparison to other types of big data. Time-series data are: (1) indexed by its corresponding timestamps; (2) continuously expanding in size; (3) usually aggregated, down-sampled, and queried in ranges; (4) and has very write-intensive requirements. Different TS-DBs developed distinct technologies to exploit these characteristics by designing storage engines that are capable of handling the increasing data volumes and by accommodating indexing algorithms that provide low data retrieval latency.

We plan to design a novel scalable monitoring system in scientific infrastructure based on time-series databases. However, to evaluate which of the time-series databases' technologies are suitable for scientific infrastructure, we require a benchmark that is specifically designed for this specific use-case. The next sections will discuss SciTS, a benchmark for time-series databases in scientific infrastructure whose design is inspired by the use-case of scientific infrastructure [3]. We use SciTS to evaluate different time-series databases, and then we discuss which are suitable for our use-case i.e. fast data storage and retrieval. We dedicate a section to explain how our solution can be integrated with all diverse sensors in scientific infrastructure.

7.3 Related Work

Understanding the performance of databases has been a topic of interest for so long. Performance evaluation of databases helps in capacity planning and in choosing the most suitable database for a specific use case like time-series data workloads, big data workloads, or transaction-based workloads. The most notable benchmarks are the benchmarks from the TPC council for Online Transaction Processing (OLTP) databases e.g. TPC-C [85], TPC-DS [86], and TPC-H [86]. The scientific community also introduced other benchmarks like [87] for OLTP databases or YCSB [88] for big data databases.

TPCx-IoT is the Internet of Things benchmark from the TPC council. Its workloads simulate data from energy power plants in the form of data ingestion and concurrent queries. TPCx-IoT supports very basic queries which makes it unsuitable for many practical uses. TSBS [89] is a benchmark from the developers of the TimescaleDB company. TSBS simulates a load of IoT devices in addition to DevOps, but TSBS lacks concurrency and the ability to read the usage of system resources. Rui Lui et al. propose the IoTDB-Benchmark [90] for IoT scenarios. IoTDB-Benchmark supports concurrent, aggregation, and down-sampling queries. YCSB-TS [91] adopts the structure and the workloads of YCSB and adds basic time functions and thus inherits unoptimized workloads to benchmark time-series databases. *ts-benchmark* [92] is a time-series benchmark developed by Yuanzhe Hao et al. It uses a generative adversarial network (GAN) model to generate synthetic time-series data to ingest data and supports diverse workloads for data loading, injection, and loading in addition to monitoring usage of system resources. *ts-benchmark*, however, does not take into consideration aggregation and down-sampling queries which are very important for data visualization and analysis.

7.4 SciTS: Benchmarking Time-Series Databases in Scientific Infrastructure

This section provides an overview of the architecture of SciTS and its design that supports the requirements discussed in Section 7.1. SciTS is an extensible configurable client-side benchmark that can work for any single node DBMS. Fig. 7.1 shows the architecture and the control flow of SciTS. The benchmark flow starts the configurator that reads the user's configurations and parameters from the workload definition file to create and launch a parallelized benchmark scenario. The configurator then creates the requested parallel clients. Each client operates a workload manager to create and submit workloads to the target database server. For ingestion workloads, the workload manager submits a request to the data generator abstraction layer to create sensor data. The generated sensor data is then passed to the database abstraction layer, an abstract interface that wraps the implementations of database clients. On the other hand, the parameters of query workloads are submitted directly to the database abstraction layer for execution. While executing the workloads, SciTS asynchronously monitors the usage of the system resources on the target database server. The collected workload performance metrics and the system resources metrics are then recorded and persisted in separate files.

SciTS is extensible through its abstract interfaces and resilient configurations. It abstracts database access, workloads, and data generations that are easy to extend for additional benchmark scenarios. For instance, SciTS uses a random data generator by default, but additional data generators can be added by providing other implementations of the data generation abstraction interface. Similarly,

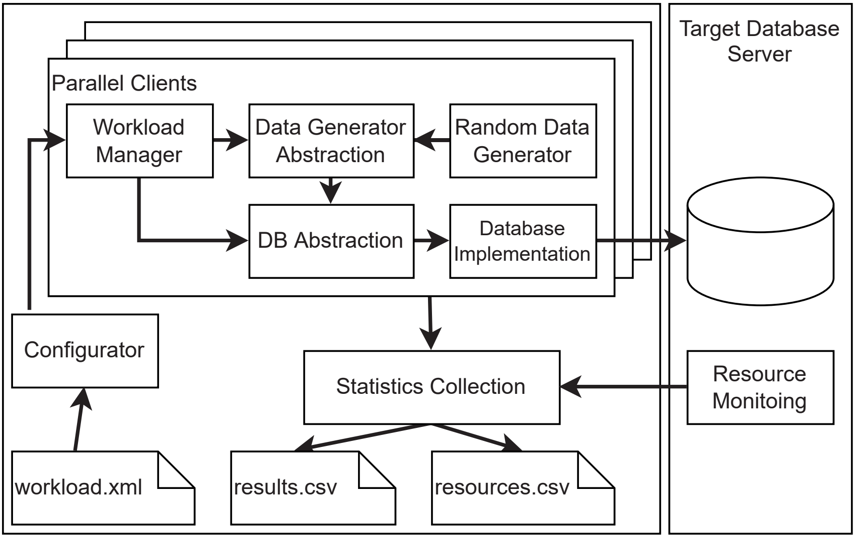


Figure 7.1: The Architecture and Process Flow of SciTS

additional queries and new database servers can be added by extending the relevant interfaces. Data ingestion workloads are extensible via benchmark specifications described in the workload definition file e.g. a concurrency workload in SciTS is a data ingestion workload that varies the number of clients in its definition file and fix the batch size.

7.4.1 Design Considerations

The benchmark simulates heavy INSERT workloads as well as 5 queries inspired by the use case of time-series data in scientific infrastructure in the KATRIN infrastructure [40]. SciTS can simulate any workload by parameterizing concurrency, cardinality, and size of batches while considering the best performance practices for each workload type. Unlike other benchmarks, it introduces a new workload "Scaling Workload" to understand the performance of TSDBs as the data inside the database grows larger. In addition, SciTS also collects usage of the system resources like CPU and memory usage.

As mentioned earlier, our benchmark is the product of hours of research in testing and evaluating TSDBs for scientific experiments. Based on our experiences, we gathered a list of requirements for a good TSDB benchmark:

- **Customizability & Extensibility:** an easy and highly extensible interface to generate and test different types of INSERT workloads;
- **Practical Queries:** queries from practical and real-life environments e.g. range queries, out-of-range queries, and more complex queries like aggregation and down-sampling queries;
- **Scalability Performance:** the ability to understand the performance of a TSDB as its size grows larger;
- **System Monitoring:** the capability of monitoring the usage of system resources.

Existing TSDB benchmarks only support a limited set of queries or do not reflect on the scalability performance of a TSDB [89]–[92]. Our benchmark builds on previous efforts by providing queries from the scientific infrastructure, and by giving insights into the scalability performance of TSDBs.

7.4.2 Benchmark Workloads

We propose eight types of benchmark workloads (3 data ingestion workloads and 5 query workloads) for time-series databases.

7.4.2.1 Data Ingestion Workloads

Ingestion workloads simulate writing process data to the TSDB. Extensions to SciTS ingestion workloads is possible by changing three relevant parameters:

concurrency i.e. number of clients, size of data batches, and cardinality i.e. number of sensors. Using these parameters, the benchmark user can create any workload scenario. For our study, we introduce 3 data ingestion workloads focusing on batching, concurrency, and scaling.

Batching Workload Understanding the performance of databases under different batch sizes helps in evaluating how they react to small and big batches. This evaluation is important to decide how to use the database e.g. what is the most suitable batch size for a specific database? Or how far can we go in the size of inserted data batches? For this workload, we consider varying the batch size while using only one database client. We consider batch sizes as small as 1000 points per batch and as large as 100000 points per batch. We study the latency brought in by inserting data of different batch sizes.

Concurrency Workload Any practical use of databases in a scientific infrastructure includes using numerous clients that are responsible for reading sensor data from multiple devices and writing the results into the database in batches. The concurrency workload tests the performance of TSDBs by varying the number of clients and monitoring the ingestion rate of the database as well as the usage of system resources.

Scaling Workload Different databases have different backends that use memory and storage resources in distinct ways. While some databases may support higher ingestion rates than others, it is important to study the performance of the database as data grows larger. The goal of this workload is to study the performance of TSDBs as they grow in size over time. It involves collecting and studying the usage of system resources to understand the impact of system resources on data ingestion.

7.4.2.2 Queries Workload

SciTS proposes five queries that are inspired by the KATRIN's data visualization and offline analysis workloads. The queries focus on returning raw, aggregated, or down-sampled data of one or more sensors. We define data aggregation as summarizing a duration of time using one single value e.g. average, standard deviation, etc. On the other hand, we define down-sampling as the practice of summarizing the sensor's data on a time interval basis using a statistical function to create a new time-series of summarized intervals.

Assuming the data is defined using the constructs of a traditional DBMS table, the table schema would be *(time_field, sensor_id, value_field)*. We also assume that the function *TRUNCATE* is a function that returns a list of time intervals of a specified length e.g. *TRUNCATE('1min', time_field)* will return a list of time-intervals where each item represents a 1-minute of data using the column *time_field*. Using this schema, the queries and their SQL equivalents can be described as follows:

(Q1) Raw Data Fetching: Get the raw values of one or more sensors over a duration of time. It is used to visualize and analyze data of specific sensors.

```sql
SELECT *
FROM sensors_table
WHERE time_field > ?
AND time_field < ?
AND sensor_id = ANY(?, ?, ?, ...)
```

Listing 7.1: Query 1: Raw Data Fetching

(Q2) Out of Range Query: Get the intervals over a duration of time when the value of a specific sensor was out of a defined range. It is used to detect when the sensor was acting abnormally in a specific interval of time.

```sql
SELECT TRUNCATE(period, time_field)
AS interval, MAX(value_field),
```

```
MIN(value_field)
FROM sensors_table
WHERE time_field >= ?
AND time_field <= ?
AND sensor_id = ?
GROUP BY interval
HAVING MIN(value_field) < ?
OR MAX(value_field) > ?
```

Listing 7.2: Query 1: Raw Data Fetching

(Q3) Data Aggregation: Represent the data of one or more sensors over a specific duration of time using one aggregated value of an aggregation function denoted by *agg_func* e.g. the standard deviation, the mean, etc.

```
SELECT agg_func(value_field)
FROM sensors_table
WHERE time_field >= ?
AND time_field <= ?
AND sensor_id = ANY(?, ?, ?, ...)
```

Listing 7.3: Query 3: Data Aggregation

(Q4) Data Down-Sampling: down-sample one or more sensors using a specific sampling function denoted by *agg_func* over a duration of time.

```
SELECT TRUNCATE(period, time_field)
AS interval, sensor_id,
agg_func(value_field)
FROM sensors_table
WHERE time_field >= ?
AND time_field <= ?
AND sensor_id = ANY(?, ?, ?, ...)
GROUP BY interval, sensor_id
```

Listing 7.4: Query 4: Data Down-Sampling

(Q5) Operations on Two Down-sampled Sensors: Down-sample the data of two sensors over a duration of time and using the function *agg_func*, then compare the results using the function *comp_func*. A use case of this query is comparing the data of two down-sampled sensors using value subtraction.

```sql
SELECT Sensor1.period,
comp_func(Sensor1.val, Sensor2.val)
FROM
  (SELECT TRUNCATE(period, time_field)
   AS interval,
   agg_func(value_field) AS val
   FROM sensors_table
   WHERE time_field >= ?
   AND time_field <= ?
   AND sensor_id = ANY(?, ?, ?, ...)
   GROUP BY interval)Sensor1
INNER JOIN
  (SELECT TRUNCATE(period, time_field)
   AS interval,
   agg_func(value_field) AS val
   FROM sensors_table
   WHERE time_field >= ?
   AND time_field <= ?
   AND sensor_id = ANY(?, ?, ?, ...)
   GROUP BY interval)Sensor2
ON Sensor1.period = Sensor2.period
```

Listing 7.5: Query 5: Operations on Two Down-sampled Sensors

7.4.3 Workload Definitions

A SciTS workload is a set of parameters in its XML configuration file in addition to information about the target database server and its connection specifications.

Date and time span can be described in a workload definition to describe how sensors' timestamps are distributed over a specific period.

Table 7.1 shows the user-defined parameters of SciTS. In addition to generic parameters like *TargetDatabase*, *DaySpan*, and *StartTime*, SciTS defines parameters for each workload type. An ingestion workload is defined by parameterizing SciTS using: (1) *ClientNumberOptions* to represent concurrency i.e. the number of database clients to insert records into the database, (2) *BatchSizeOptions* to configure the batch size to insert in one operation, (3) and *SensorNumber* to parameterize the cardinality of the database table by configuring a specific number of sensors. For instance, a concurrency workload is defined by setting the *ClientNumberOptions* to a set of number of clients to test with e.g. setting it to 1,2,4 means run the same workload with one database client, then two clients, then four clients in one atomic run without changing the configuration. The batching workload is another example where the user can similarly set *BatchSizeOptions* to a set of batch sizes to test the database server with in one atomic run.

On the other hand, the user can specify in the configuration file what query he needs to execute using the *QueryType* option. The five queries can be parameterized by choosing the queried time intervals (*DurationMinutes* in Fig. 7.1) for, and by filtering on one or more sensors using the *SensorsFilter* parameter. Down-sampling and aggregation queries are additionally parameterized by specifying aggregation or sampling interval. The benchmark uses the *average* function to calculate aggregations. Other queries like out-of-range queries that require filtering on the *value* column can be parameterized in the configuration file using the *MinValue* and *MaxValue* fields. To assess the results' correctness, the user can repeat the same query with the same parameters as much as needed using the *TestRetries* parameter.

7.4.4 Performance Metrics

We evaluate the performance of data ingestion workloads by monitoring the latency taken to insert batches to the target database. We also consider the

Name	Description	Workload Type
TargetDatabase	The type of the target database server e.g. InfluxDB, ClickHouse, etc	Ingestion/Query
DaySpan	Length of the whole time-series in the database table in days	Ingestion/Query
StartTime	Earliest timestamp to be stored into or retrieved from the database	Ingestion/Query
BatchSizeOptions	Size of batch to insert into table	Ingestion
ClientNumberOptions	Number of concurrent clients	Ingestion
SensorNumber	Number of sensors to simulate to represent cardinality	Ingestion
QueryType	An enum representing the query type e.g. Q1-Q5	Query
TestRetries	How many times to repeat the query test	Query
DurationMinutes	Length of time-series data in minutes	Query (Q1 to Q5)
AggregationIntervalHour	Length of time window to apply the down-sampling function on	Query (Q3 to Q5)
SensorsFilter	A list of sensor IDs to filter on in the query	Query (Q1 to Q5)
MaxValue	The upper boundary of the sensor's value used in Q2	Query
MinValue	The lower boundary of the sensor's value used in Q2	Query

Table 7.1: User-defined Parameters of SciTS Workloads

ingestion rate of the database (the sum of all inserted data points divided by the time it has taken to finish the insertion transaction). In scaling workloads, we consider a rolling ingestion rate where we resample the data on a one-minute interval basis then we calculate the ingestion rate for each of these intervals.

To evaluate query workloads, we consider the latency taken to execute and return the query results. We use the *TestRetries* parameter to repeat the queries 1000 times then we study the samples' minimum, maximum, average, standard deviation, and 95% percentile.

The benchmark monitors the usage of system resources of the server by using Glances [93]. In general, SciTS monitors CPU (I/O wait, system, user, context switches), physical memory (used and cached memory), swap usage, disk I/O (read/write in bytes per second, count of I/O operations), and network usage (sent and received).

7.4.5 The Implementation

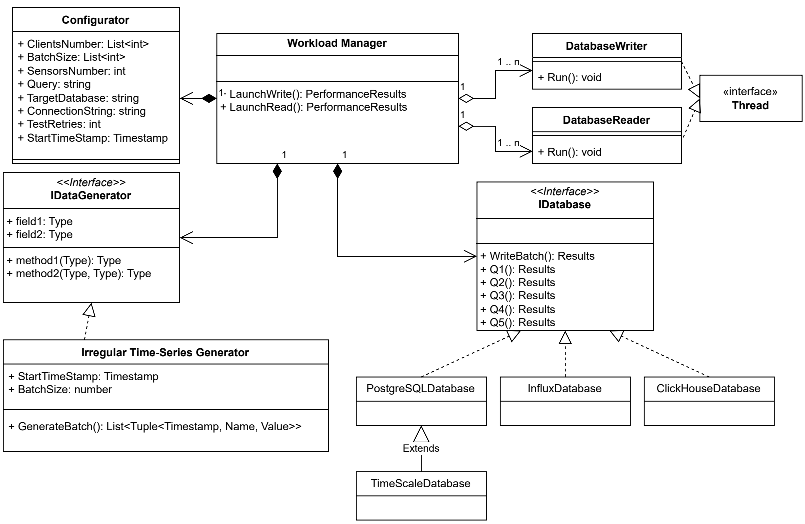


Figure 7.2: A UML Diagram of SciTS implementation

SciTS is implemented using C# and is highly extensible using its abstraction layers and resilient configuration. The benchmark can support any DBMS as long as it is implemented using the database abstraction layer. The current implementations include *ClickHouse* [94], *InfluxDB* [95], *PostgreSQL* [96], and *TimescaleDB* [97]. We try to adopt best practices for each implementation to achieve the best possible performance, for instance: *PostgreSQL* and *TimescaleDB* bulk inserts are powered by PostgreSQL SQL *COPY* statement that is optimized for high-performance ingestion rates with less locking and fewer indexing updates.

SciTS implements a random data generator for data ingestion. The data generator generates timestamps incrementally based on the date and periods defined in the workload definition file. The granularity of the timestamps is configured also configurable in the workload definition file. Sensors' values are considered to be random values that are uniformly ranging between zero and the max value of a signed 32-bits integer.

7.5 Performance Evaluation

We use SciTS to evaluate the targeted TSDBs and compare their performance to ACID databases.

7.5.1 Experimental Setup

We use two machines for our benchmarks: *M1* & *M2* which work as a server and a client to perform our tests. Machine *M1* is an enterprise-grade storage server that we use to host the database servers. It is equipped with an Intel Xeon CPU E5-1620 v2 @ $3.70\,\mathrm{GHz}$ of 8 logical cores, $32\,\mathrm{GB}$ DDR3 RAM, and 24 physical drives formatted with XFS and configured with RAID60 and connected using a $6\,\mathrm{Gbit/sec}$ SAS-2 controller. Machine *M2* acts as the client machine and is equipped with Intel Xeon CPU E5-2680 v3 @ $2.50\,\mathrm{GHz}$ over 2 sockets of 48 logical cores in total, and $512\,\mathrm{GB}$ DDR4 RAM in total. Both machines are connected over a $1\,\mathrm{Gbit/sec}$ Ethernet switch. We monitor the network traffic of both servers to make sure the connection is not saturated.

For our tests, we consider the following table schema for all databases where we store all measurements and data points in one table: *(timestamp, sensor_id, value)*. A record in this schema is represented by an 8 bytes timestamp of when the data point of the sensor was taken, an 8 bytes long integer as the ID of the corresponding sensor, and 8 bytes double-precision float. In all databases, we add indexes (a combined index) on the *timestamp* field and *sensor_id*, so it is faster to query for data points for a specific duration and specific sensors.

We use machine M1 for all the database servers. Each server runs independently of the others while the others are all down. For all database servers, we use only one node. Evaluating the performance of a cluster of database server nodes is out of the scope of this paper.

To evaluate our benchmark, we choose 3 TSDBs of three distinct storage engines: *InfluxDB* [95] to represent TSDBs based on LSM trees, *TimescaleDB* [97] to represent TSDBs based on traditional ACID-based DBMSs, and *ClickHouse* [94] to represent column-oriented OLAP-based TSDBs. We compare the performance of the three chosen TSDBs to *PostgreSQL* as a reference line for traditional ACID databases. We optimize theirs configurations to allow the best possible performance as follows:

ClickHouse It is a column-oriented DBMS designed for high ingestion rates. ClickHouse's storage engine is called *MergeTree* that writes the data directly to the table part by part to offer high-speed unrestricted data insertion. A background job then merges the parts. Data in ClickHouse can be stored sorted on the disk which allows using sparse indexing to locate data in the partitions quickly. We configure the database server to partition data every day. Each partition is then ordered by the table's primary key the tuple *(timestamp, sensor_id)*. Indices are defined on both of the fields: *timestamp*, and *sensor_id*. We use

ClickHouse v22.1.3.7 with its native TCP protocol, and we set the following configurations: *max_server_memory_usage_to_ram_ratio* to 0.9, *index_granularity* is 8192 rows, and *async_insert* is off.

InfluxDB It is a TSDB that uses the *Time-Structured Merge Tree* (TSM Tree), a storage engine that resembles Log-Structured Merge (LSM) trees [98] in its design. Inserted data in TSM trees is written to a Write-Ahead Log (WAL) at first and copied to the cache while maintaining indexes in memory. The data is then persisted on the storage using immutable shards, each shard contains the data of a corresponding duration of time. An InfluxDB record consists of a timestamp, a value, and one or more tags. Tags are key-value pairs that are used to add data to the record. InfluxDB uses timestamps and tags for indexing. It uses per-type data compression algorithms e.g. ZigZag encoding for integers, the Gorilla algorithm [99] for float numbers, simple8b [100] for timestamp indexes, bit packing for booleans, and the snappy algorithm [101] for strings. We use InfluxDB v2.1.1 and the *Line* protocol to insert data. Its server is set up with the following configuration: *storage-wal-fsync-delay* is set to 0, *storage-cache-max-memory-size* is set to 1048 MB, and *storage-cache-snapshot-memory-size* is set to 100 MB.

PostgreSQL It is an ACID-based DBMS that uses WAL to insert data. The WAL ensures the reliability of the data written to the database. It protects the data from power loss, operating system failure, and unanticipated hardware failures. We set up a PostgreSQL table with the previously discussed schema on one PostgreSQL v13.5 server and use B-Tree indexes on the fields *timestamp* and *sensor_id* to find data quickly on a time range and for specific sensors. To optimize configurations for the host machine, the server is configured with *pgtune* [102] with the following configurations: *shared_buffers* is 7994 MB, *maintenance_work_mem* is 2047 MB, and *max_parallel_workers* is 8 workers.

TimescaleDB It is an extension of PostgreSQL. TimescaleDB benefits from the reliability and the robustness of PostgreSQL in addition to its SQL query engine. To solve the problem of always growing data, TimescaleDB uses hypertables that partition the data by the time column into several chunks. Each chunk is a standard PostgreSQL table. Standard SQL queries can be applied to the hypertable. This architecture handles time-series data better than traditional PostgreSQL. Indexing per chunk and chunks that can fit in the memory allows higher ingestion rates than traditional PostgreSQL. For low query latency, TimescaleDB uses age-based compression that transforms rows into a columnar format. Based on TimescaleDB recommendations, we set up a TimescaleDB v2.5.1 server with a *hypertable* of a 12-hours chunking interval, so chunks constitute no more than 25% of the main memory. TimescaleDB compression is configured to compress row data into the columnar format every 7 days of data and to order the columnar data by *timestamp* and *sensor_id*. The server is configured with the tool *timescale-tune*.

7.5.2 Results

This section discusses the results and the analysis we did after applying SciTS workloads to the target databases. For each of the experiments below, we consider the scientific infrastructure scenario with 100,000 sensors in total to provide a realistic case of cardinality in the database.

7.5.2.1 Data Ingestion

Batching Workload The goal of this workload is to understand how different database servers react to different batch sizes. We vary the batch size for each database then we measure the latency taken to insert each of these batches. For all databases and each of the batch sizes, we start from an empty database to keep the data of the experiments statistically independent as much as possible. We vary the batch size from 1000 records until we reach 100 000 records, the maximum number of records KATRIN control system can have in a second.

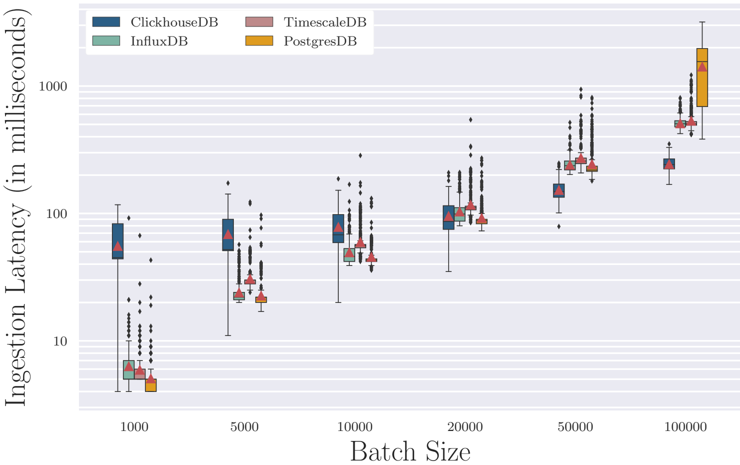


Figure 7.3: Batch Ingestion Latency as Function of Batch Size

Fig. 7.3 shows a box plot of the batch ingestion latencies and their mean values on a log scale as a function of the batch size for each of the target databases. Each box plot corresponds to the insertion of 500 batches into the target database. For batch sizes smaller than 10,000, the traditional ACID-based design of PostgreSQL performs better than time-series databases. Since ClickHouse's MergeTree writes directly to the storage, the latency produced by frequent write operations prevents ClickHouse from performing as well as other databases. For 20,000 data points in a batch, the four databases perform close to each other, and their means are in the range 95 ms–116 ms. For huge batch sizes like 50,000 and 100,000, ClickHouse outperforms all other databases.

TimescaleDB, InfluxDB, and PostgreSQL provide close performance for most of the batch sizes except in batch size 100,000 where PostgreSQL fails to handle very large data batches and the latency to insert one batch can reach more than 3000 ms while the chunks of TimescaleDB hypertables provide much better performance compared to traditional PostgreSQL.

Concurrency Workload The goal of this workload is to study the performance of the databases as the number of clients varies. For each of the target databases, we start from an empty table then we start varying the number of clients that are inserting data into the table. As we vary the number of clients, we calculate the total ingestion rate and check the CPU and the memory usages for each database. We choose a batch size of 20,000 since all targeted databases have a close ingestion latency as shown in Fig. 7.3.

Fig. 7.4 shows the ingestion rate as a function of clients for each of the target databases. ClickHouse achieves the best ingestion performance where the ingestion rate can hit 1.3 million data points per second on average while using 48 clients. While ClickHouse shows an increasing performance with the increasing number of concurrent clients, other databases show some performance limits: InfluxDB is saturated with 24 clients and cannot achieve more than 790,000 points per second; TimescaleDB and PostgreSQL reach their peak performance at 550,000 and 400,000 respectively.

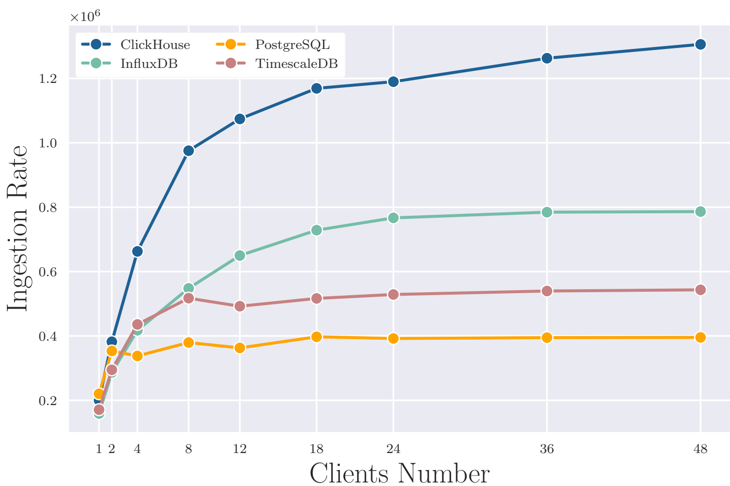
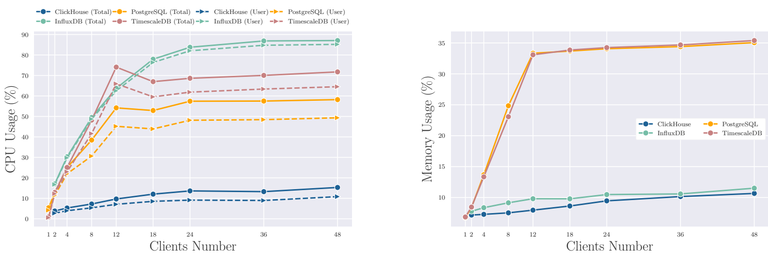


Figure 7.4: Ingestion Rate (in records per second) as Function of the Number of Concurrent Clients

Fig. 7.5 shows the usage of system resources as a function of varying the number of concurrent clients. Fig. 7.5a shows average total (solid lines) and user-space (dashed lines) CPU usage per clients number. Although InfluxDB provides a considerably high ingestion rate we notice that its average CPU usage is high even when the number of concurrent clients is below 8. For a higher number of clients, InfluxDB can overload the CPUs. This explains why InfluxDB reached its peak performance at 790,000 in Fig. 7.4. TimescaleDB and PostgreSQL also show high CPU usage, especially beyond 12 concurrent clients with a wider gap between total CPU usage and user-space usage. The high CPU usage in these two databases is expected as a result of the process forks that are created for each PostgreSQL connection. On the other hand and accompanied by much higher ingestion rates, ClickHouse maintains a considerably low CPU usage even with many concurrent clients.



(a) Average CPU Usage

(b) Average Memory Usage

Figure 7.5: Usage of System Resources as Function of the Number of concurrent clients

Fig. 7.5b shows the memory usage of the target databases. While InfluxDB and ClickHouse keep a low footprint where they do not exceed more than 15% as an upper limit even with high numbers of concurrent clients. TimescaleDB and PostgreSQL have the same memory usage footprint, and they reach up to 34% with only 12 concurrent clients.

Scaling Workload The goal of this workload is to stress and check the performance of the target database server as its size goes larger. We stress each of the target databases with 48 concurrent clients, the maximum number of logical cores the machine *M2* is capable of. Each client continuously inserts batches of 20,000 records where the ingestion latency is most similar for all databases (as shown in Fig. 7.3) and until we reach around 2.8 billion records in total. Table 7.2 shows the ingestion rate and the total time taken to insert around 2.8 billion records in each of the databases. ClickHouse shows the best ingestion performance with the ability to ingest more than 1.2 million records per second then InfluxDB, TimescaleDB, and finally PostgreSQL in order. Compared to PostgreSQL, ClickHouse provides 6x speedup in data ingestion where it writes directly to the storage without passing into leveled write procedures like LSM trees. On the other hand, InfluxDB provides 3.5x speedup in data ingestion using its LSM tree-based storage engine. Being based on PostgreSQL, TimescaleDB inherits some of its limitations and provides only 2.33x speedups in ingestion rate.

Target Database	Total Time	Ingested Records/second	MB/sec
ClickHouse	00h:37m:32s	1,278,928	~30.69
InfluxDB	01h:04m:43s	741,688.5	~17.8
TimescaleDB	01h:37m:55s	490,149.8	~11.76
PostgreSQL	03h:48m:10s	210,361.9	~5.04

Table 7.2: Total Time, Ingestion Rate, and the Throughput of the Scaling Workload Experiment

Fig. 7.6 shows that time-series databases not only perform much better than PostgreSQL but also provide stable performance with respect to the table size in the database. To understand why the performance of PostgreSQL is dropping, we look at its corresponding collected system metrics. Fig. 7.7 shows the system metrics of the scaling workload for the target database servers as a function of the duration of the experiments. We noticed that the percentage of CPU spent I/O Wait in Fig. 7.7a is very high for PostgreSQL sever reaching the maximum value around 50% and averaging around 14.79%. In addition, Fig. 7.7b shows the percentage of used memory of the target database servers. As the data in the database server grows larger, PostgreSQL and TimescaleDB memory usage keep increasing until they reach around 40% and the operating system starts swapping database pages to the storage disks as shown in Fig. 7.7c. On the other hand, InfluxDB and ClickHouse use up to 20% of the physical memory with a negligible swap usage. PostgreSQL's ingestion rate performance degradation is caused by swapping indexes in and out from the physical memory as the time-series data in the database grows larger. TimescaleDB solves these shortcomings of PostgreSQL by optimizing the usage of the physical memory through chunking the big table to partitions whose indexes are independent and can fit into the physical memory, thus it does not rely on the swap as PostgreSQL does.

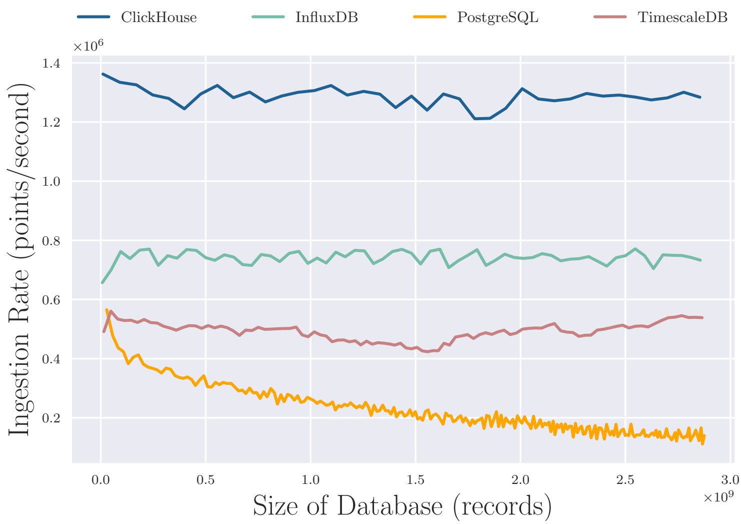
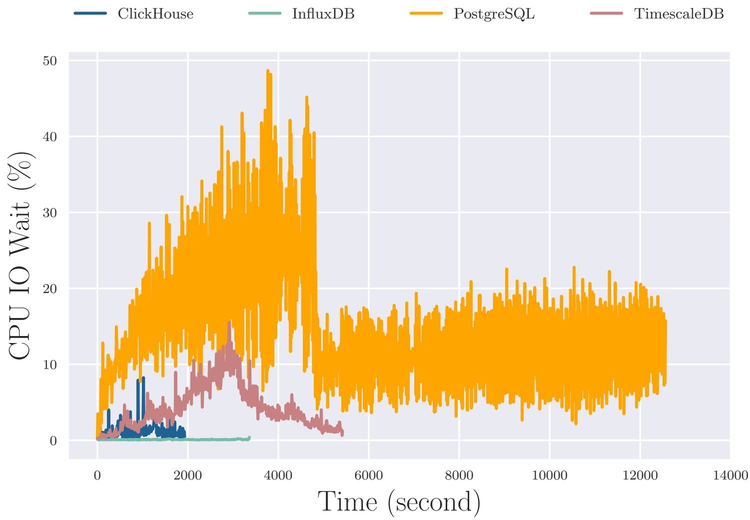


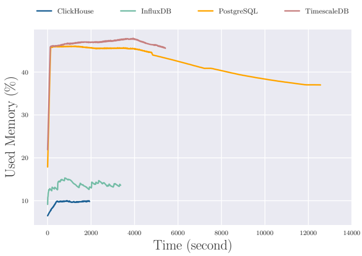
Figure 7.6: Ingestion Rate (in million records per second) as function of the size of the database.

7.5.2.2 Queries Latency

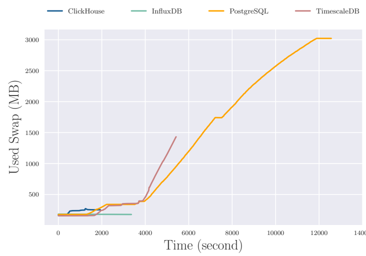
We discuss the performance of SciTS queries. We fill the database with 2.8 billion records that correspond to a duration of 15 days and for 100,000 sensors. For each query, we execute 20 runs. For each query run, we clear the database tables



(a) Percentage of CPU Usage of I/O Wait



(b) Percentage of Used Physical Memory



(c) Used Swap Space in Megabytes

Figure 7.7: The Scaling Workload System Metrics for Different Database Servers as Function of Time

and the operating system caches, and restart the database server to make sure the query results are directly returned from disk and choose distinct parameters.

Q1. Raw Data Fetching It queries the database to read the time-series data of a 10 min duration for 10 distinct sensors. Each 10 min interval is randomly selected from 15-day dataset using uniform distribution. The duration corresponds to around 5000 data points. Table 7.3 shows the query latency statistics in milliseconds for Q1. The query latency is the lowest on ClickHouse where it records 272 ms as a maximum value and 177.7 ms as an average value. PostgreSQL with its B-Tree indexing is second in performance with 457 ms as a maximum value and 361.7 ms on average. InfluxDB is third with 1172 ms as a maximum value and 1352 ms in average and greater deviation than that of ClickHouse and PostgreSQL. The disadvantages of chunking a table are realized when the TimescaleDB is forth with 1352 ms as a maximum value and 1284.55 ms in average and with the greatest deviation.

Database	Min.	Mean	95%	Max.	Std. Dev.
ClickHouse	131	177.7	241.6	272	32.64
InfluxDB	567	737.5	1058.95	1172	161.36
TimescaleDB	608	910.75	1284.55	1352	217.57
PostgreSQL	283	361.7	426.6	457	51.64

Table 7.3: Query Latency Statistics (in ms) for Q1

Q2. Out of Range We query the database for the day hours when the data of exactly one sensor is considered out of range according to user-defined boundaries in a duration of 180 min of time-series of data. The 180 min duration is randomly selected from the 15-day dataset using uniform distribution. Table 7.4 shows the query latency statistics in milliseconds for Q2. ClickHouse again achieves first place with a maximum value of 263 ms and an average value of 188.35 ms. TimescaleDB comes in second with a maximum value of 602 ms and 440.3 ms average value. InfluxDB achieves very similar performance to TimescaleDB but with a maximum value of 627 ms and a 442.35 ms average value. With complex queries like Q2, PostgreSQL starts to show some performance limitations where the maximum value records 1950 ms and the average value records 1707.15 ms.

Database	Min.	Mean	95%	Max.	Std. Dev.
ClickHouse	142	188.35	219.3	263	26.04
InfluxDB	387	442.35	512.05	627	54.96
TimescaleDB	314	440.3	544.05	602	81.61
PostgreSQL	1539	1707.15	1779	1950	90.71

Table 7.4: Query Latency Statistics (in ms) for Q2

Q3. Data Aggregation We query the database to calculate the standard deviation of the values of 10 sensors over a $60\,\mathrm{min}$ time interval. The $60\,\mathrm{min}$ duration is randomly selected from 15-day dataset using uniform distribution. Table 7.5 shows the query latency statistics in milliseconds for Q3. ClickHouse ranks first in Q3 performance with a maximum value of $244\,\mathrm{ms}$ and $203.55\,\mathrm{ms}$ average value. InfluxDB performs better than TimescaleDB with a maximum value of $594\,\mathrm{ms}$ and $427.85\,\mathrm{ms}$ average value while TimescaleDB records $791\,\mathrm{ms}$ as a maximum value and $571.95\,\mathrm{ms}$ average value, but TimescaleDB records a high standard deviation and lower minimum value than InfluxDB. PostgreSQL records the least performance for Q4 with a maximum value of $763\,\mathrm{ms}$ and an average value of $657.4\,\mathrm{ms}$.

Database	Min.	Mean	95%	Max.	Std. Dev.
ClickHouse	167	203.55	238.3	244	22.33
InfluxDB	280	427.85	555.05	594	69.04
TimescaleDB	268	571.95	691.25	791	106.54
PostgreSQL	600	657.4	737.35	763	47.12

Table 7.5: Query Latency Statistics (in ms) for Q3

Q4. Data Downsampling We query the database to summarize the data of 10 sensors over $24\,\mathrm{h}$ every hour. The $24\,\mathrm{h}$ duration is randomly selected from 15-day dataset using uniform distribution. Table 7.6 shows the query latency statistics in milliseconds for Q4. Even with a complex query like Q4, ClickHouse is still ranking first with a maximum value of 300 ms and 293.35 ms average value. InfluxDB and TimescaleDB give a reasonable performance with a maximum value of 873 ms and 647.9 ms average value for InfluxDB while 1024 ms as a maximum value and 754.6 ms average value for TimescaleDB. With a complex query like Q4, PostgreSQL records a bad performance that is ranging between 9858 ms and 14 157 ms and averaging at 13 445.95 ms.

Database	Min.	Mean	95%	Max.	Std. Dev.
ClickHouse	175	237.45	293.35	300	33.42
InfluxDB	464	647.9	816	873	87.15
TimescaleDB	548	754.6	965.1	1024	114.35
PostgreSQL	9858	13445.95	13974.6	14157	894.96

Table 7.6: Query Latency Statistics (in ms) for Q4

Q5. Operations on Two Down-sampled Sensors We query the database to calculate the difference between the summarized data of two sensors over 24 h. The data is summarized every one hour that is randomly selected from 15-day dataset using uniform distribution. Table 7.7 shows the query latency statistics in milliseconds for Q5. ClickHouse records the best performance with a maximum value of 419 ms and an average of 301.7 ms. TimescaleDB outperforms InfluxDB in this query with a maximum value of 701 ms and 448.6 ms on average for TimescaleDB while a maximum value of 810 ms and 522.4 ms on average for InfluxDB. PostgreSQL does not perform well for complex queries, it records very high latencies that are 20 806.15 ms on average.

ClickHouse provides an outstanding stable query performance because of its unique data storage. In addition to its foundational columnar format, ClickHouse partitions data in multiple files and uses a sparse indexing algorithm where indexes are stored for every N-th row of the table instead of indexing every row which

Database	Min.	Mean	95%	Max.	Std. Dev.
ClickHouse	167	301.7	397.15	419	68.43
InfluxDB	430	522.4	779.6	810	109.49
TimescaleDB	209	448.6	666.8	701	138.29
PostgreSQL	20344	20806.1	21134.8	21151	236.76

Table 7.7: Query Latency Statistics (in ms) for Q5

supports querying data in ranges as is the case of time-series data. Even for complex queries like Q4 and Q5, ClickHouse provides very good performance without being impacted because of the performance benefits of cross-breeding vectorized query execution and compiled query execution [103]. InfluxDB and TimescaleDB offer close performance while their backends are different but with conceptual similarities. InfluxDB uses the columnar format and a multi-level indexing mechanism where a query starts by determining in which partition/file the time range is, once the partition and its corresponding files are determined, InfluxDB does a binary search to find the requested data. On the other hand, TimescaleDB is row-based for recent data, but if compression is enabled, it uses a row-column hybrid model where the columns of multiple are stored in separate arrays. TimescaleDB queries start by determining which chunks have the requested data then it uses B-Tree indexes of this chunk to determine which rows have the data. The clear disadvantages of a complete row-based model and the absence of data partitioning are present with PostgreSQL.

7.6 Integration with Field-level Devices

TSDBs are promising candidates to design a scalable monitoring system for scientific infrastructure. Their design supports the data flow of process data in both stages of Storage and Offline Analytics. However, it is still an open question how to integrate them with field-level devices through the slow control system.

To integrate TSDBs with field-level devices, we recommend using a slow control system with standard communication protocols, e.g. IPv4-based protocols, which provide high interoperability with storage systems like TSDBs. An example of such a slow control systems are ones that are based on the Open Platform Communications Unified Architecture (OPC-UA) protocol. The OPC-UA protocol provides a standard communication protocol for industrial automation that is based on standard TCP/IP protocols to exchange data among machines or between machine and data storage like TSDBs. It is therefore highly interoperable especially that is maintained and sponsored by a worldwide consortium of the largest automation companies e.g. ABB, Siemens, Mitsubishi Electric, etc. An example of an OPC-UA-based slow control system is shown in our poster [4].

7.7 Conclusion

In this chapter, we proposed a novel monitoring system for scientific infrastructure based on time-series databases. We have shown that TSDBs are more scalable, available, faster in storing and retrieving process data in scientific infrastructure than traditional DBMS.

We present a benchmark-guided approach using the proposed SciTS benchmark to evaluate TSDBs and compare them to traditional ACID-based databases. The 4 evaluated databases using SciTS are: ClickHouse, InfluxDB, TimescaleDB, and PostgreSQL. We demonstrate the advantages of time-series databases as compared to traditional ACID-based databases like PostgreSQL as an example. Our evaluation shows that the selected TSDBs outperform PostgreSQL up to 6-times and preserves stable ingestion rate over time. The queries of the data mining and analysis workloads in scientific infrastructures even has a higher performance advantage.

Based on the evaluations from SciTS results, we plan to use ClickHouse to empower the monitoring system of the KATRIN infrastructure. With its simple but efficient storage engine, ClickHouse supports very high ingestion rates up to

1.3 million records per second. Even for high ingestion rates, ClickHouse keeps low CPU usage and a very low memory footprint. Therefore, ClickHouse can augment the performance and scalability of the scientific monitoring system of the KATRIN infrastructure.

8 Discussion and Conclusion

The Data acquisition Functions Virtualization (DFV) is a new data acquisition paradigm that minimizes the efforts to run, manage, and maintain software-based DAQ functions. DFV proposes to run software-based data acquisition functions on campus computing facilities that are already operated in most research institutes. The building blocks of DFV are high-performance computer networking and computer virtualization.

8.1 Conclusion and Research Results

To summarize the results of this thesis, we answer the questions formulated in the introduction.

> *Question 1: What are the qualifications to choose a suitable networking technology for Campus Computing Facilities and DFV?*

To design virtualized data acquisition functions, we need a networking technology that is suitable for the target campus computing facilities and respects the requirements of the scientific data acquisition process. In Chapter 3, we classify that a networking technology for DFV must handle the *high throughput of modern detector systems* by processing hundreds of gigabits per second. DFV expects the participation of detector electronics in the data transfer process to the software-based data acquisition function running on campus computing facilities. The protocols of this networking technology must be *resource-efficient* when implemented on detector electronics whose resources are considered scarce and expensive reducing DFV's overhead on the overall design of the data acquisition system. While running software-based data acquisition functions on campus computing facilities, they must integrate with campus computing facilities seamlessly. Therefore, DFV *must not change the environment of campus computing facilities* by requiring special hardware, violating their zero-trust security model, or breaking compatibility with other applications running simultaneously.

> *Question 2: What networking technology is a good compromise between performance and these qualifications to realize DFV?*

Chapter 3 surveys three networking technologies: DPDK, RDMA, and AF_XDP. Based on the qualifications in the answer of question 8.1, we choose AF_XDP as a candidate for high performance networking in DFV. The technology: (1) can process millions of packets per second supporting the hundreds of gigabits per second throughput of modern detector systems; (2) uses lightweight Ethernet protocols that are easy to implement on detector electronics with a few resources; and (3) is totally a software feature that does not impose strict requirements on its execution environment making it compatible with campus computing facilities. To evaluate AF_XDP empirically, we perform a top-down analysis of AF_XDP performance. Although AF_XDP is an emerging technology with some limitations, we conclude that it is a perfect fit for DFV. It uses Ethernet protocols that are resource-efficient for detector electronics, can process up to 16 million packets of seconds, and does not require a special hardware that majorly changes the environment of campus computing facilities. By looking at the software engineering metrics of AF_XDP like lines of code and count of commits, we also find that it is evolving and maturing to fix software bugs and enhance performance through new features like supporting NIC hardware offloading.

> *Question 3: What are the sources of higher data transfer latencies in general-purpose computer systems of campus computing facilities?*

Chapter 4 shows that memory hierarchy in general-purpose computer systems of campus computing facilities can impact the performance of a software-based data acquisition function. The memory architecture in modern general-purpose systems can increase the latency of memory accesses of DFV impacting their overall throughput. We find that the Non-Uniform Memory Access and Symmetric Multi-Processing memory architectures have significant overheads on DFV and software-based data acquisition functions by violating the Principle of Locality of memory. Unoptimized memory access patterns that do not store or access data in far memory can cause cache incoherence and an increased number of cache misses. This is especially true when the operating system interferes in the data acquisition function execution without taking into consideration the memory architecture underneath. Examples of such memory access patterns are False Sharing, Cache Contention and Thrashing, and memory accesses that do not consider the NUMA architecture of the computer system.

> *Question 4: How can we autotune a general-purpose computer system for DFV without significant intervention from scientists?*

Mitigating performance bottlenecks resulting from memory architecture require a solid understanding of computer architecture and operating systems principles. To simplify DFV-based data acquisition functions for scientists in a scientific infrastructure, we propose the Data acQuisition Development Kit (DQDK) in Chapter 4. DQDK provides all the needed boilerplate code to mitigate memory bottlenecks without significant intervention from scientists through adopting a cache-aware design. We have proven that DQDK is scalable to handle the rising demands in detectors' throughput, and it can integrate with available execution environment seamlessly by auto-detection and autoconfiguration of underlying memory architecture. While DQDK is designed to handle hundreds of gigabits per second, it provides great compatibility with other applications running simultaneously on computer systems. DQDK adopts the rest of DFV design principles by exploiting AF_XDP for high-performance networking and suitability for virtualization in campus computing facilities, and by using resource-efficient UDP and IPv4 Ethernet protocols. The DQDK framework prevents data loss while maintaining compatibility with other applications by isolating computing resources and the DMA memory buffers of the software-based DF from that of the operating system and other applications. We assume the bandwidth of the network link in the computer cluster at campus computing facilities is wide enough to serve the throughput of the software-based DF and other running applications.

> *Question 5: What is a computer virtualization model that can handle hundreds of gigabits per second in throughput?*

Campus computing facilities have different computer virtualization models to support DFV. In Chapter 5, we evaluate the overhead of 4 different virtualization setups on DQDK-based data acquisition functions. We consider both virtual machines and containers equipped with hardware-based and software-based network virtualization. We deduce that software-based network virtualization limits DQDK performance for only $12.78\,\mathrm{Gbit/s}$ at maximum. The path to $100+\,\mathrm{Gbit/s}$ for modern detector systems is only possible through hardware-based network virtualization in both virtual machines and containers. However, we notice that using that hardware-based network virtualization can degrade DQDK performance due to network and CPU virtualization opposite to using containers which performs similarly to DQDK without virtualization. Thus, we recommend using a container-based architecture due to their minimal overhead on software-based data acquisition functions.

8.1.1 Contributions

This work has proposed Data Acquisition Functions Virtualization, a new data acquisition paradigm for scientific instrumentation and infrastructure. To realize this paradigm, several contributions had to be made. We survey high-performance networking technologies for Ethernet-based readout in DAQ systems. We study their qualifications and quantify their performance. The impact of modern general-purpose computer systems has been studied and quantified. A framework for high-performance Ethernet-based readout on general-purpose computer systems, that we call the *Data Acquisition Development Kit*, has been designed and implemented as an open-source project available publicly on GitHub:

https://github.com/kit-ipe/dqdk. We also study the 4 virtualization setups to realize DFV, and we quantify the impact of each on the performance of a software-based data acquisition function.

After constructing the building blocks of DFV, we apply it to the infrastructure of the TRISTAN experiment. The TRISTAN experiment will employ a new detector whose data rates can reach up to $200\,\mathrm{Gbit/s}$ and will increase the operational efforts in the KATRIN infrastructure. We take the detector of the TRISTAN experiment as a use-case for DFV. DFV lowers the operational efforts of the TRISTAN detector by eliminating the requirement to deploy a computer cluster at the site of the TRISTAN experiment. Our results show that beside lowering the operational efforts, DFV can save up to 2.67x less computing resources and up to 15% of consumed energy.

DFV is adaptable to the software-based data acquisition functions from other scientific infrastructure like DUNE and Belle II as discussed in Section 2.1.2. This is because DFV relies on general-purpose computer systems and standard Ethernet protocols that are already in use in some of these experiments, e.g. DUNE uses UDP. However, separate evaluations must be done for each of these experiments by porting their software-based DFs to the DQDK framework.

8.1.2 Limitations

The current design of DFV has some limitations. As it is now, DFV is designed only for software-based data acquisition functions and does not intend to replace detector electronics or hardware-based data acquisition functions. DFV performance is dependent on the number of the physical CPU cores available on the hosting computer system. Our implementation only supports up to $4\,\mathrm{kB}$-MTUs. DFV has not been tested with software-defined networking switches, e.g. OpenvSwitch or Cilium, used extensively in campus computing facilities. However, the Cilium software switch has been reported to achieve up to $90\,\mathrm{Gbit/s}$ [104].

It assumes there is a network link of the required bandwidth between the detector electronics and the software-based DF running in campus computing facilities. This means that there will be at least one network link for each scientific infrastructure running on campus computing facilities. Due to the probable long distance between detector electronics and campus computing facilities, some network setups may use several Ethernet routers or switches on the way to campus computing facilities. We recommend signing a service level agreement with the operators of campus computing facilities to ensure their network setup does not drop data on the way. Technically, this is possible through lossless Ethernet switches and quality of service protocols.

DFV considers the user is responsible for the design and performance of the DAQ logic that is provided to the DQDK framework. However, we recommend reusing the memory allocator of the DQDK framework as suggested in Chapter 4.

8.2 Future Work

Modern computing technologies are pushing the boundaries of high-performance computing and data processing. There are still opportunities to improve DFV by integrating emerging computing technologies and interconnects. This section discusses some research ideas that might improve DFV.

8.2.1 Integration of Hardware Accelerators

Hardware accelerators are new hardware components designed to execute specific workloads more efficiently than general-purpose computer architecture and memory architecture. Here, we discuss opportunities to integrate 2 hardware accelerators: Smart Network Interface Cards and General-Purpose Graphics Processing Units.

8.2.1.1 Near-Memory Processing using Smart Network Interface Cards

Smart Network Interface Cards, in short Smart NICs, are programmable NICs that make computer networking more efficient. Smart NICS reduce data movements between the NIC and main memory by directly processing near to its source in the NIC buffers. This form of computing is called Near-Memory Processing where the NIC has its own programmable processing unit that can directly access the NIC buffers to mitigate the data movement overhead from the NIC to main memory over interconnects like PCIe or from main memory to CPU caches and registers. Examples of commercially and currently available Smart NICs are: NVIDIA BlueField-3 (400 Gbit/s with 16 ARMv8.2 cores - 256 threads in total, 8 MB L2 Cache, 16 MB L3 Cache, 32 GB DDR5 RAM, and 128 GB SSD), and AMD ALVEO SN1000 (200 Gbit/s with Xilinx XCU26 FPGA + 2x 4 GB DDR4 RAM and a 16 64-bit Arm Cortex®-A72 cores at 2.0 GHz + 8 MB L3 Cache).

Example of such a use-case is building TRISTAN energy histograms inside the memory of a Smart NIC. This would eliminate the need for energy events to go outside the NIC. The histogram is going to be computed inside the NIC, and only moved to the main memory and the CPU when the DAQ process is finished. However, it is important to study what the performance, cost, and energy benefits of this approach to DFV would be.

8.2.1.2 Parallelization through General-Purpose Graphics Processing Units

Initially designed for processing computer graphics, General-Purpose Graphics Processing Units (GPGPUs) are being adopted in several fields to parallelize computation-bound workloads. Due to their high processing core count (usually thousands of processing cores), they excel in performing parallel processing of multiple chunks of the same data. Most notably, GPGPUs perform very well in matrix arithmetic workloads that are used extensively in artificial intelligence.

In DAQ systems, GPGPUs plays an important role in accelerating software-based data acquisition functions. For example, the HLT algorithm of the CMS experiment started using GPGPUs from Run 3 to reduce the processing latency from $690.1\,\mathrm{ms}$ to $397.8\,\mathrm{ms}$ per event [22].

Because DFV is deployed on Campus Computing Facilities where GPGPUs are usually available, it can exploit their parallelization power. However, GPGPUs usually do not share the same memory with their hosts. Thus, network data coming at a NIC has to be moved first to main memory and then copied from the host memory to the GPU internal memory which significantly impact performance. NVIDIA GPUDirect reduce data movement by performing high-performance data transfer through RDMA from the NIC directly to the GPU without passing through the host memory. Since GPUDirect uses RDMA, detector electronics has to participate in the data transfer by implementing the RDMA which might consume a lot of resources as discussed in Chapter 3.

DFV can make use of GPGPUs available of Campus Computing Facilities. However, to efficiently integrate GPGPUs in DFV, an open question is: how to reduce data transfer between host memory and GPU memory through AF_XDP?

8.2.2 DFV and Hardware-based Data Acquisition Functions

Due to the flexibility of software, scientists have started to softwarize more components of data acquisition systems. There is an increasing trend in adopting software solutions instead of hardware one when possible in order to simplify the design and the operation of data acquisition systems. The work in [6] proposes the Software-Defined Data Acquisition (SD-DAQ) paradigm which transforms multiple data acquisition functions from hardware-based functions to software-based functions running on commercial off-the-shelf hardware like GPUs and RDMA-based Data Transfer. Their work shows that commercial off-the-shelf hardware is only $2\,\mu\mathrm{s}$-far from adopting GPGPUs for the Hough-Transformation-based track-finding algorithm that is usually implemented on detector electronics in the

CMS experiment [6]. The DUNE experiment has also succeeded to softwarize its online data reduction algorithm by adopting a new design for Ethernet-based readout [19]. The CERN roadmap for scientific infrastructure and published in 2019 has put cloud computing and computer virtualization as one of the targets to implement in the future for the infrastructure of the Large Hadron Collider [36].

Enabling DFV for hardware-based data acquisition functions is another step in this softwarization trend. DFV will not only provide higher flexibility through softwarization of hardware-based functions but will also increase their maintainability and availability through computer virtualization. To realize DFV for hardware-based data acquisition functions several performance challenges has to be taken care of. However, with FPGA-based Smart NICs or the availability of FPGA-based acceleration as a commercial product in the data center, these performance requirements might no longer be a barrier. We keep this feasibility study as an open question whose answer is to be studied in a dedicated research project.

A AF_XDP Features

A.1 Drivers Implementations for AF_XDP

This section provides a list of all 39 AF_XDP kernel drivers and the corresponding supported features as of Linux kernel v6.5-rc2 (the latest kernel release at the time of writing this section). Table. A.1 show the list of these drivers by the vendor and the supported AF_XDP features. The column *Mode* in the table tells if AF_XDP zero-copy mode (denoted by *ZC*) or if only copy mode (denoted by *C*) is supported. The column *Need Wake Up* tells if the need wake up flag (added in Linux kernel 5.4) is supported by the driver. This flag is a necessity if the developer wants to run the application and the interrupt handler on the same core. Running an AF_XDP application and its interrupt handler on the same core without this flag will kill performance [61]. We extract the information from the kernel drivers source code: zero-copy support is deduced using the drivers' support for AF_XDP user-space memory pools, specifically the `XDP_SETUP_XSK_POOL` flag, and the support for the *need wake-up* flag is deduced by checking the driver's calls to the functions: `xsk_set_rx_need_wakeup` and `xsk_set_tx_need_wakeup` which are actually responsible for enforcing the need wake-up logic.

Table A.1: AF_XDP Drivers Support in Linux Kernel v6.5-rc2

Vendor	Driver	Mode	Need Wake Up
Intel	i40e	ZC/C	✓
	ice	ZC/C	✓

Continued on next page

Table A.1: AF_XDP Drivers Support in Linux Kernel v6.5-rc2 (Continued)

Vendor	Driver	Mode	Need Wake Up
	igb	C	✗
	igc	ZC/C	✓
	ixgbe	ZC/C	✓
	ixgbevf	C	✗
NVIDIA/Mellanox	mlx5	ZC/C	✓
	mlx4	C	✗
Broadcom	bnxt	C	✗
Netronome/Corigine	nfp	ZC/C	✗
Marvell	mvneta	C	✗
	mvpp2	C	✗
	octeontx2	C	✗
Qlogic (now Marvell)	qede	C	✗
Cavium (now Marvell)	thunder	C	✗
Aquantia (now Marvell)	atlantic	C	✗
MediaTek	mtk	C	✗
MicroChip	lan966x	C	✗
SolarFlare (now Xilinx)	efx	C	✗
	siena-efx	C	✗
SocioNext	netsec	C	✗
STMicroelectronics	stmmac	ZC/C	✓
Texas Instruments	cpsw	C	✗
Freescale (now NXP)	dpaa	C	✗

Continued on next page

Table A.1: AF_XDP Drivers Support in Linux Kernel v6.5-rc2 (Continued)

Vendor	Driver	Mode	Need Wake Up
	dpaa2	ZC/C	✗
	enetc	C	✗
	fec	C	✗
Engleder	tsnep	ZC/C	✓
Fungible (now Microsoft)	funeth	C	✗
Microsoft	mana	C	✗
Microsoft Hyper-V	netvsc	C	✗
Amazon	ena	C	✗
Google	gve	ZC/C	only on TX path
Xen	netfront	C	✗
VirtIO	virtio_net	C	✗
Linux	tun	C	✗
	veth	C	✗
	bonding	C	✗
	netdevsim	C	✗

Acronyms & Symbols

KATRIN
Karlsruhe Tritium Neutrino Experiment

DF
Data Acquisition Function

CMS
Compact Muon Solenoid

LHC
Large Hadron Collider

DAQ
Data Acquisition

UDP
User Datagram Protocol

TCP
Transmission Control Protocol

IPv4
Internet Protocol version 4

IPv6
Internet Protocol version 6

ASIC
Application–Specific Integrated Circuit

GPGPU
General Purpose Graphics Processing Unit

FPGA
Field Programmable Gateway Array

MPI
Message Passing Interface

OS
Operating System

I/O
Input/Output

DPDK
Data Plane Development Kit

RDMA
Remote Direct Memory Access

AF_XDP
Address Family Express Data Path

XDP
Express Data Path

eBPF
Extended Berkeley Packet Filter

DMA
Direct Memory Access

PCIe
Peripheral Component Interconnect Express

UIO
User-space Input/Output

VFIO
Virtual Function Input/Output

MMIO
Memory-Mapped Input/Output

BIOS
Basic Input/Output System

CPU
Central Processing Unit

SMI
System Management Interrupts

API
Application Programming Interface

LTS
Long-Term Support

RoCE
RDMA over Converged Ethernet

MTU
Maximum Transfer Unit

RTT
Round Trip Time

CQE
Completion Queue Entry

MPWQE
Multi-Packet Work Queue Entry

SKB
Socket Buffers

DQDK
Data Acquisition Development Kit

DRAM
Dynamic Random Access Memory

SRAM
Static Random Access Memory

FIFO
First-In First-Out

LRU
Least Recently Used

UPI
Ultra Path Interconnect

NUMA
Non-Uniform Memory Access

ccNUMA
Cache-Coherent Non-Uniform Memory Access

IETF
Internet Engineering Task Force

RFC
Request For Comments

SIMD
Single Instruction Multiple Data

DDIO
Data Direct Input/Output

IoT
Internet of Things

WLCG
Worldwide LHC Computing Grid

RoI
Regions of Interest

CERN
European Organization for Nuclear Research

HLT
High-Level Trigger

KARA
Karlsruhe Research Accelerator

FLUTE
Far-infrared Linac and Test Experiment

DUNE
Deep Underground Neutrino Experiment

LCRC
Link Cyclic Redundancy Check

ECRC
End to End Cyclic Redundancy Check

IOMMU
Input/Output Memory Management Unit

IOVA
Input/Output Virtual Address

IRQ
Interrupt Request

GRO
Generic Receive Offload

RPS
Receive Packet Steering

RFS
Receive Flow Steering

LRO
Large Receive Offload

aRFS
Accelerated Receive Flow Steering

VM
Virtual Machine

VMM
Virtual Machine Monitor

VMX
Virtual Machine Extensions

KVM
Kernel-based Virtual Machine

VMCS
Virtual Machine Control Structure

cgroups
Control Groups

SECCOMP
Secure Computing

SELinux
Security-Enhanced Linux

JUNO
Jiangmen Underground Neutrino Observatory

TRISTAN
Tritium Sterile Anti-Neutrino Experiment

SDD
Silicon Drift Detector

KaaS
KATRIN as a Service

SCC
Scientific Computing Center

VETH
Virtual Ethernet

SRIOV
Single Root Input/Output Virtualization

SoC
System-on-Chip

OPC – UA
Open Platform Communications Unified Architecture

Bibliography

First-Author Publications

[1] J. Mostafa, D. Tcherniakhovski, S. Chilingaryan, M. Balzer, A. Kopmann, and J. Becker, “100-gbit/s udp data acquisition on linux using af_xdp: The tristan detector,” *IEEE Transactions on Nuclear Science*, vol. 72, no. 3, pp. 295–300, 2025. doi: `10.1109/TNS.2024.3452469`.

[2] J. Mostafa, S. Chilingaryan, and A. Kopmann, “Are kernel drivers ready for accelerated packet processing using af_xdp?” In *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2023, pp. 117–122. doi: `10.1109/NFV-SDN59219.2023.10329590`.

[3] J. Mostafa, S. Wehbi, S. Chilingaryan, and A. Kopmann, “Scits: A benchmark for time-series databases in scientific experiments and industrial internet of things,” in *Proceedings of the 34th International Conference on Scientific and Statistical Database Management*, ser. SSDBM ’22, Copenhagen, Denmark: Association for Computing Machinery, 2022, isbn: 9781450396677. doi: `10.1145/3538712.3538723`. [Online]. Available: `https://doi.org/10.1145/3538712.3538723`.

[4] J. Mostafa, A. Beglarian, S. Chilingaryan, and A. Kopmann, “Interfacing EPICS and LabVIEW Using OPC UA for Slow Control Systems,” *JACoW*, vol. ICALEPCS2021, TUPV011, 2022. doi: `10.18429/JACoW-ICALEPCS2021-TUPV011`.

Further References

[5] T. A. Bawej, U. Behrens, J. Branson, *et al.*, "The New CMS DAQ System for Run 2 of the LHC," CERN, Geneva, Tech. Rep., 2015. DOI: `10.1109/RTC.2014.7097437`. [Online]. Available: `https://cds.cern.ch/record/1711011`.

[6] T. Dritschler, "High-performance commodity data acquisition systems for scientific applications in the terascale era," Ph.D. dissertation, Karlsruhe Institute of Technology, 2023.

[7] J.-M. Andre, U Behrens, J Branson, *et al.*, "The cms data acquisition - architectures for the phase-2 upgrade," *Journal of Physics: Conference Series*, vol. 898, no. 3, p. 032 019, Oct. 2017. DOI: `10.1088/1742-6596/898/3/032019`. [Online]. Available: `https://dx.doi.org/10.1088/1742-6596/898/3/032019`.

[8] T. C. Collaboration, "The cms high level trigger," *The European Physical Journal C - Particles and Fields*, vol. 46, no. 3, pp. 605–667, Jun. 2006, ISSN: 1434-6052. DOI: `10.1140/epjc/s2006-02495-8`. [Online]. Available: `https://doi.org/10.1140/epjc/s2006-02495-8`.

[9] P. Jenni, M. Nessi, M. Nordberg, and K. Smith, *ATLAS high-level trigger, data-acquisition and controls: Technical Design Report* (Technical design report. ATLAS). Geneva: CERN, 2003. [Online]. Available: `https://cds.cern.ch/record/616089`.

[10] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, p. 160 018, Mar. 2016, ISSN: 2052-4463. DOI: `10.1038/sdata.2016.18`. [Online]. Available: `https://doi.org/10.1038/sdata.2016.18`.

[11] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl, and G. Carle, "User space network drivers," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Washington, DC, USA: IEEE, 2019, pp. 1–12. DOI: 10.1109/ANCS.2019.8901894.

[12] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in xen," in *2006 USENIX Annual Technical Conference (USENIX ATC 06)*, Boston, MA: USENIX Association, May 2006. [Online]. Available: https://www.usenix.org/conference/2006-usenix-annual-technical-conference/optimizing-network-virtualization-xen.

[13] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rfaas: Enabling high performance serverless with rdma and leases," in *Proceedings of the 37th IEEE Interational Parallel and Distributed Processing Symposium*, ser. IPDPS '23, 2023. eprint: 2106.13859.

[14] C. Karle, M. Neu, J. Pfau, J. Sperling, and J. Becker, "Relodaq: Resource-efficient, low-overhead 200 Gbits^{-1} data acquisition system for 6g prototyping," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 209–209. DOI: 10.1109/FCCM57271.2023.00037.

[15] W. Mansour, N. Janvier, and P. Fajardo, "Fpga implementation of rdma-based data acquisition system over 100-gb ethernet," *IEEE TNS*, vol. 66, no. 7, pp. 1138–1143, 2019. DOI: 10.1109/TNS.2019.2904118.

[16] N. Schelten, F. Steinert, A. Schulte, and B. Stabernack, "A high-throughput, resource-efficient implementation of the rocev2 remote dma protocol for network-attached hardware accelerators," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 241–249. DOI: 10.1109/ICFPT51103.2020.00042.

[17] L. Scomparin, E. Blomley, E. Bründermann, *et al.*, "Kingfisher: A framework for fast machine learning inference for autonomous accelerator systems," in *Data Acquisition and Processing Platforms - International Beam*

Instrumentation Conference (11th), Kraków, Poland, 11-15 September 2022, (Krakau, Polen, Sep. 11–15, 2022), 54.12.03; LK 01, JACoW Publishing, 2022, pp. 151–155, ISBN: 978-3-95450-241-7. DOI: 10.18429/JACoW-IBIC2022-MOP42.

[18] B. Abi, R. Acciarri, M. Acero, *et al.*, "Volume i. introduction to dune," *Journal of Instrumentation*, vol. 15, no. 08, T08008, Aug. 2020. DOI: 10.1088/1748-0221/15/08/T08008. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/15/08/T08008.

[19] R. Sipos, "The ethernet readout of the dune daq system," *IEEE Transactions on Nuclear Science*, pp. 1–1, 2024. DOI: 10.1109/TNS.2024.3486059.

[20] D. Hufnagel, C. Offline, and Computing, "The architecture and operation of the cms tier-0," *Journal of Physics: Conference Series*, vol. 331, no. 3, p. 032 017, Dec. 2011. DOI: 10.1088/1742-6596/331/3/032017. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/331/3/032017.

[21] T. Bawej, U. Behrens, J. Branson, *et al.*, "Achieving high performance with tcp over 40gbe on numa architectures for cms data acquisition," in *2014 19th IEEE-NPSS Real Time Conference*, 2014, pp. 1–1. DOI: 10.1109/RTC.2014.7097439.

[22] Parida, Ganesh, "Run-3 commissioning of cms online hlt reconstruction using gpus," *EPJ Web of Conf.*, vol. 295, p. 11 020, 2024. DOI: 10.1051/epjconf/202429511020. [Online]. Available: https://doi.org/10.1051/epjconf/202429511020.

[23] S. Varghese, "CMS High Level Trigger Performance for Run 3," CERN, Geneva, Tech. Rep., 2023. DOI: 10.22323/1.449.0517. [Online]. Available: https://cds.cern.ch/record/2879813.

[24] M. T. Prim, N. Braun, Y. Guan, *et al.*, "Design and Performance of the Belle II High Level Trigger," in *Proceedings of 40th International Conference on High Energy physics — PoS(ICHEP2020)*, vol. 390, 2021, p. 769. DOI: 10.22323/1.390.0769.

[25] PCI-SIG, *PCI Express® Base Specification Revision 6.0, Version 1.0*, `https://pcisig.com/specifications/pciexpress`, PCI Special Interest Group (PCI-SIG), Jan. 2022.

[26] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, Virtual Event, USA: Association for Computing Machinery, 2021, 65–77, ISBN: 9781450383837. DOI: `10.1145/3452296.3472888`. [Online]. Available: `https://doi.org/10.1145/3452296.3472888`.

[27] A. Lerner, C. Binnig, P. Cudré-Mauroux, *et al.*, "Databases on modern networks: A decade of research that now comes into practice," *Proc. VLDB Endow.*, vol. 16, no. 12, 3894–3897, 2023, ISSN: 2150-8097. DOI: `10.14778/3611540.3611579`. [Online]. Available: `https://doi.org/10.14778/3611540.3611579`.

[28] H. Aoba, R. Kawashima, and H. Matsuo, "Accelerating key-value store with layer-2 transparent proxy cache using dpdk," in *2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*, 2023, pp. 298–302. DOI: `10.1109/CANDARW60564.2023.00056`.

[29] A. Bhattacharyya, S. Ramanathan, A. Fumagalli, and K. Kondepu, "An end-to-end dpdk-integrated open-source 5g standalone radio access network: A proof of concept," *Computer Networks*, vol. 250, p. 110 533, 2024, ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2024.110533`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S1389128624003657`.

[30] V. Jain, S. Panda, S. Qi, and K. K. Ramakrishnan, "Evolving to 6g: Improving the cellular core to lower control and data plane latency," in *2022 1st International Conference on 6G Networking (6GNet)*, 2022, pp. 1–8. DOI: `10.1109/6GNet54646.2022.9830519`.

[31] Microsoft, *Ebpf for windows*, online, 2025. [Online]. Available: `https://github.com/microsoft/ebpf-for-windows`.

[32] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, *et al.*, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: Association for Computing Machinery, 2018, 54–66, ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. [Online]. Available: https://doi.org/10.1145/3281411.3281443.

[33] S. A. S. Kohroudi, J. Mostafa, M. Mohiuddin, W. Saab, and J.-Y. L. Boudec, "Experimental validation of the suitability of virtualization-based replication for fault tolerance in real-time control of electric grids," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18, Oulu, Finland: Association for Computing Machinery, 2018, ISBN: 9781450358231. DOI: 10.1145/3239235.3267429. [Online]. Available: https://doi.org/10.1145/3239235.3267429.

[34] K. Bos, N. Brook, D. Duellmann, *et al.*, *LHC computing Grid: Technical Design Report. Version 1.06 (20 Jun 2005)* (Technical design report. LCG). Geneva: CERN, 2005. [Online]. Available: https://cds.cern.ch/record/840543.

[35] CERN Openlab, "Future ICT Challenges in Scientific Research," CERN, Tech. Rep., 2017, https://cds.cern.ch/record/2301895/files/Whitepaper_brochure_ONLINE.pdf.

[36] J. Albrecht, A. A. Alves, G. Amadio, *et al.*, "A roadmap for hep software and computing r&d for the 2020s," *Computing and Software for Big Science*, vol. 3, no. 1, p. 7, Mar. 2019, ISSN: 2510-2044. DOI: 10.1007/s41781-018-0018-8. [Online]. Available: https://doi.org/10.1007/s41781-018-0018-8.

[37] Avolio, Giuseppe, Cadeddu, Mattia, and Hauser, Reiner, "Evaluating kubernetes as an orchestrator of the event filter computing farm of the trigger and data acquisition system of the atlas experiment at the large hadron collider," *EPJ Web Conf.*, vol. 214, p. 07 024, 2019. DOI: 10.1051/

epjconf/201921407024. [Online]. Available: https://doi.org/10.1051/epjconf/201921407024.

[38] T. A. Collaboration, *Total cost of ownership and evaluation of google cloud resources for the atlas experiment at the lhc*, 2024. arXiv: 2405.13695. [Online]. Available: https://arxiv.org/abs/2405.13695.

[39] M. M. Storetvedt, "A new grid workflow for data analysis within the alice project using containers and modern cloud technologies," https://hdl.handle.net/11250/3061978, Ph.D. dissertation, Western Norway University of Applied Sciences, 2023.

[40] T. K. collaboration, M. Aker, K. Altenmüller, *et al.*, "The design, construction, and commissioning of the katrin experiment," *Journal of Instrumentation*, vol. 16, no. 08, T08015, Aug. 2021. DOI: 10.1088/1748-0221/16/08/T08015. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/16/08/T08015.

[41] J. Li, M. Gu, F. Li, and K. Zhu, "An soa-based design of juno daq online software," *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1199–1203, 2019. DOI: 10.1109/TNS.2019.2907367. [Online]. Available: https://ieeexplore.ieee.org/document/8674600.

[42] F. Zhang, N. Wang, Z. Hu, *et al.*, "A study of udp and tcp fpga implementation for data acquisition system," *JINST*, vol. 16, no. 07, P07044, Jul. 2021. DOI: 10.1088/1748-0221/16/07/P07044. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/16/07/P07044.

[43] J. Subraveti, "Implementation of UDP communication on a ZYNQ platform for processing clustered data based on multiple Gigabit Ethernet ports for phenoPET," Masterarbeit, Hochsch. Bremerhaven, 2017, Masterarbeit, Hochsch. Bremerhaven, Jülich, 2018, iv, 56 p. [Online]. Available: https://juser.fz-juelich.de/record/844070.

[44] M. Christensen and T. Richter, "Achieving reliable UDP transmission at 10 gb/s using BSD socket for data acquisition systems," *JINST*, vol. 15, no. 09, T09005, Sep. 2020. DOI: 10.1088/1748-0221/15/09/T09005.

[Online]. Available: https://dx.doi.org/10.1088/1748-0221/15/09/T09005.

[45] T Dritschler, S Chilingaryan, T Farago, A Kopmann, and M Vogelgesang, "Infiniband interconnects for high-speed data acquisition in a tango environment," in *Proceedings of PCaPAC2014*, ser. PCaPAC 2014, Karlsruhe, Germany, 2014.

[46] W. Mansour, P. Fajardo, and N. Janvier, "High Performance RDMA-Based Daq Platform Over PCIe Routable Network," in *Proc. of International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS'17), Barcelona, Spain, 8-13 October 2017*, (Barcelona, Spain), ser. International Conference on Accelerator and Large Experimental Control Systems, https://doi.org/10.18429/JACoW-ICALEPCS2017-THBPL06, Geneva, Switzerland: JACoW, Jan. 2018, pp. 1131–1136, ISBN: 978-3-95450-193-9. DOI: https://doi.org/10.18429/JACoW-ICALEPCS2017-THBPL06. [Online]. Available: http://jacow.org/icalepcs2017/papers/thbpl06.pdf.

[47] R. Sipos, "The ethernet readout of the dune daq system," *IEEE Transactions on Nuclear Science*, vol. 72, no. 3, pp. 317–324, 2025. DOI: 10.1109/TNS.2024.3486059.

[48] D. Géhberger, D. Balla, M. Maliosz, and C. Simon, "Performance evaluation of low latency communication alternatives in a containerized cloud environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 9–16. DOI: 10.1109/CLOUD.2018.00009.

[49] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO: USENIX Association, Jun. 2016, pp. 437–450, ISBN: 978-1-931971-30-0. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[50] W. A. Hanafy, L. Wang, H. Chang, S. Mukherjee, T. V. Lakshman, and P. Shenoy, "Understanding the benefits of hardware-accelerated communication in model-serving applications," in *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*, 2023, pp. 1–10. DOI: 10.1109/IWQoS57198.2023.10188785.

[51] Q. Li, Z. Ding, X. Tong, *et al.*, "6g cloud-native system: Vision, challenges, architecture framework and enabling technologies," *IEEE Access*, vol. 10, pp. 96 602–96 625, 2022. DOI: 10.1109/ACCESS.2022.3205341.

[52] R. J. Recio, P. R. Culley, D. Garcia, B. Metzler, and J. Hilland. "A Remote Direct Memory Access Protocol Specification." (Oct. 2007), [Online]. Available: https://www.rfc-editor.org/info/rfc5040.

[53] M. Beck and M. Kagan, "Performance evaluation of the rdma over ethernet (roce) standard in enterprise data centers infrastructure," in *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, ser. DC-CaVES '11, San Francisco, California: International Teletraffic Congress, 2011, 9–15, ISBN: 9780983628323.

[54] DPDK Project. "Data Plane Development Kit." (2022), [Online]. Available: https://www.dpdk.org/.

[55] Snabb. "Snabb." (2022), [Online]. Available: https://github.com/snabbco/snabb.

[56] NVIDIA. "Nvidia nics performance report with dpdk 23.07." (2023), [Online]. Available: https://fast.dpdk.org/doc/perf/DPDK_23_07_NVIDIA_NIC_performance_report.pdf.

[57] L. Askari, P. Majidzadeh, O. Ayoub, and M. Tornatore, "Exploiting dpdk in containerized environment with unsupported hardware," in *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2020, pp. 7–12. DOI: 10.1109/NFV-SDN50289.2020.9289904.

[58] G. Gaio, L. Anastasio, A. Bogani, *et al.*, "A New Real-Time Processing Platform for the Elettra 2.0 Storage Ring," *JACoW*, vol. ICALEPCS2023, TUMBCMO24, 2023. DOI: 10.18429/JACoW-ICALEPCS2023-TUMBCMO24.

[59] M. Karlsson and B. Töpel, "The Path to DPDK Speeds for AF_XDP," in *Linux Plumbers Conference*, 2018.

[60] D. Miller, *Generic xdp*, 2017. [Online]. Available: https://lwn.net/Articles/720072/.

[61] M. Karlsson, *add need_wakeup flag to the AF_XDP rings*, 2019. [Online]. Available: https://lore.kernel.org/netdev/1560411450-29121-1-git-send-email-magnus.karlsson@intel.com/.

[62] P. Waskiewicz Jr., A. Singhai Jain, N. Parikh, and P. Sarangam, "Accelerating XDP programs using hw-based hint," in *Netdev Conference*, 2017.

[63] S. Bradner and J. McQuaid, *Benchmarking Methodology for Network Interconnect Devices*, RFC 2544, Mar. 1999. DOI: 10.17487/RFC2544. [Online]. Available: https://www.rfc-editor.org/info/rfc2544.

[64] Mellanox Technologies. "Nvidia mlx5 ethernet driver." (2023), [Online]. Available: https://doc.dpdk.org/guides-23.03/nics/mlx5.html.

[65] M. Primorac, E. Bugnion, and K. Argyraki, "How to measure the killer microsecond," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 5, 61–66, Aug. 2017, ISSN: 0146-4833. DOI: 10.1145/3155055.3155065. [Online]. Available: https://doi.org/10.1145/3155055.3155065.

[66] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with linux ebpf," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, 2018, pp. 209–217. DOI: 10.1109/ITC30.2018.00039.

[67] Mellanox Technologies. "Mlx5 maximum frame size in xdp." (2018), [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/net/ethernet/mellanox/mlx5/core/en/xdp.c?h=v6.0.5#n38.

[68] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, 20–24, Mar. 1995, ISSN: 0163-5964. DOI: 10.1145/216585.216588. [Online]. Available: https://doi.org/10.1145/216585.216588.

[69] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," in *2007 IEEE 10th International Symposium on Workload Characterization*, 2007, pp. 35–43. DOI: 10.1109/IISWC.2007.4362179.

[70] W. Choe, S.-H. Kim, and J. Ahn, "Rethinking remote memory placement on large-memory systems with path diversity," in *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '21, Hong Kong, China: Association for Computing Machinery, 2021, 63–69, ISBN: 9781450386982. DOI: 10.1145/3476886.3477516. [Online]. Available: https://doi.org/10.1145/3476886.3477516.

[71] *User Datagram Protocol*, RFC 768, Aug. 1980. DOI: 10.17487/RFC0768. [Online]. Available: https://www.rfc-editor.org/info/rfc768.

[72] *Internet Protocol*, RFC 791, Sep. 1981. DOI: 10.17487/RFC0791. [Online]. Available: https://www.rfc-editor.org/info/rfc791.

[73] M. Carminati, M. Gugiatti, D. Siegmann, *et al.*, "The tristan 166-pixel detector: Preliminary results with a planar setup," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 1049, p. 168 046, 2023, ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2023.168046. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168900223000360.

[74] RedHat, *Af_xdp plugins for kubernetes*, online, 2025. [Online]. Available: https://github.com/redhat-et/afxdp-plugins-for-kubernetes.

[75] J. Li, S. Xue, W. Zhang, R. Ma, Z. Qi, and H. Guan, "When i/o interrupt becomes system bottleneck: Efficiency and scalability enhancement for sr-iov network virtualization," *IEEE Transactions on Cloud Computing*, vol. 7, no. 4, pp. 1183–1196, 2019. DOI: 10.1109/TCC.2017.2712686.

[76] T. group in the KATRIN Collaboration, *Conceptual design report: Katrin with tristan modules*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248068381.

[77] K. Collaboration, M. Aker, D. Batzler, *et al.*, "Direct neutrino-mass measurement based on 259 days of katrin data," *Science*, vol. 388, no. 6743, pp. 180–185, 2025. DOI: 10.1126/science.adq9592. eprint: https://www.science.org/doi/pdf/10.1126/science.adq9592. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.adq9592.

[78] I. Park, "The online central data acquisition system of zeus," *IEEE Transactions on Nuclear Science*, vol. 41, no. 1, pp. 18–24, 1994. DOI: 10.1109/23.281450.

[79] M. Howe, G. Cox, P. Harvey, *et al.*, "Sudbury neutrino observatory neutral current detector acquisition software overview," in *2003 IEEE Nuclear Science Symposium. Conference Record (IEEE Cat. No.03CH37515)*, vol. 1, 2003, 169–173 Vol.1. DOI: 10.1109/NSSMIC.2003.1352023.

[80] M. P. I. for Physics, *Tristan*, online, 2025. [Online]. Available: https://www.hll.mpg.de/2967603/TRISTAN.

[81] N. T. Jerome, T. Dritschler, S. Chilingaryan, and A. Kopmann, "Bora: A personalized data display for large-scale experiments," *IEEE Transactions on Nuclear Science*, vol. 72, no. 3, 498–505, 2025, 54.12.02; LK 01, ISSN: 0018-9499, 1558-1578. DOI: 10.1109/TNS.2024.3471071.

[82] S Chilingaryan, A Beglarian, A Kopmann, and S Vöcking, "Advanced data extraction infrastructure: Web based system for management of time series data," *Journal of Physics: Conference Series*, vol. 219, no. 4,

p. 042 034, 2010. DOI: 10.1088/1742-6596/219/4/042034. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/219/4/042034.

[83] S. D. Guida, G. Govi, M. Ojeda, A. Pfeiffer, R Sipos, and on behalf of the ATLAS Collaboration, "The cms condition database system," *Journal of Physics: Conference Series*, vol. 664, no. 4, p. 042 024, 2015. DOI: 10.1088/1742-6596/664/4/042024. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/664/4/042024.

[84] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, 981–992, ISBN: 9781605581026. DOI: 10.1145/1376616.1376713. [Online]. Available: https://doi.org/10.1145/1376616.1376713.

[85] Transaction Processing Performance Council, *TPC Benchmark C (TPC-C)*, http://www.tpc.org/tpcc/, 2010.

[86] Transaction Processing Performance Council, *TPC Benchmark DS (Decision Support)*, http://www.tpc.org/tpcds/, 2017.

[87] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proc. VLDB Endow.*, vol. 7, no. 4, 277–288, 2013, ISSN: 2150-8097. DOI: 10.14778/2732240.2732246. [Online]. Available: https://doi.org/10.14778/2732240.2732246.

[88] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, 143–154, ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. [Online]. Available: https://doi.org/10.1145/1807128.1807152.

[89] Timescale, *Tsbs*, online, 2022. [Online]. Available: `https://github.com/timescale/tsbs`.

[90] R. Liu and J. Yuan, *Benchmarking time series databases with iotdb-benchmark for iot scenarios*, 2019. arXiv: `1901.08304 [cs.DB]`.

[91] TSDBBench, *Tsdbbench*, online, 2022. [Online]. Available: `https://tsdbbench.github.io/`.

[92] Y. Hao, X. Qin, Y. Chen, *et al.*, "Ts-benchmark: A benchmark for time series databases," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, New York, NY, USA: IEEE, 2021, pp. 588–599. DOI: `10.1109/ICDE51399.2021.00057`.

[93] N. Hennion, *Glances*, online, 2022. [Online]. Available: `https://github.com/nicolargo/glances`.

[94] B. Imasheva, N. Azamat, A. Sidelkovskiy, and A. Sidelkovskaya, "The practice of moving to big data on the case of the nosql database, clickhouse," in *Optimization of Complex Systems: Theory, Models, Algorithms and Applications*, H. A. Le Thi, H. M. Le, and T. Pham Dinh, Eds., Cham: Springer International Publishing, 2020, pp. 820–828, ISBN: 978-3-030-21803-4.

[95] InfluxData, *Influxdb time series platform | influxdata*, online, 2022. [Online]. Available: `https://www.influxdata.com/products/influxdb/`.

[96] PostgreSQL, *Postgresql*, online, 2022. [Online]. Available: `https://www.postgresql.org/`.

[97] Timescale, *Time-series data simplified | timescale*, online, 2022. [Online]. Available: `https://www.timescale.com/`.

[98] W. Zhang, Y. Xu, Y. Li, and D. Li, "Improving write performance of lsmt-based key-value store," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, New York, NY, USA: IEEE, 2016, pp. 553–560. DOI: `10.1109/ICPADS.2016.0079`.

[99] T. Pelkonen, S. Franklin, J. Teller, *et al.*, "Gorilla: A fast, scalable, in-memory time series database," *Proc. VLDB Endow.*, vol. 8, no. 12, 1816–1827, Aug. 2015, ISSN: 2150-8097. DOI: 10.14778/2824032.2824078. [Online]. Available: https://doi.org/10.14778/2824032.2824078.

[100] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Softw. Pract. Exper.*, vol. 40, no. 2, 131–147, 2010, ISSN: 0038-0644.

[101] Google, *Snappy*, online, 2022. [Online]. Available: https://google.github.io/snappy/.

[102] G. Smith, *Pgtune*, online, 2010. [Online]. Available: https://git.postgresql.org/gitweb/?p=pgtune.git.

[103] J. Sompolski, M. Zukowski, and P. Boncz, "Vectorization vs. compilation in query execution," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, ser. DaMoN '11, Athens, Greece: Association for Computing Machinery, 2011, 33–40, ISBN: 9781450306584. DOI: 10.1145/1995441.1995446. [Online]. Available: https://doi.org/10.1145/1995441.1995446.

[104] Isovalent, *Cni performance benchmark*, online, 2025. [Online]. Available: https://docs.cilium.io/en/stable/operations/performance/benchmark/.

List of Figures

List of Tables

Listings