SDN based Layered Backhaul Optimization and Hardware Acceleration

by

Prateek Shantharama

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved Januvary 2022 by the
Graduate Supervisory Committee:

Martin Reisslein, Chair
Yanchao Zhang
Michael McGarry
Akhilesh Thyagaturu

ARIZONA STATE UNIVERSITY

May 2022

ABSTRACT

Existing radio access networks (RANs) allow only for very limited sharing of the communication and computation resources among wireless operators and heterogeneous wireless technologies. The introduced LayBack architecture facilitates communication and computation resource sharing among different wireless operators and technologies. LayBack organizes the RAN communication and multiaccess edge computing (MEC) resources into layers, including a devices layer, a radio node (enhanced Node B and access point) layer, and a gateway layer. The layback optimization study addresses the problem of how a central SDN orchestrator can flexibly share the total backhaul capacity of the various wireless operators among their gateways and radio nodes (e.g., LTE enhanced Node Bs or Wi-Fi access points). In order to facilitate flexible network service virtualization and migration, network functions (NFs) are increasingly executed by software modules as so-called "softwarized NFs" on General-Purpose Computing (GPC) platforms and infrastructures. GPC platforms are not specifically designed to efficiently execute NFs with their typically intense Input/Output (I/O) demands. Recently, numerous hardware-based accelerations have been developed to augment GPC platforms and infrastructures, e.g., the central processing unit (CPU) and memory, to efficiently execute NFs. The computing capabilities of client devices are continuously increasing; at the same time, demands for ultra-low latency (ULL) services are increasing. These ULL services can be provided by migrating some micro-service container computations from the cloud and multi-access edge computing (MEC) to the client devices.

DEDICATION

To my Pappa & Mummy

For thier advice, thier patince and their Faith

beacuse they always understood

ACKNOWLEDGMENTS

I would first like to thank my supervisor, Prof. Martin Reisslein, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. Second, I would like to thank my committee members, Prof. Yanchao Zhang and Prof. Michael P McGarry for all your support, feedback, and guidance throughout my PhD. I would also like to give special thanks to adjunct faculty ”The Expert” Dr. Akhilesh Thyagaturu for his continuous support and understanding when undertaking my research.

To my lab-mates Ahmed Nasrallah, Venkat and Ziyad, thank you for your wonderful patience, continual support and warm humor.

I am forever indebted to my family members Suma, Tharakeshwar, Sunil, Pavana, Shoba, Shivanna, Swetha, Dr. Harshith, my grandparents Hanumanthappa and Rathna, my late grandparents Dasappa and Puttamma and my little brothers Pranith and Pratham for giving me the opportunities and experiences that have made me who I am. They selflessly encouraged me to explore new directions in life and seek my own destiny. This journey would not have been possible if not for them, and I dedicate this milestone to them.

To Vinya, Kottur, Krishna, UD, Amogh, Ashra, Sameeksha, Karan and Vivek: thank you for the pep-talks, cups of tea, hugs, sound advice, encouragement and kicks-up-the-backside. All administered with impeccable judgment and timing.

I would also like to give special thanks to Mr.Mohan Kumar who was always a silent shadow through out my PhD Degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

PREFACE

Chapter 2 of this thesis has been published as Wang, Mu, et al. "A multi-layer multi-timescale network utility maximization framework for the SDN-based LayBack architecture enabling wireless backhaul resource sharing." Electronics 8.9 (2019): 937. I was responsible for the data collection and analysis. Mu Wang, Nurullah Karakoc, Lorenzo Ferrari, Akhilesh S. Thyagaturu, Martin Reisslein & Anna Scaglione assisted with the data collection and contributed to manuscript edits, supervisory authors and was involved with concept formation and manuscript composition.

Chapter 4 of this thesis has been published as Shantharama, Prateek, et al. "Hardware acceleration for container migration on resource-constrained platforms." IEEE Access 8 (2020): 175070-175085. I, Prof.Martin & Akhilesh Thyagaturu was responsible for the data collection and analysis as well as the manuscript composition. Anil Yatavelli, Poornima Lalwaney, Georgii Tkachuk & Edward J Pullin assisted with the data collection and contributed to manuscript edits, supervisory authors and was involved with concept formation and manuscript composition.

Chapter 1

LAYERED BACKHAUL

1.1 Introduction

Wireless access networks have emerged as a critical bottleneck in Internet access. One of the root causes of the access bottleneck is that each wireless service provider (operator) and each wireless technology (such as LTE or WiFi) operates typically in an operator/technology-specific “silo”. That is, each operator/technology has its own radio access network (RAN) chain consisting of the RAN Pateromichelakis *et al.* (2017) and the corresponding backhaul network Jaber *et al.* (2016). For brevity, we refer to the entire RAN chain as RANC. While there have been some efforts in wireless standards Liu *et al.* (2016b) and in academic research to share network resources across wireless technologies, the solutions available to date provide very limited flexibility (see Section 1.2.1). Thus, there is only very limited statistical multiplexing (sharing) of network resources among wireless operators and technologies Niu *et al.* (2016). The *status quo* is, to a large degree, due to the lack of a convenient effective signaling infrastructure across the wireless access networks. To provide this missing signaling infrastructure, in this paper we propose a novel SDN-based architecture: the Layered Backhaul (LayBack) architecture. Our contributions are summarized next.

1.1.1 Contributions

The Layered Backhaul (LayBack) architecture, which is illustrated in Fig. 1.1, addresses the wireless access bottleneck by judiciously employing the existing RAN and multi-access edge computing (MEC) Fan *et al.* (2018) resources under a unifying

Software Defined Networking (SDN) orchestrator Huang *et al.* (2018a); Narmanlioglu *et al.* (2018). We strategically place the SDN orchestrator at the network backhaul behind the gateways of the different wireless access technologies. The centralized SDN orchestrator manages the use of MEC resources distributed across the network to provide network services, such as the RAN services.

We make three main contributions.

1. We introduce the novel LayBack architecture which comprehensively integrates the wireless fronthaul and backhaul of heterogeneous wireless technologies and operators in Section 1.3. LayBack places the *coordination point between the heterogeneous wireless technologies and operators behind the gateways of the respective technologies and operators.* Thus, from this coordination point, an SDN switching network can flexibly interconnect the respective gateways with a unifying SDN orchestrator and the backhaul (core) networks, see Fig. 1.1.

2. We introduce an SDN based management framework for coordinating distributed MEC resources to support network services in Section 1.4. The SDN based management is executed at a unifying SDN orchestrator. The unifying SDN orchestrator performs the inter-layer management and coordination within the LayBack architecture so as to readily utilize the distributed communication and computing resources across heterogeneous technologies and operators.

3. We illustrate the usage of the LayBack architecture and management framework through a quantitative case study on resource sharing in a RAN with multiple operators or technologies in Sections 1.5 and 1.6. The case study considers fluid RAN function splits, where RAN function block computations are dynamically assigned to MEC nodes. The evaluation results indicate that for non-uniform call arrivals, the resource sharing enabled by LayBack can increase the revenue

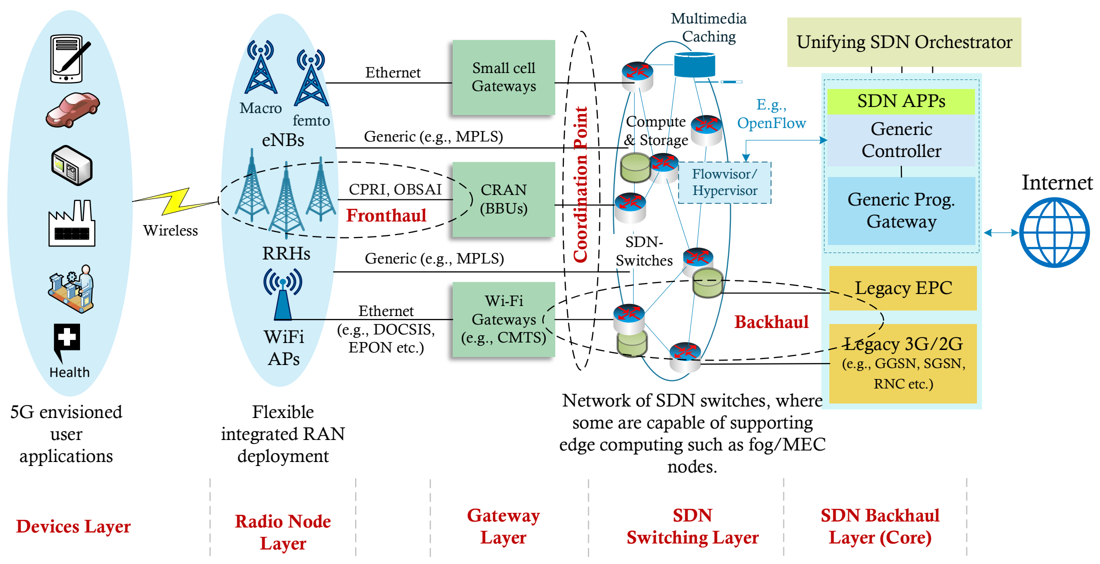


Figure 1.1: Illustration of Proposed Layback Architecture: Layback Flexibly Interfaces with Heterogeneous Radio Access Network (Ran) Technologies Through a Network of Gateways and Sdn Switches. At the "coordination Point" Just Behind (to the Right) of The Respective Gateways, Layback Accesses and Controls the Heterogeneous Rans Through the Sdn Switching Layer. The Sdn Switching Layer Consistently Decouples the Ran Fronthaul from the Backhaul. The Unifying Sdn Orchestrator Integrates the Legacy Backhaul, Existing Architectures, and Future Sdn Architectures. The Sdn Orchestrator Is The Central Authority That Controls Every Part of the Architecture, Including Fronthaul and Backhaul. Multi-access Edge Computing (Mec) Nodes May Be Distributed Throughout The Radio Node, Gateway, Sdn Switching, and Sdn Backhaul Layers.

from completed calls by more than 25%. We have presented another case study that utilizes LayBack for the optimization of communication resource allocations across different operators, gateways, and radio nodes in Ferrari *et al.* (2018a).

1.2 Related Work

1.2.1 RAN Chain (RANC): Fronthaul and Backhaul Architectures

In contrast to clean-slate SDN-based RAN architectures, such as Ameigeiras *et al.* (2015), LayBack flexibly accommodates existing as well as new technologies and deployments. The European project 5G Xhaul has studied a wide range of RANC aspects, including 5G network requirements Bartelt *et al.* (2017) and the benefits of SDN control Gutiérrez *et al.* (2016); Oliva *et al.* (2015). The 5G Xhaul project also investigated aspects of specific frontend radio technologies, such as MIMO Chaudhary *et al.* (2017) and mmWave Huerfano *et al.* (2017), and optical network technologies for the backhaul Tzanakaki *et al.* (2017). Moreover, the slicing (virtualization) of the network has been studied Costanzo *et al.* (2018). Similarly, the European Crosshaul project has considered the RANC combining radio fronthaul and the backhaul Cavaliere *et al.* (2017); Costa-Perez *et al.* (2017a). The Crosshaul project has investigated aspects of the SDN control González *et al.* (2016), as well as the mmWave Ogawa *et al.* (2017) and MIMO Huang *et al.* (2018b) transmissions. In addition, the slicing of the network Li *et al.* (2017) and wired (including optical) transport have been considered Alimi *et al.* (2018). These transport aspects are currently further examined in the European Metrohaul project Casellas *et al.* (2018). Similarly, other research groups have examined slicing in RANCs Richart *et al.* (2016a), as well as the transport solutions for fronthaul Chanclou *et al.* (2018) and backhaul Thyagaturu *et al.* (2016a). LayBack complements the 5G Xhaul and Crosshaul architectures as well as other recently proposed SDN-based RANC architectures, such as CROWD Auroux *et al.* (2015), iJOIN Wang *et al.* (2015a), and U-WN Zhang *et al.* (2014), as well as similar architectures Droste *et al.* (2016); Qadir *et al.* (2014), in that LayBack consistently decouples the wireless radio access (fronthaul) technologies, such as LTE or WiFi, and corresponding gateways from the backhaul access network.

The recently proposed SDN-based architectures generally retain some dependencies or direct interconnections between the fronthaul and the backhaul and thus have limited flexibility to accommodate heterogeneous wireless access technologies and to allow the fronthaul to evolve independently. In contrast, LayBack achieves these flexibilities by moving the management *"coordination point"* between different wireless access technologies *behind the gateways of the respective technologies*, as illustrated in Fig. 2.1 and elaborated in Section 1.3. In brief, LayBack coordinates heterogeneous fronthauls and their respective gateways through central coordination behind the gateways through an SDN switching network that connects to a unifying SDN orchestrator (see Section 1.4). The positioning of the coordination point and the SDN switching layer just behind the respective gateways gives the LayBack SDN orchestrator direct access to the fronthauls and allows for flexible switching between heterogeneous fronthauls and backhauls.

The proposed LayBack architecture is also different from recent SDN tiered control architectures, e.g., three-tiered architectures Elgendi *et al.* (2016), as well as prior research on SDN-based architectures, such as Oliva *et al.* (2015); Tzanakaki *et al.* (2017), through the tight integration of the distributed MEC with the provisioning of network access service.

1.2.2 MEC for RAN Function Splits

A computing infrastructure that is installed in close proximity to the wireless users and radio nodes is referred to as multi-access edge computing or mobile-edge computing (MEC) Wang *et al.* (2017c). The MEC mechanism by Wang et al. Wang *et al.* (2017a) jointly performs user-computation offloading and radio node physical resource block (PRB) allocation (as a mechanism to manage wireless interference). Similar MEC mechanisms that jointly optimize user computations and wireless resources have been examined in Luong *et al.* (2018).

Building on this prior work, we assume that computing nodes are distributed across the network. The emerging challenge is to coordinate and manage the distributed computing and network services for increasing numbers of nodesAkhilesh Thyagaturu (2021). Advanced management mechanisms are necessary, such as distributed agent-based edge computing Bumgardner *et al.* (2016) and computational resource management Mao *et al.* (2017). To the best of our knowledge, the existing fog and MEC resource management studies have been limited to the offloading of user application computations and computations for specific individual function blocks (steps) involved in wireless physical layer transmissions e.g., interference management. Complementary to the existing mechanisms, we propose a uniform framework to comprehensively manage the computations for the full range of steps involved in providing RAN service.

The function split in RANs between the radio nodes, also referred to as remote radio heads (RRHs) or remote radio units, and the base band units (BBUs) has been investigated in several recent studies, including Checko *et al.* (2015); Chih-Lin (2017); Garcia-Saavedra *et al.* (2018); Thyagaturu *et al.* (2018a). Our fluid RAN function split in Sections 1.5 and 1.6 fundamentally differs from prior work in that we generalize the RAN computations to be performed via flexible function chaining Leivadeas *et al.* (2017) on distributed MEC nodes. The RAN computations are coordinated on demand through SDN control. A traditional cloud RAN (CRAN) provides the computing in a centralized manner. On the other hand, the emerging Next Generation Fronthaul Interface (NGFI) sagroups (1914); Chih-Lin *et al.* (2018a) architecture allows for the *static* assignment of the RAN computation tasks to two specific MEC nodes, namely a Digital Unit (DU) and a Central Unit (CU), and to complete the remaining computations at the BBU. In contrast, our SDN controlled fluid RAN approach *flexibly* assigns RAN computations tasks to an arbitrary number of MEC nodes.

1.3 Proposed LayBack Network Architecture

LayBack is enabled by recent advances in software defined networking (SDN) Ramirez-Perez and Ramos (2016). LayBack breaks down the boundaries separating different wireless technologies by providing a unifying SDN-based signalling infrastructure. As shown in Fig. 2.1, by bringing all wireless access technologies (and corresponding operators that are willing to share their available resources and dynamic reconfiguration policies) under the umbrella of a unifying SDN orchestrator (top right of Fig. 2.1), LayBack achieves (*i*) the benefits of the individual wireless technologies, and (*ii*) the benefits that can be reaped through the coexistence and cooperation of multiple wireless technologies and operators.

LayBack complements and augments the potential of the popular Cloud RAN (CRAN) abstraction, because the backhaul is the point of convergence of Internet traffic and therefore is the ideal point to orchestrate the cooperative management of different wireless Internet technologies.

In traditional network infrastructures, network functions are tightly coupled with the network elements, such as the gateways, and the network elements are therefore commonly referred to as “communication nodes”. In contrast in emerging MEC based infrastructures, network functions are implemented as virtualized entities on generic computing resources; hence the network elements are often referred to as “computing nodes”. The LayBack architecture homogeneously considers both existing traditional and newly emerging network infrastructure deployments; thus we refer to the network elements generally as “nodes”. The computing capabilities in the communication nodes in existing infrastructures can be enabled by augmenting the communication nodes with MEC nodes.

1.3.1 The Layers of the LayBack Architecture

We proceed to describe the key components and functionalities of the proposed LayBack architecture in more detail. We note that MEC nodes permeate all layers from the radio node layer to the SDN backhaul layer in Fig. 2.1.

Wireless End Devices Layer

Mobile wireless end devices are heterogeneous and have a wide range of requirements. Providing reasonable quality-oriented services to every device is a key challenge of wireless network design. Future devices that are part of the so called Internet of Things (IoT)Balasubramanian *et al.* (2021a); Guck *et al.* (2016), will likely be highly application-specific, such as health monitoring biosensors Rodrigues *et al.* (2018). Visions for 5G wireless systemsHoeschele *et al.* (2021) foresee that a user can request network services and applications independently of the wireless technology, i.e., physical aspects of the network connectivity, wireless protocols, and physical infrastructures of the core networks. As no single wireless technology can serve all purposes, we believe that it will be vital to provide a unifying network architecture and management framework so as to flexibly and efficiently provide wireless services.

Radio Nodes Layer

Radio nodes, such as the evolved NodeB (eNB) in LTE or an access point (AP) in WiFi, provide RAN services to the end devices. Aside from LTE and WiFi, there exists a wide range of wireless access technologies (and protocols), including Wi-MAX, Zig-Bee, Bluetooth, and near field communication (NFC) Kim *et al.* (2017). These wireless technologies have unique advantages and serve unique purposes; therefore, a fluidly flexible radio node that seamlessly supports a diverse range of wireless protocols is desired Sundaresan *et al.* (2016).

RANs are not only heterogeneous in the wireless access technologies, RAN operational and deployment aspects are also highly operator specific. RAN technology advancements in the area of CRAN Checko *et al.* (2015) have pushed the limits of scalability and flexibility through leveraging SDN and NFV concepts Leivadeas *et al.* (2017). As a result of the wide range of network applications, which may be specific to operators and network architectures, the operation of the radio nodes layer is highly complex. Through our proposed LayBack architecture we can bring transparency to the network, easing the transitions among multiple heterogeneous RANs.

Gateway Layer

The gateway layer encompasses the network entities between the radio node layer and the SDN switching layer in Fig. 2.1. A CRAN consists of a BBU gateway that collectively processes the basebands of several RRHs, which in turn may simultaneously support multiple wireless technologies. Radio nodes operating in a non-CRAN environment, such as non-CRAN macro cell eNBs, process the baseband locally and connect directly to the core (backhaul layer) network gateways via the SDN switching layer. Similarly, WiFi APs at residential sites typically connect to a cable or DSL modem, eventually connecting to a cable modem transmission system (CMTS) or customer premise equipment (CPE) gateway. Interactions between the gateways can be enabled by extending the gateway functions to support SDN actions, under the control of a unifying SDN orchestrator.

SDN Switching Layer

SDN switches are capable of a wide range of functions, such as forwarding a packet to any port, duplicating a packet on multiple ports, modifying the content inside a packet, or dropping the packet Huang *et al.* (2018a); Narmanlioglu *et al.* (2018). The LayBack architecture homogeneously accommodates different technologies embedded in the networking switching elements. For example, a group of users who are connected to different operators, such as WiFi and LTE, can request a common content delivery service. In such a scenario, by supporting caching, the switching network elements can enable the content caching mechanism Zhao *et al.* (2016) serving uniformly all the users, irrespective of their wireless connectivity (i.e., LTE or WiFi) and gateway layers. The LayBack SDN switching layer directly connects to the gateways of the respective RAN technologies and operators and thus effectively provides a "coordination point" to control all RANs. At the same time, the SDN switching layer decouples the RANs (fronthaul) from the backhaul.

SDN Backhaul (Core) Network Layer

The backhaul (core) network layer comprises technology-specific network elements, such as the Evolved Packet Core (EPC) which supports the connectivity of LTE eNBs. Similarly, for 2G/3G legacy cellular architectures, the core network includes networking elements, such as a Gateway GPRS Support Node (GGSN) and a Radio Network Controller (RNC). We define a generic programmable gateway and the SDN controller to represent all the SDN-based core network architectures, such as iJOIN and xHAUL Tzanakaki *et al.* (2017). The generic SDN controller abstracts the underlying design of the data plane and control plane specific to the architecture. The unifying SDN orchestrator extends the SDN functions to the core network elements that are not native to SDN, such as EPC and SGSN, so as to dynamically reconfigure the core network. Communication between multiple core network elements can implement the multi-operator network sharing mechanisms as well as user mobility, e.g., handover, across multiple technologies.

Unifying SDN Orchestrator

The unifying SDN orchestrator plays an important role in creating a common platform for all the heterogeneous network technologies and operators (which can be viewed as heterogeneous network domains) across all the layers in the LayBack architecture. Although we view the SDN orchestrator as a single entity, actual orchestrator deployments can consist of multiple SDN controllers that are hierarchically organized to form a single virtual orchestrator. The unifying SDN orchestrator maintains the current topology information of the entire network and tracks the network capabilities by exchanging messages with the network elements. Network elements can either be physical entities or virtual entities obtained through NFV or network service chaining Leivadeas *et al.* (2017).

The unifying SDN orchestrator has access to all the LayBack layers to flexibly reconfigure the network. Through the central SDN orchestrator control, existing and future architectures can be flexibly integrated to achieve seamless resource sharing and mobility of users (devices) across multiple technologies. Networks maintained by different operators need to communicate their requirements and reconfiguration capabilities to the SDN orchestrator. An operator may choose not to advertise its capabilities or can selectively share capabilities based on real-time statistics, such as resource availability.

1.4 SDN Based Management of Distributed Computing for a Network Service

This section introduces a management framework and the management processes to fulfill the computing requirements in a decentralized manner by dynamically reconfiguring the network based on SDN. In traditional cloud computing based networking, the computing requirements for a given user's network service are addressed

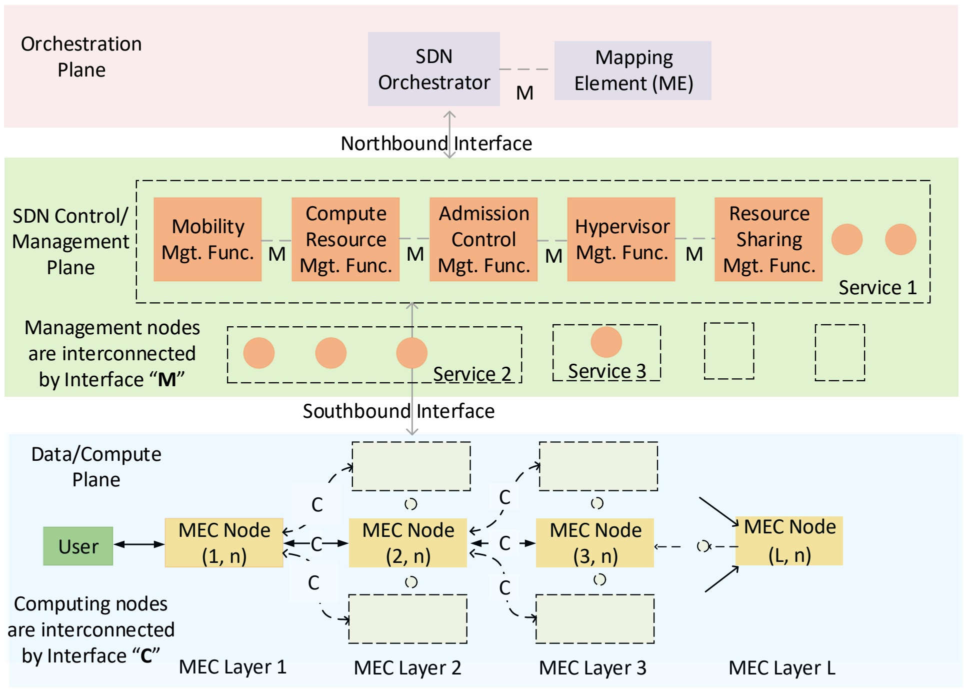


Figure 1.2: Management Framework for Sdn Based Distributed Computing: The Orchestration Plane Coordinates the Overall Service Provisioning Through Instantiating Control/Management Vms on The Control/Management Plane. The Management Plane in Turn Controls The Data/Compute Plane.

in a centralized manner. Our approach not only decentralizes the computing, but also collectively delivers the distributed computing as an aggregated network service to the users.

1.4.1 *Management Framework Planes and Interfaces*

We introduce the management framework planes and interfaces illustrated in Fig. 1.2 for managing the provisioning of services with the LayBack architecture. In particular, we introduce from bottom to top, the data/compute plane, the control/management plane, and the orchestration plane. These planes interface with the conventional southbound and northbound interfaces of SDN. We introduce

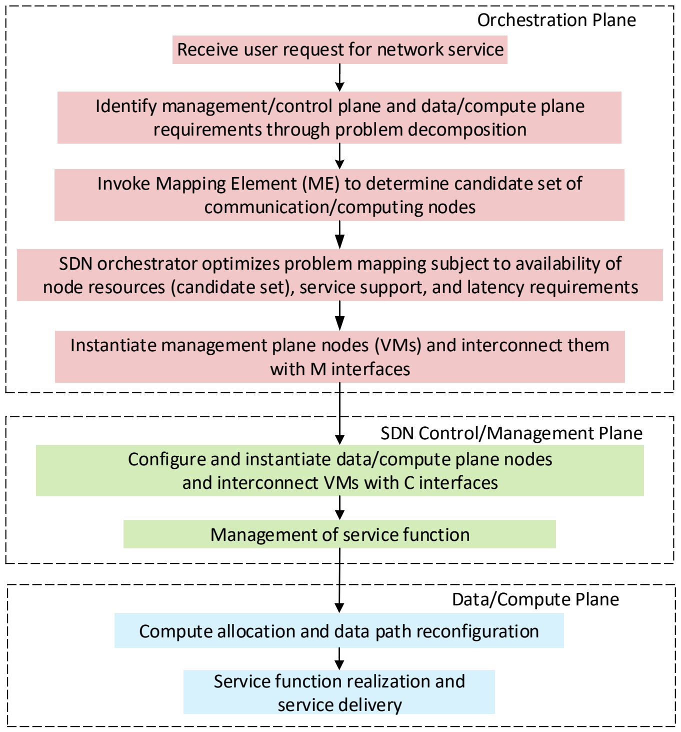

Orchestration Plane
Receive user request for network service
Identify management/control plane and data/compute plane requirements through problem decomposition
Invoke Mapping Element (ME) to determine candidate set of communication/computing nodes
SDN orchestrator optimizes problem mapping subject to availability of node resources (candidate set), service support, and latency requirements
Instantiate management plane nodes (VMs) and interconnect them with M interfaces
SDN Control/Management Plane
Configure and instantiate data/compute plane nodes and interconnect VMs with C interfaces
Management of service function
Data/Compute Plane
Compute allocation and data path reconfiguration
Service function realization and service delivery

Figure 1.3: Flow Chart for Sdn Based Management: Upon Receiving A User Request, the Sdn Orchestrator Coordinates with the Mapping Element (Me) in the Orchestration Plane. The Sdn Orchestrator Decomposes the Problem and Provisions the Network Connectivity Among the Management Nodes in the Sdn Control/Management Plane To Enable the Control and Management of the Requested Service. The Management Nodes Then in Turn Provision the Data/Compute Plane Nodes and Their Interconnections in the Data/Compute Plane for The Service Delivery. Overall, the Sdn Orchestrator Is Responsible For Provisioning the Management Functions in Order to Achieve The End-to-end Delivery of Network Services.

a management (M) interface for the interactions of the orchestration and control/management plane entities as well as a compute (C) interface for the interactions of the data/compute plane entities.

Data/Compute Plane

The data/compute plane consists of all the SDN controlled communication and computing nodes that can be reconfigured by a logical control plane. A computing node can belong to any of the LayBack architecture layers (see Fig. 2.1), i.e., the RAN, gateway, switching, and core network layers.

Control/Management Plane

The control/management plane is a logical entity that is instantiated by the SDN orchestrator. More specifically, the control plane is a collection of all the management functions corresponding to the network services hosted in the data/compute plane. Essentially, the control/management plane is implemented as VMs on the MEC nodes, whereby the SDN orchestrator instantiates the management nodes, such as the SDN controller specific to a network service requested by the user. Once the control plane is provisioned, the control/management nodes (VMs) are responsible for the run-time management of the network services.

SDN Orchestration Plane

The SDN orchestration plane consisting of the SDN orchestrator and the mapping element (ME, introduced in Section 1.4.2) is the logically centralized high level decision entity. In particular, the network grid is typically heterogeneous, comprising of several domains, such as, different operator and technology domains. In LayBack, the SDN orchestrator unifies these heterogeneous domains by centralizing the control decisions. The SDN orchestrator instantiates, implements, monitors, and tears down the management nodes (VMs) for the network services at the requests of users.

For inter-operator management, an operator can hide the deployment characteristics, selectively expose the deployment characteristics, or present a abstracted (virtualized) infrastructure to the centrally managed orchestrator. The SDN orchestrator then acts as the coordination point for the interaction of different network services, such as the multi-operator network sharing.

Interfaces

The interactions between the various planes of the management framework and the entities within a given plane occur across pre-defined interfaces. To reduce the overhead and to ensure consistency with the general SDN management framework, conventional SDN interfaces are used for the interactions between the planes of the LayBack management framework. In particular, the interactions between the control/management plane and the data/compute plane can be supported by a conventional southbound interface, such as OpenFlow. Similarly, the interactions between the orchestration plane and the control/management plane can be supported by a conventional northbound interface, such as the representational state transfer (REST).

We introduce the management M interface for the interactions between the individual entities in the orchestration and control/management planes. Furthermore, we introduce the compute C interface for the interactions between the compute nodes in the data/compute plane. The M and C interfaces are general interface constructs that flexibly allow particular protocol interfaces to be incorporated within the general M and C interface constructs. For instance, the X2 interface (for eNB to eNB connections in LTE) or the N interface (for interconnections of network functions in the 5G backhaul) can be incorporated within the general M interface as needed to fulfill user requests. On the hand, the S1-U interface (between eNB and S-GW in the LTE backhaul) can be incorporated into the C interface in the data/compute plane.

1.4.2 Orchestration Layer Processing

Adapting SDN principles Huang *et al.* (2018a); Narmanlioglu *et al.* (2018), we centralize the decision making involved in the service provisioning at the SDN orchestrator. In particular, the orchestration plane coordinates the service provisioning by executing the steps illustrated in Fig. 1.3 for each service request.

User Request for Network Service

We define a network service as a user desired network application that enables the user to interact with a remote client (cloud service) or other end-users. For instance, a network service in the 5G context could include enhanced Mobile BroadBand (eMBB), Ultra Reliable and Low Latency (URLL) communications, or a massive mobile Internet of Things (IoT). In addition to specific applications, such as eMBB, URLL, and IoT, LayBack can also support the entire 5G framework as a network service. Thus, LayBack may provide specific network applications, such as the eMBB, URLL, and IoT, as a network service either within the framework of 5G connectivity or as independent services.

Generally, we refer to the node desiring to offload a communication or computation task arising from a service request partially or entirely to the network grid as a “user”. We note that the “users” are not only the end devices, but could also include the communication and computing nodes themselves, such as, radio nodes. A user sends the request corresponding to a network service to the network grid. The network grid forwards the request to the logically centralized SDN orchestrator. Criticality aspects of the service, such as latency and reliability requirements, are either reported or estimated based on the request type.

Problem Decomposition from Original Problem to Sub-Problems

For the purpose of management, we collectively refer to a network service or a network application as the "original problem", or simply as the "problem". Problem decomposition refers to the transformation of complex original problems into simpler constituent sub-problems preserving the problem integrity. Problem decomposition requires the consideration of the localization properties of the problem as well as the problem structure.

Many network service applications involve only a finite localized set of nodes, i.e., have specific localization properties. For instance, only the co-located radio nodes are responsible for interference coordination. Similarly, the sharing of uplink transmissions over a limited backhaul link requires the coordination of all connected users. Accordingly, for the efficient provisioning of communication and computing resources for different applications, the SDN orchestrator should consider the different sets of nodes that are co-located within a prescribed region when provisioning network services.

Depending on the problem structure, the solution of the original problem may require coordination among the sub-problems. Such coordination can be provided by a root-problem. A root-problem is a special sub-problem that is executed on a locally centralized entity that has connectivity to all the end users involved in the original problem (i.e., network application or service). In addition, the root-problem has connectivity to all other computing entities that solve sub-problems or are involved in the decision making processes. The root-problem and the individual sub-problems mutually exchange information for solving the original problem.

Mapping Element: Determining Candidate Communication/Computing Nodes Set

An important factor to consider during the problem decomposition is the availability status of the communication and computing resources. Computing entities that are part of the networking grid can simultaneously execute multiple sub-problems, in addition to their respective network functions, such as switching and forwarding. Therefore, a computing entity experiences dynamic loading based on the user requests and the current state of the network. The candidate set evaluation of the nodes needs to consider the availability of the nodes, the support for computations, the dynamic loading, the vicinity to the user, and the support for required networking services. As this involves a complex evaluation process, we propose a dedicated network Mapping Element (ME) to evaluate the candidate set of nodes for a given service request. The ME maintains and regularly updates the current states of the network nodes In particular, each node that supports communication/computing services periodically reports its utilization statistics to the ME. The ME considers the latest utilization statistics for evaluating the candidate set of nodes for a user request.

Optimize Problem Mapping

The SDN orchestrator employs the candidate set provided by the ME to optimize the mapping of the sub-problems (obtained from the problem decomposition) to the communication/computing nodes. More specifically, the SDN orchestrator optimizes the problem mapping subject to the node resource availability (i.e., the candidate set from the ME), the service support at the various candidate nodes, and the latency requirements.

As part of the optimization of the mapping to communication/compute nodes, the

SDN orchestrator optimizes the mapping of communication services with prescribed quality of service (QoS) or quality of experience (QoE) requirements to the available access network technologies. In this communication optimization, the SDN orchestrator considers the characteristics of the different access network technologies, e.g., the different radio propagation characteristics. The specific optimization mechanisms to employ within the LayBack management framework are beyond the scope of this article and are an important direction for future research. For an initial study on optimizing communication resource allocations in the LayBack context, we refer to Ferrari *et al.* (2018a); Karakoç *et al.* (2020, 2022); Balasubramanian *et al.* (2021b).

Instantiate Control/Management Plane Nodes

As a final step in its support of service provisioning, the SDN orchestrator instantiates the control/management plane nodes as VMs and interconnects the instantiated VMs with M interfaces through reconfigurable SDN switching. Alternatively, the SDN orchestrator assigns the control/management functions to existing VMs that support the required functions and have sufficient available capacity.

1.4.3 Control/Management Plane Processing

Instantiate Data/Compute Plane Configuration

The control/management VMs (that were instantiated by the SDN orchestrator, see Section 1.4.2) configure the data/compute plane to instantiate and to interconnect the communication/compute nodes. More specifically, analogously to forwarding rules in an SDN switch, computing rules can be installed on the computing nodes. Each computing node is configured to process the requests if a rule pertaining to the request exists on the computing node, else the requests can be ignored, denied, or forwarded to the SDN orchestrator. Computing rules can be assigned with expiry timeout based on the idle status of the nodes. For the typical VM based computing services, the control/managment VMs control the instantiation, migration, and tear down of data/compute plane VMs.

Moreover, the control/management VMs configure the network grid to establish the communication paths that interconnect the data/compute plane VMs. In addition, auxiliary network control functions, such as redundancy provisioning for reliability and load balancing, are conducted by the control/management VMs.

Maintain Service Functions

Once the data/compute plane service has been instantiated, the control/managment plane maintains the service. As part of the service maintenance, the control/management plane monitors and ensures the end-to-end QoS, and preserves the service integrity in case of disruptions or network changes through recovery operations.

1.4.4 Data/Compute Plane Processing: Service Delivery

Overall, the end-to-end service is provided through the coordinated allocation of the sub-problem communication/computation tasks to the data/compute plane nodes; whereby the data/compute plane nodes are configured by the control/management VMs. The data/compute plane nodes intercommunicate through the data paths configured via C interfaces by the control/management VMs. The coordinated sub-problem communication/computation actions of the data/compute plane nodes provide the overall networking services to the users.

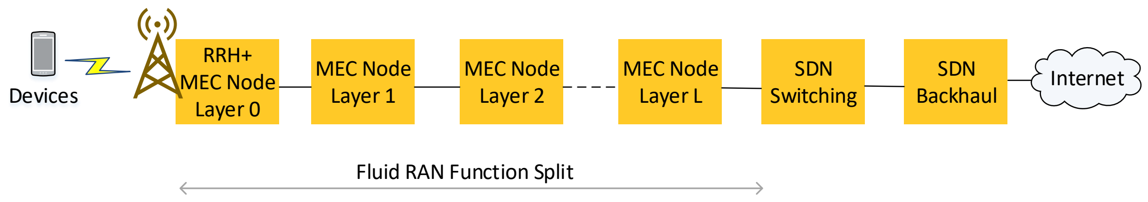


Figure 1.4: Illustration of Proposed Fluid Ran Which Dynamically And Flexibly Distributes Ran Compute Function Blocks Across Multiple Mec Nodes. The Function Blocks Are Chained to Operate in Cohesion To Achieve Common Function Goal, I.E., Provide the Ran Service. The Mec Node Layers $l = 0, 1, 2, \ldots, L$ Are Assumed to Exist Across The Radio Node Layer, the Fronthaul Network, and the Gateway Layer In The Overall Layback Architecture.

1.5 LayBack Use Case: Novel Fluid RAN Function Split with Resource Sharing across Operators

The purpose of this section and the subsequent Section 1.6 is to illustrate the use of the LayBack architecture and management framework for an exemplary use case. The exemplary use case is the provisioning of a network service through the management of distributed MEC nodes; specifically, the provisioning of a RAN service. We illustrate how the computing tasks for the RAN service can be distributed over MEC nodes and multiple operators. The distribution of the RAN service computing tasks is enabled through the management framework introduced in Section 1.4, which operates within the LayBack architecture introduced in Section 1.3.

1.5.1 Background on Existing RANs

In a CRAN, an RRH is the radio frequency (RF) processing entity which is typically implemented as a part of the RF transmission antennas of cellular radio access technologies. On the other hand, the BBU performs the baseband processing. A fronthaul network interconnects the RRHs and BBUs. BBUs are softwarized entities that are typically implemented as VMs on general purpose computing entities, such as micro and macro data centers. SDN and NFV technologies can compose virtualized BBU functions through the chaining of virtualized network service functions Medhat *et al.* (2017). To date, network virtualization and service chaining have been mainly applied only to the BBU functions in CRANs and to the backhaul (from BBUs toward the Internet). In contrast, we pursue network virtualization and service chaining for the RRH functions and the fronthaul (from RRH to BBU). We examine the spreading of the RRH functions across multiple layers of the LayBack architecture, while flexibly chaining function blocks together to compose efficient fronthaul links. Thus, we effectively study the extension of the benefits of VMs, NFV, and function chaining to the fronthaul.

The recently introduced generalized Next Generation Fronthaul Interface (NGFI, IEEE P1914.1) sagroups (1914); Chih-Lin *et al.* (2018a) architecture allows for the *static* functional split assignments of RAN computation tasks to the RRH and BBU as well as two intermediate nodes, namely a Digital Unit (DU) and a Central Unit (CU). Based on the LayBack architecture and SDN based centralized management of computing for a network service, we propose a fluid RAN function split. The fluid RAN function split dynamically and flexibly assigns RAN computation tasks to arbitrary MEC nodes.

1.5.2 Proposed Concept of Fluid Function Blocks

Each function block in the NGFI fronthaul and CRAN architecture is essentially a computing entity, that transforms the incoming data to a form that is suitable for processing in the subsequent computing entity. Each computing entity may belong to a part of the radio protocol layer operations, such as PHY or MAC of LTE. In the proposed fluid function split, the CRAN function problem is partitioned into multiple sub-problems (function blocks), without a prescribed arbitrary limitation of the number of function blocks. The function blocks can be dynamically created and assigned to the computing entities, which are interconnected through Ethernet or time sensitive networking (TSN) based networks. This process not only provides a high degree of flexibility, but also facilitates new schemes for infrastructure resource utilization. NGFI limits the fronthaul function blocks to be statically split (assigned) to only two computing entities, namely the DU and the CU, in addition to the RRH and BBU. In contrast, our proposed LayBack architecture provides a unique platform for the centralized management of distributed computing so as to extend the existing fixed fronthaul and backhaul architecture to a distributed computing framework. That is, the function blocks can be flexibly assigned to distributed MEC nodes without an arbitrary limitation on the number of utilized MEC nodes.

The fluid RAN function block assignment can be implemented through software entities, i.e., VMs, on generic computing entities. The generalized computing entities are MEC nodes distributed throughout the radio node, gateway, SDN switching layers in the LayBack architecture in Fig. **??**. Existing advanced VM management methods for inter and intra data center networks Bari *et al.* (2013) can be applied for the VM duplication, setup, tear down, and migration to other nodes.

1.5.3 Proposed LayBack Implementation of Fluid Function Split

The fundamental principle of the LayBack architecture is to unify the wide variety of heterogeneous infrastructures that exist due to different operators and technologies. LayBack categorizes these heterogeneous infrastructures in terms of layers, and interconnects them through a configurable network, i.e., the SDN switching layer, see Fig. ??. In the LayBack architecture, the RRH is located at the radio node layer, which requests services through the fluid function split paradigm. A given RRH may require different fronthaul services due to changing RRH characteristics, such as varying numbers of connected users, varying bandwidth demands, or varying power requirements. For each change in the RRH characteristics, there may be a corresponding change in the interconnecting fronthaul link requirements, and the function block implementations to complete the RAN processing.

We employ the centralized management of distributing computing, as introduced in Section 1.4, to meet the computing requirements for the RAN functions. More specifically, the LayBack SDN unifying orchestrator implements the SDN based management framework illustrated in Fig. 1.2 to assign the function blocks to MEC nodes and to configure the fronthaul network to maximize the overall utilization while seamlessly maintaining continuous service.

1.5.4 System Model

RAN Network

As summarized in Table 1.1, we denote N for the number of parallel CRAN systems, e.g., the number of service providers that operate a CRAN in a given area. For simplicity, we assume that each of the N parallel CRAN systems has $L+1$ layers of MEC nodes. We denote $Z_{l,n}$ for the computation capacity at the MEC node in layer l, $0 \leq l \leq L$, of CRAN n, $1 \leq n \leq N$. We model the communication capacity of the reconfigurable SDN network interconnecting the MEC nodes as follows. The MEC nodes within a given layer l are interconnected with a shared intra-layer communication capacity C_l [bit/s]. The successive MEC layers l and $l+1$ are interconnected by a shared inter-layer communication capacity $C_{l;l+1}$ [bit/s].

Data Call

We define r as the payload data bitrate [in bit/s] of a given data call (stream) and let τ denote the expected (mean) call duration [in seconds]. That is, r corresponds to the user payload data rate, which we consider to be effectively the bitrate at the IP datagram level. We consider low, medium, and high user payload data rates denoted by r_{low}, r_{med}, and r_{high}. We consider independent data call generation according to a Poisson process with prescribed rate λ [data calls/s] for each of the N CRAN systems, i.e., the total call arrival rate to the N parallel CRANs is $N\lambda$.

Function Blocks

We define the function $\mathcal{F}$ to represent the complete set of of fronthaul and baseband computations for a given data call in a CRAN system. Analogous to the series expansion of any bounded function, such as the Fourier and Taylor series expansion, the CRAN function $\mathcal{F}$ can be represented in terms of function blocks as $\mathcal{F} = \sum_{b=0}^{B} f_b$, where $B + 1$ is the total number of function blocks for a given CRAN system.

In our model, a given data call (stream) has to complete the function blocks (computation tasks) $f_b,\ b = 0, 1, 2, \ldots, B$, with corresponding computation requirements (demands, loads) $\beta_b,\ b = 1, 2, \ldots, B$. The function block f_0 computation has to be performed at layer $l = 0$. All other function block computations $f_b,\ b = 1, 2, \ldots, B$, can be flexibly (fluidly) performed at any of the layers $l = 1, 2, \ldots, L$.

Note that in our model, a conventional fully distributed RAN performs the function blocks $f_b,\ b = 1, 2, \ldots, B$, for a call in a given RAN in layer $l = 1$ of the RAN. That is, the computation load $\sum_{b=1}^{B} \beta_b$ is placed on layer $l = 1$ of the RAN. In contrast, in the classical CRAN scenario, the function blocks $f_b,\ b = 1, 2, \ldots, B$ are performed in layer $l = L$, i.e., at the BBU, placing computation load $\sum_{b=1}^{B} \beta_b$ on the BBU.

We denote ρ_b for the data bitrate emanating from function block f_b processing. Specifically, after function block f_0, the data bitrate is the fixed I/Q time domain data rate $\rho_0 = R_{\text{I/Q time}}$. Each successive function block reduces the data bitrate towards the (IP packet level) payload data rate $\rho_B = r$.

Service Policy

Following the optimization results for a substantial MEC load in Garcia-Saavedra *et al.* (2018), we consider an elementary greedy service policy that strives to perform the function block computations for a given call generated for CRAN m within the own CRAN m at the lowest possible layer, i.e., as close as possible to the radio nodes. The investigation of other service policies is an important direction for future research.

We consider layer $l = 0$ as a "special" layer that conducts only the essential function block f_0 that results in the time-domain I/Q stream. We do not load layer $l = 0$ with any additional computations. Instead, we greedily try to place *all remaining* function blocks $f_b,\ b = 1, 2, \ldots, B$ on node $(l = 1, m)$. If node $(l = 1, m)$ cannot accommodate this full remaining computation load $\sum_{b=1}^{B} \beta_b$, then the SDN orchestrator tries to place the maximum integral number of function blocks on the node. That is, functions $f_b,\ b = 1, 2, \ldots, \mu$, are placed on node $(l = 1, m)$ with $\mu = \{\max_{0 \le b \le B} b \text{ subject to} \sum_{a=1}^{b} \beta_a \le Z^{\text{avail.}}_{l=1,m}\}$, where $Z^{\text{avail.}}_{l=1,m}$ denotes the currently available computing capacity at node $(l = 1, m)$.

If not all $(\mu < B)$ or none $(\mu = 0)$ of the function block computation loads $\beta_b,\ b = 1, 2, \ldots, B$, can be accommodated on node $(l = 1, m)$, then the SDN orchestrator tries to move the remaining function block computations [that could not be placed on node $(l = 1, m)$] to the next higher layer, i.e., layer $l = 2$, of the same operator m, i.e., to node $(l = 2, m)$. Again, the SDN orchestrator tries to place the maximum integral number of the remaining function blocks on node $(l = 2, m)$.

If node $(l = 2, m)$ cannot accommodate all remaining function block computations, then the SDN orchestrator tries to offload the remaining function blocks to the "parallel" neighbors, i.e., to the other nodes $n \neq m,\ 1 \leq n \leq N$, within layer $l = 1$.

If there are still some remaining function blocks, then the SDN orchestrator tries to place these remaining computations on the next "higher" layer $l = 3$ within the own CRAN m, i.e., on node $(l = 3, m)$. Then, if there are still some remaining function blocks, the SDN orchestrator tries the other nodes $n \neq m,\ 1 \leq n \leq N$, in layer $l = 2$, and so on. That is, the SDN orchestrator always tries first one layer up higher in the own CRAN and if this fails, then tries the other nodes one layer back. This process continues until all nodes have been checked. Note that on the last search iteration, the SDN orchestrator cannot try to offload to layer $L + 1$ (as this layer does not exist); instead, after attempting to place the remaining function blocks on the other CRAN nodes $n \neq m,\ 1 \leq n \leq N$, in layer $L - 1$, the SDN orchestrator immediately proceeds to the other CRAN nodes $n \neq m,\ 1 \leq n \leq N$, in layer L. If some (one or more) of the functions blocks for a data call cannot be accommodated, then the call is blocked.

Throughout, the transfer of a function block from a node (k, m) to a node (l, n) requires that the data bitrate rate emanating for the call from node (k, m) can be accommodated within the currently available communication capacity out of the total intra-layer communication capacity C_k if the nodes are in the same layer $(k = l)$ or the total inter-layer communication capacity $C_{k;l}$ if the nodes are in different layers $k \neq l$. We also note that we only consider the transfer (offloading) of complete function blocks, i.e., we do not consider the splitting of a given function block f_b into sub-blocks.

Performance Metrics

We evaluate the call blocking probability O for the low, medium, and high data rate calls. We evaluate the total mean revenue rate R defined as the long run average rate of completed calls weighed by the call payload data bitrate r. Moreover, we evaluate the MEC node utilization, i.e., the long-run average load level of each MEC node; in order to avoid clutter, we report the average (across the parallel N nodes in a layer) of these long-run average MEC loads for each layer $l = 1, 2, 3, 4$. We also evaluate the communication capacity utilization, i.e., the long run average bitrate transported across each of the intra-layer and inter-layer networks.

1.6 Fluid RAN Function Split Evaluation

1.6.1 Approximate Analysis

MEC node (l, n) can be viewed as a stochastic knapsack Ross (1995) of capacity $Z_{l,n}$. A function block f_b that is computed on node (l, n) occupies computing capacity β_b for the duration of the call. Similarly, the intra- and inter-layer communication capacities can be viewed as stochastic knapsacks. A detailed stochastic knapsack model with the different call data rates would become quite tedious. The main goal of our approximate analysis is to give insight into the sharing of the CRAN resources across the N parallel CRANs. Generally, by the scaling characteristics of stochastic knapsacks Ross (1995), one large system can support substantially more calls than a set of separate smaller systems (with the same overall capacity).

In order to derive a simple intuitive model that still captures the essential sharing dynamics, we focus on the computing aspect. We consider an approximate system model with compute capacity Z in each MEC node and one "average" call type with data bitrate $\bar{r}$ and corresponding average compute load $\bar{\beta}_b$ for function block b. In order to process an "average" call, the total computing demand $\bar{\beta}_{\text{tot}} = \sum_{b=1}^{B} \bar{\beta}_b$ has to be provided by the CRAN system. With one call type, the total CRAN compute capacity can be viewed as a classical trunking system that is characterized by the

Erlang B loss formula. For a classical trunking system with a call handling capacity of Γ calls and offered load E (call arrival rate times average call holding time in Erlangs), the blocking probability is

$$O(E,\ \Gamma) = \frac{E^\Gamma/\Gamma!}{\sum_{\gamma=0}^{\Gamma} E^\gamma/\gamma!}. \tag{1.1}$$

In our context, a given data call requires the processing of B function blocks in the CRAN system, i.e., places a compute load $\bar{\beta}_{\text{tot}}$ on the CRAN system. The call handling capacity of one conventional CRAN system is thus $\Gamma = LZ/\bar{\beta}_{\text{tot}}$. Data calls are generated at a rate of λ call/s for a given CRAN system, whereby a given call lasts on average τ seconds. Thus, the offered load for a CRAN system is $E = \lambda\tau$. Hence, one of the stochastically identical and independent conventional CRAN systems has approximately the blocking probability $O(\lambda\tau,\ LZ/\bar{\beta}_{\text{tot}})$.

Our system with resource sharing across the N parallel CRAN systems has a total call handling capacity of $\Gamma = NLZ/\bar{\beta}_{\text{tot}}$ and a total offered load of $E = N\lambda\tau$. Thus, the blocking probability is approximately $O(N\lambda\tau,\ NLZ/\bar{\beta}_{\text{tot}})$. By the classical trunking efficiency characteristics Smith and Whitt (1981), the system with resource sharing has substantially lower blocking probability, and correspondingly higher call completion rate. Accordingly, resource sharing increases the revenue rate $R = N\lambda(1 - O)\bar{r}$.

1.6.2 Simulation Setup

We consider $N = 3$ parallel CRANs, each with $L + 1 = 5$ MEC node layers. We initially set all node computing capacities to $Z_{l,n} = 200$ [arbitrary computing units]. We set all communication capacities to $C_l = C_{k;l} = 1000$ Gbps. For each given generated call, we independently randomly select a lifetime according to an exponential distribution with mean $\tau = 2$ [s], and we uniformly randomly select a payload data bitrate r from a set of three prescribed rates, i.e., $r \in \{r_{\text{low}} = 5$ Mbps, $r_{\text{med}} =$

30 Mbps, $r_{\text{high}} = 100$ Mbps}. We set the corresponding function block computing demands in the last function block $b = B = 4$ to $\beta_4^{\text{low}} = 1$, $\beta_4^{\text{med}} = 2$, and $\beta_4^{\text{high}} = 4$.

The compute loads and bitrates are typically highest for the function blocks near the radio node and decrease towards the BBU Garcia-Saavedra *et al.* (2018); Yeoh *et al.* (2016). We assume that each function block reduces the bitrate to a third of the bitrate entering the function block, i.e., we set $\rho_0 = R_{\text{I/Q time}} = 81r$, $\rho_1 = 27r$, $\rho_2 = 9r$, $\rho_3 = 3r$, and $\rho_B = \rho_4 = r$. We assume that the computing demands of the function blocks are halved for each successive function block, e.g., for a low rate call, $\beta_1 = 8$, $\beta_2 = 4$, $\beta_3 = 2$, and $\beta_4 = 1$.

Since function block 0 is often implemented with extensive specialized hardware support, we focus on function blocks $b = 1$ through $b = B = 4$ in our evaluations. Specifically, we assume that layer $l = 0$ in each CRAN m has always enough resources to accommodate the function block 0 processing of all calls arriving to CRAN m and that bitrate $\rho_0 = 81r$ is required to offload function block $b = 1$ from node $(l = 1, m)$ to another MEC node.

We evaluate statistical confidence intervals with the batch means method. We run the simulation for a given scenario until the 95% confidence intervals for all performance metrics are less than 5% of the corresponding sample means. The confidence intervals are not plotted to avoid visual clutter.

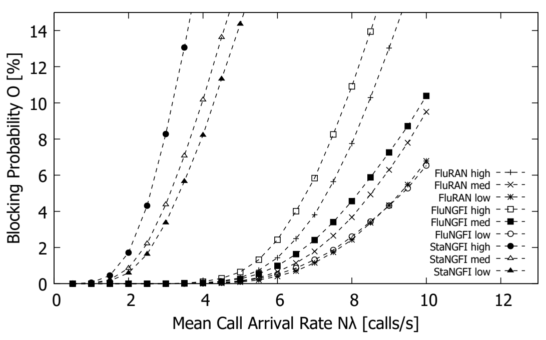
1.6.3 Evaluation Results

Fluid RAN Function Split

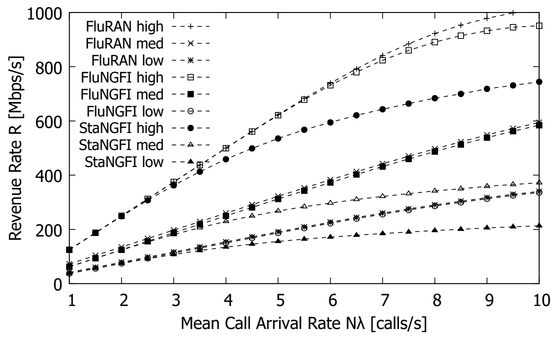
This section examines the fluid assignment of the RAN function blocks to MEC nodes. We compare our fluid RAN approach introduced in Section 1.5 with the state-of-the-art NGFI (IEEE P1914.1) based approaches, which statically assign the

RAN function blocks to RRH, DU node, CU node, and BBU Checko *et al.* (2015); Chih-Lin (2017); Garcia-Saavedra *et al.* (2018); Mharsi *et al.* (2018); Thyagaturu *et al.* (2018a); sagroups (1914); Chih-Lin *et al.* (2018a). Specifically, in our evaluation context, we consider the static assignment of function block $f_b,\ b = 1, 2, \ldots, B$, to the MEC node (b, m) of the considered CRAN m. That is, the static NGFI approach features a fine-granular splitting of the B computation tasks among B MEC node layers; however, this fine-granular assignment is statically fixed. As an additional fluid RAN evaluation benchmark we consider a fluid NGFI which we define as follows. The function block f_0 is conducted in the DU node attached to the RRH, while the remaining function blocks $f_b,\ b = 1, 2, \ldots, B = 4$, with aggregate computation demand $\sum_{b=1}^{B} \beta_b$ have to be completed at one flexibly assigned MEC node. We consider the placement of this aggregate computation load according to the greedy service policy on any of the MEC nodes $(l, m),\ l = 1, 2, \ldots, L$, of the considered CRAN m. The assigned MEC node takes on the role of the CU for the considered call, resembling the PHY split scenario in Garcia-Saavedra *et al.* (2018). We conduct the fluid RAN benchmark comparisons in the context of a CRAN system without resource sharing among parallel CRANs in order to bring out the performance trade-offs of the fluid (flexible) assignment of the RAN function blocks (computing tasks) to the MEC nodes within a given CRAN as enabled by the LayBack SDN based management framework.

We observe from Fig. 1.5(a) that StaNGFI has substantially higher blocking probability than FluNGFI, which in turn has slightly higher blocking probability than FluRAN. The static function block assignment with StaNGFI overloads the MEC nodes in layer $l = 1$ already for low call arrival rates with the high computation load β_1 of function block f_1. The flexible FluNGFI assignment of the complete (aggregate) set of function blocks with load $\sum_{b=1}^{B} \beta_b$ to any of the MEC nodes $l = 1, 2, 3$, or 4 avoids the overloading of the layer $l = 1$ MEC nodes. However, the complete set of function



(a) Call Blocking Probability O

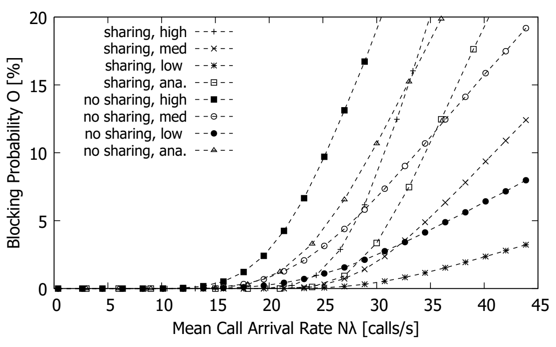


(b) Revenue Rate R

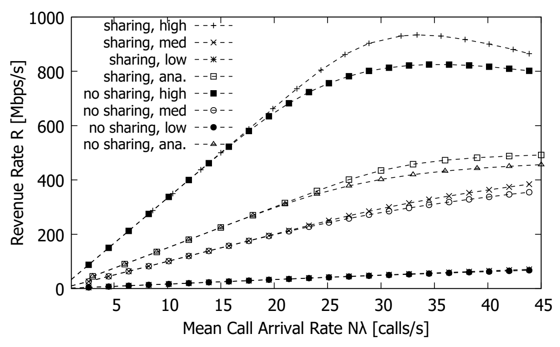
Figure 1.5: Performance of a CRAN with Flexible Fluid Assignment Of $b = 4$ Ran Function Block Computations to $l = 4$ Mec Nodes (Fluid Ran, Abbreviated as Fluran in Plot, Enabled by the Sdn Management Framework in Layback Architecture), Static Ngfi Based Assignment of Ran Function f_b Computation to Mec Node b (Stangfi), and Fluid Ngfi Based Assignment of Complete Set of b Ran Function Computations to a Mec Node out of the l Mec Nodes (Flungfi) for Uniform Data Call Arrivals to Each Cran.

blocks requires an available capacity of at least $\sum_{b=1}^{B} \beta_b$ at a MEC node, whereas the FluRAN approach requires only at least β_1 available computing capacity at a MEC node and then enough available capacity to accommodate the other function blocks with the smaller computation loads β_2, β_3, and β_4 at the subsequent (higher indexed) MEC nodes.

We observe from Fig. 1.5(b) that for the practically relevant blocking probability ranges, e.g., below 5%, the revenue rates for FluRAN and FluNGFI are essentially equivalent; however, the revenue rates for StaNGFI are significantly lower. These results underscore that the flexible assignment of RAN function blocks to the MEC nodes is important for extracting high revenues from a CRAN system. On the other hand, the granularity of the function block assignment (individual function blocks



(a) Call Blocking Probability O



(b) Revenue Rate R

Figure 1.6: Performance of System of n Parallel Crans with Resource Sharing among the n Crans (Enabled by Layback Coordination Point Just Behind Ran Gateways to Consistently Decouple Fronthaul From Backhaul and to Allow for Sdn Control of Fronthaul and Backhaul) Vs. Without Resource Sharing (Representing Conventional Architectures With Coupled Fronthaul and Backhaul That Make Sharing Prohibitively Complex, See Section 1.2.1) for Uniform Data Call Arrivals to Each Cran.

with fluid RAN approach vs. aggregate of function blocks with FluNGFI) has only a relatively minor impact.

RAN Sharing for Uniform Call Load

This section evaluates the resource sharing among CRAN systems, which LayBack enables through the positioning of the coordination point just behind the gateways of the respective RAN systems, see Fig. 2.1. This unique positioning of the coordination point in the LayBack architecture consistently decouples the fronthaul from the backhaul and allows for the flexible SDN control of the fronthaul and backhaul and the flexible coordination and cooperation between the different CRAN systems. In contrast, the RAN architectures in the existing literature reviewed in Section 1.2.1 generally retain some dependencies and direct interactions between fronthaul and backhaul. In these existing architectures, the fronthaul and backhaul are effectively coupled and a coordination among different RAN systems would only be possible through a coordination point behind the core networks, e.g., to the right of the legacy EPC in Fig. 2.1. Such a coordination point behind the core network layer would make the coordination prohibitively complex and far removed from the RAN fronthaul, i.e., would allow only for indirect control of the RAN fronthaul. Thus, fronthaul RAN sharing is generally not practical in the existing architectures. We compare a fluid RAN "no sharing" scenario representing the existing architectures (as reviewed in Section 1.2.1) with a fluid RAN "sharing" scenario representing the LayBack architecture (whereby the sharing is among the CRANs).

In particular, we first consider a uniform call generation scenario where each of the N CRAN systems receives the same call request rate λ. The fluid RAN approach shares the resources across the N CRAN systems. In contrast, the set of N parallel "no sharing" CRAN systems do not share resources, i.e., each of the "no sharing" CRANs processes calls only within its own system. The "no sharing" CRAN system offloads function blocks according to the fluid RAN approach but only to its own MEC nodes. That is, the "no sharing" CRAN system places function blocks greedily on the own MEC nodes as close to the radio node as possible; there is no offloading to MEC nodes in parallel CRANs.

We observe from Figs. 1.6(a) and (b) that resource sharing among parallel CRAN systems reduces the blocking probability while increasing the revenue rate. These performance gains are due to the sharing (statistical multiplexing) of resources across a larger system with our service policy. As noted in Section 1.6.1, our sharing service policy essentially lumps the N parallel CRANs into one large aggregate CRAN system with a total computing capacity of NLZ; whereas, the conventional approach has N separate CRAN systems, each with a computing capacity of LZ. The one large

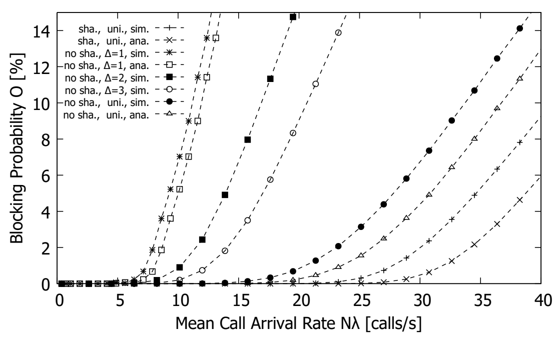
CRAN system obtained through the sharing can support more calls than a set of separate smaller systems (with equivalent overall capacity) due to the more flexible resource utilization in one large system compared to the separate small systems Smith and Whitt (1981).

Specifically, we observe from Fig. 1.6(a) that for the considered uniform call load scenario, the blocking probability reduction with sharing is relatively modest, typically on the order of 5% in the critical call arrival rate range when the blocking becomes noticeable, for aggregate call arrival rates $N\lambda$ around $20 - 30$ calls per second. The high rate calls require substantially more computing and communication resources than the medium and low rate calls and accordingly the high rate calls experience substantially higher blocking probabilities than the other call types. The rough analytical approximation from Section 1.6.1 does not consider the different call types, but confirms the general trends of the blocking probability and revenue dynamics.

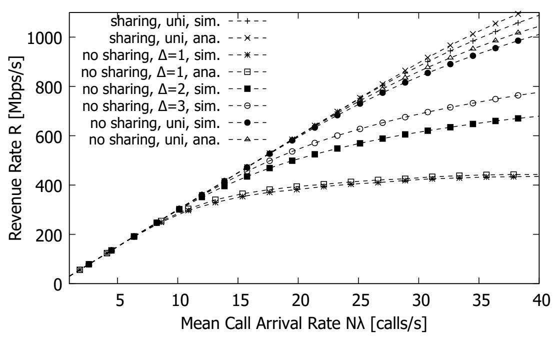
We observe from Fig. 1.6(b) that sharing brings only relatively small revenue increases for the uniform call load scenario. The increases with sharing are largest for the high rate calls around $N\lambda = 30 - 35$ calls/s. The high rate calls present a favorable combination of high revenue (which we set equal to the data bitrate) and low to moderate blocking probabilities up to around $N\lambda = 30 - 35$ calls/s. For higher arrival rates, more frequent low and medium rate calls fill up the free capacities and block the high rate calls, resulting in a drop of the revenue from high rate calls.

RAN Sharing for Non-Uniform Call Load

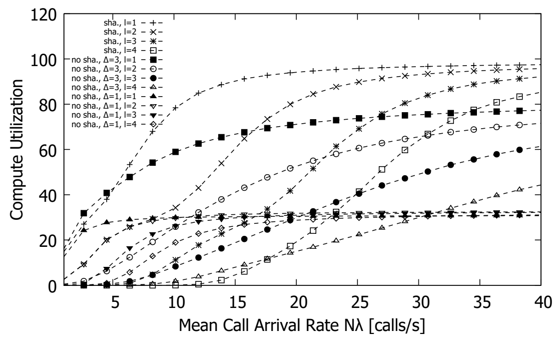
We evaluate different skewness levels of call arrivals that may arise due to shifts in call generation, e.g., due to popular events. We consider the Zipf distribution Adamic and Huberman (2002) with exponent $\zeta = 1$ for different numbers Δ of CRAN systems that receive medium rate calls. In particular, for $\Delta = 1$, the entire call generation



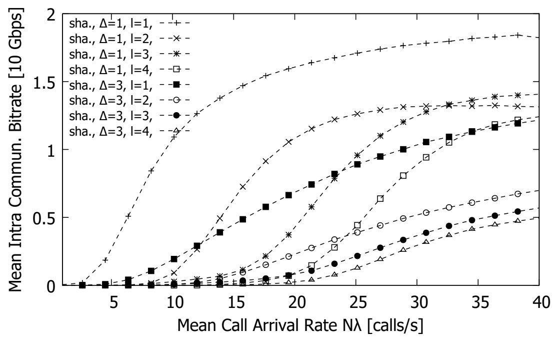
(a) Call Blocking Probability O



(b) Revenue Rate R



(c) MEC Utilization



(d) Intra-layer Communication Bitrates with Sharing

Figure 1.7: Performance of System of n Parallel Crans With Resource Sharing among the n Crans (Enabled by Layback) Vs. Without Resource Sharing (Conventional Architectures with Prohibitive Sharing Complexity) for Non-uniform Arrivals of Medium Rate Data Calls According to Zipf Distribution to δ Crans.

rate $N\lambda$ arrives to one CRAN system. For $\Delta = 2$, the call generation rate $N\lambda$ arrives to two CRAN systems according to the Zipf distribution with support 2, i.e., a given generated call arrives to one CRAN system with probability 2/3 and to the other CRAN system with probability 1/3. For $\Delta = 3$, the calls arrive to the three CRAN systems with proportions 6/11, 3/11, and 2/11.

We observe from Fig. 1.7(a) that for the CRAN system without resource sharing, the blocking probability substantially increases with increased skewness of the call arrivals, i.e., smaller Δ. We confirmed in additional simulations that are not included to avoid clutter that the blocking probability of the CRAN system with resource sharing is essentially unaffected by the skewness of the call arrivals. With sufficient intra-layer communication capacities, the resource sharing flexibly diverts the function blocks to the available resources in the parallel CRANs, keeping the call blocking low even for highly skewed call arrivals. We observe from Fig. 1.7(b) that the sharing greatly increases the revenue for skewed call arrivals. For the moderate case of skewed call arrivals with $\Delta = 3$ to all $N = 3$ CRAN systems, sharing can increase the revenue by approximately 25%, while more pronounced skewness in the call arrival pattern allows for even larger gains.

We also observe from Figs. 1.7(a) and (b) that the rough approximation with the Erlang B trunking model from Section 1.6.1 gives slightly lower blocking probabilities than the simulated CRAN system. This is mainly because the CRAN system function blocks f_1, f_2, f_3, and f_4 have specific placement constraints that are neglected in the lumped trunking system model. Mainly, the function blocks have decreasing computing demands $\beta_1 > \beta_2 > \beta_3 > \beta_4$ and need to be executed one after the other with the placement on MEC nodes according to the considered greedy service policy. Thus, the "no sharing" CRAN system blocks for example a call if MEC layer $l = 4$ has enough free compute capacity for β_1, but not enough to accommodate $\sum_{b=1}^{B} \beta_b$, and the other MEC layers $l = 1, 2$, and 3 have enough free capacity for β_2, β_3, and β_4, but not enough for β_1; while this example call would be accommodated in the trunking system.

For the compute utilization, we observe from Fig. 1.7(c) that without sharing the utilization levels are quite low compared to the CRANs with sharing. Without sharing, the $\Delta = 1$ scenario already fully loads the resources in the one CRAN receiving all the calls for fairly low call arrival rates. The resources in the two parallel CRANs cannot be utilized, i.e., they have a utilization of zero. Thus, the overall utilization of the compute resources of the $N = 3$ parallel CRANs is limited to one third. In contrast, the sharing utilizes the compute resources across all $N = 3$ parallel CRANs. (Additional simulations that are not included to avoid clutter confirmed that sharing achieves very similar utilization levels for all considered call arrival patterns.) The considered greedy function block placement policy more and more fully utilizes the successive MEC layers $l = 1, 2, 3$, and 4 as the call arrivals increase. For the $\Delta = 3$ scenario in Fig. 1.7(c), all $N = 3$ CRANs receive calls, but with rates skewed according to the Zipf distribution. Without sharing, the CRAN system receiving the highest proportion of calls reaches near full utilization already for moderate call arrival rates and blocks calls, while the two parallel CRAN systems have still unutilized compute resources. In contrast, the fluid RAN system with resource sharing among the N CRAN systems consistently achieves high resource utilization, low blocking probability, and high revenue for a wide range of call arrival patterns.

Fig. 1.7(d) shows the intra-layer communication bitrates for layers $l = 1, 2, 3$, and 4 for the resource sharing between the $N = 3$ parallel CRAN systems. We observe that the extreme case of all calls arriving to $\Delta = 1$ CRAN system results in relatively high bitrates in layer $l = 1$ already for low call arrival rates. The intra-layer bitrates in the successive layers $l = 2, 3$, and 4 increase as the call arrival rate increases. This behavior is in accordance with the considered greedy service policy that strives to complete function blocks in the lowest indexed layers. For the more realistic case of skewed arrivals to all $\Delta = N = 3$ CRAN systems, we observe significantly lower intra-layer communication bitrates, whereby layer $l = 1$ experiences again the highest intra-layer bitrates.

1.7 Conclusions

We have introduced the Layered Backhaul (LayBack) architecture for coordinating heterogeneous radio access networks (RANs) with software defined networking (SDN). LayBack ties the heterogeneous RANs together behind their respective gateways, such as cloud RAN (CRAN) baseband units of small cell gateways. More specifically, these heterogeneous gateways are connected by an SDN network to a unifying SDN orchestrator. We have introduced an SDN based management framework that is executed in the SDN orchestrator. The management framework coordinates distributed computing resources, such as distributed multiple-access edge computing (MEC) nodes, to cohesively provide computing for network services.

We showcased the LayBack architecture and management framework for a novel fluid cloud RAN (CRAN) function split. The fluid function split partitions the entire set of CRAN fronthaul computations into multiple function blocks. The function blocks are assigned to MEC nodes according to a service policy. We evaluated an elementary greedy service policy that shares resources between the CRAN systems of different operators. We found that the resource sharing substantially increases the revenue rate from the CRAN service.

LayBack can serve as basis for a wide range of future research directions. One direction is to develop and evaluate optimization mechanisms for the function splitting (problem decomposition) and the allocation of the resulting function blocks (sub-problems). Initial work in this direction has, for instance, explored time-scale based decompositions Ferrari *et al.* (2018a); Tang *et al.* (2017). The convergence and optimality characteristics of such time-scale decompositions as well as other problem decomposition approaches need to be thoroughly examined in future research.

Table 1.1: Summary of Main Notations and Parameter Settings For Numerical Evaluations in Section 1.6.

	CRAN/MEC Network	
N	Number of parallel CRANs (e.g., operators)	3
L	Number of layers of MEC nodes	4
(l, n)	MEC node indices, $0 \leq l \leq L;\ 1 \leq n \leq N$	
$Z_{l,n}$	Compute capacity of node (l, n)	200
C_k	Intra-layer comm. capacity [Gbit/s] in layer k	10^3
$C_{k;l}$	Inter-layer comm. capacity [Gbit/s] betw. layers k and l	10^3
	Data Call	
r	Payload (IP level) data bitrate [Mbit/s]	5, 30, 100
τ	Expected duration [s]	2
λ	Arrival rate [calls/s] per CRAN	
	Function Blocks	
B	Number of function blocks for RAN function, indexed with $b,\ b = 0, 1, \ldots, B$	4
β_b	Computation demand (load) of function block b	Sec. 1.6.2
ρ_b	Bitrate [bit/s] departing function block b; $\rho_0 = R_{\mathrm{I/Q\ time}}$; $\rho_B = r$	Sec. 1.6.2
	Performance Metrics	
O	Call blocking probability	
R	Revenue rate from completed calls	

Chapter 2

LAYERED BACKHAUL OPTIMIZATION

2.1 Introduction

2.1.1 Motivation

In conventional wireless networks, each wireless service operator maintains its own wireless network infrastructure with its own backhaul network that interconnects the wireless network frontend with the Internet at large. Typically, each operator has a fixed maximum installed backhaul capacity. Sudden demand surges for backhaul capacity from the wireless devices and the corresponding radio nodes, e.g., the LTE enhanced Node Bs (eNBs) and Wi-Fi access points (APs), of one operator may overwhelm the operator's backhaul capacity and result in poor service quality, and ultimately, reduced revenue. Overall, with the advances in the wireless transmission capacities, the backhaul has emerged as a critical bottleneck of novel high-capacity wireless networks, such as small cell networks and 5G networks Ferrari *et al.* (2018b); Andrews *et al.* (2014); Lopez Rodriguez *et al.* (2019); Hassan and Gao (2019); Mikaeil *et al.* (2018); Wang *et al.* (2015b); Yang (2019).

Recently, Software-Defined Networking (SDN)-based backhaul architectures have been proposed to flexibly interconnect the backhaul networks of the different operators in a centrally controlled manner, as reviewed in detail in Section 2.2. The central SDN control enables the dynamic on-demand sharing of the backhaul resources among the various operators. Thus, sudden demand surges for backhaul capacity from the radio nodes (e.g., LTE eNBs and Wi-Fi APs) of one operator may be served by sharing the backhaul capacities of the various operators. Of course, aside from the technical capabilities, appropriate legal and business agreements need to be in place between the operators to make the sharing practically feasible and economically advantageous.

The SDN-based control of the backhaul resource sharing poses two main challenges. First, the aggregate of multiple operator networks with all their radio nodes and subscribing wireless devices considered by the central SDN orchestrator can be very large; thus posing scalability challenges. Second, the central SDN orchestrator that coordinates among multiple operators may be far removed from the distributed radio nodes resulting in long signaling delays and accordingly slow reactions to dynamic demand variations at the radio nodes. Thus, purely centralized optimization is not practical for backhaul networks. Rather, distributed optimization strategies are needed for operating large backhaul networks.

Network optimization that operates on distributed systems has so far had two flavors: (*i*) peer-to-peer optimization for a flat (i.e., not layered) system and (*ii*) Network Utility Maximization (NUM) Kelly *et al.* (1998); Lin *et al.* (2006); Chiang *et al.* (2007); Chiang (2008) which employs dual decomposition to distribute computations across different terminals that share the network resources. The dual decomposition leads to a master-slave model, where each user (slave) needs to directly interact with the SDN controller (master). The main benefit of operating the NUM is that the SDN controller simply passes the dual variable iterates, and the slaves pass only their demands (while locally monitoring their constraints). Thus, the SDN controller does not need to know all the details of the users and yet, can solve the global optimization problem. However, for large backhaul networks, it is not practical for the numerous eNBs to directly interact with the central SDN controller, as the NUM dual decomposition would require. Essentially, the well-researched NUM dual decomposition models are incompatible with the multiple layers in large practical backhaul network architectures.

2.1.2 Contributions

This article presents a generalization of the NUM framework using auxiliary variables to make NUM modeling compatible with the multiple layers in layered backhaul network architectures. More specifically, we decompose the global optimization that the central SDN orchestrator is trying to solve, through formulation of a multi-layer NUM. Moreover, we include a virtual queue framework in our formulation to allow the SDN orchestrator to comply to long-term agreements. Our formulation strives to optimize the sharing of backhaul resources in an SDN-based backhaul network architecture. In particular, this case study is conducted in the context of the recently proposed LayBack backhaul network architecture Shantharama *et al.* (2018a). LayBack, as reviewed in more detail in Section 2.3, introduces layers for the different wireless network components, including layers for wireless devices, radio nodes (e.g., eNBs, Wi-Fi APs), and gateways (e.g., small cell gateways, LTE gateways). LayBack interconnects the gateways through an SDN switching layer in a full mesh with the respective core network entities (e.g., the LTE Enhanced Packet Core (EPC)) of the various operators. The gateways and core network entities of the various operators as well as the SDN switching network are under the control of a unifying SDN orchestrator. The LayBack backhaul architecture provides centralized fine-grained tuning knobs to optimize the backhaul operation, e.g., to share backhaul capacity among the different operators in a dynamic fashion. Our layered iterative optimization formulation distributes the optimization computations over the LayBack layers. We conduct numerical evaluations with the formulated optimization to quantify the performance gains that the optimized backhaul resource sharing in the SDN-based LayBack architecture achieves compared to the conventional non-SDN backhaul.

2.2 Background and Related Work

2.2.1 SDN-Based Backhaul Architectures

The SDN paradigm with a control plane that is separate from the data plane and with a centralized SDN controller has spurred significant research interest in wireless networks Amin *et al.* (2018); Haque and Abu-Ghazaleh (2016); Jagadeesan and Krishnamachari (2014). An extensive set of studies have developed SDN-based architectures for the backhaul of wireless network traffic Marabissi *et al.* (2019); Niephaus *et al.* (2015); Tayyaba and Shah (2019). Given the complexity of the backhaul, most of this work has considered layered or tiered architectures Cavaliere *et al.* (2017); Costa-Perez *et al.* (2017a); Elgendi *et al.* (2016); González *et al.* (2016); Gutiérrez *et al.* (2016); Oliva *et al.* (2015); Shantharama *et al.* (2018a), whereby intermediate gateway nodes perform various protocol related functions Mayoral *et al.* (2017); Chih-Lin *et al.* (2018b); Thyagaturu *et al.* (2016a); Tonini *et al.* (2018). For instance, there may be gateway nodes that interface with Internet of Things nodes Cilfone *et al.* (2019); Silva *et al.* (2019) or specific wireless network technologies, such as wireless local area networks Kostal *et al.* (2019). Moreover, the efficient interconnection via metropolitan area networks to the Internet at large has received increasing attention King *et al.* (2019); Tzanakaki *et al.* (2017).

This case study considers the LayBack architecture Shantharama *et al.* (2018a) which can encompass the layering structures of a wide range of other proposed architectures and wireless technologies while allowing for fine-grained SDN control, as elaborated in Section 2.3. The enabling idea is the generalization of the decomposition approach referred to as *network utility maximization* (NUM) to a multi-tier system.

2.2.2 Network Optimization

Kelly et al. Kelly *et al.* (1998) introduced the NUM concept to solve the problem of rate allocation in a network with link capacity constraints. Extensive follow-up studies have analyzed the NUM concept in the contexts of distributed optimization and stochastic network theory Lin *et al.* (2006); Chiang *et al.* (2007); Chiang (2008). For instance, Tassiulas and Ephremides Tassiulas and Ephremides (1992, 1993) analyzed Queue length Maximum Weight (QMW) scheduling, which facilitated the subsequent analysis of throughput optimality conditions and related performance guarantees Kar *et al.* (2008); Ji *et al.* (2013). However, QMW scheduling does not guarantee minimal delay Cui and Yeh (2014), which has led to investigations of QMW variations that reduce delays in general multi-hop networks Cui *et al.* (2016) or provide better delay guarantees Kar *et al.* (2012); Neely (2013). A common shortcoming of these optimization models is that a centralized optimal scheduler can be impractical, mainly due to scalability problems and signaling delays. Decompositions of NUM models can provide the desired implementation scalability to a certain extent. The existing NUM model decompositions strictly conform to a master-slave architecture, which means that these decompositions do not truly reflect the many intermediate layers that exist between the edge devices and the core network in large-scale wireless networks.

Furthermore, NUM model decompositions commonly build on the so-called *timescale separation assumption* Palomar and Chiang (2006); Johansson *et al.* (2006), which states that the session interval is much longer than the convergence time of the greedy resource allocation policy Chiang (2008). With the timescale separation assumption, the decomposition neglects the convergence of the local control. Building on the timescale separation assumption, decentralized algorithms for link scheduling based on queue lengths have been proposed in Gupta *et al.* (2009); Bui *et al.* (2009); Teng and Song (2017).

Our multi-layer multi-timescale NUM framework contains two innovative aspects. First, rather than having a single central master (or single master layer), we consider four decomposition layers that include the SDN controller at the backhaul, the operators, the network gateways, and the eNodeBs. Second, we consider realistic network signaling latencies for the decomposition of the QMW utility over the LayBack architecture layers. The signaling delays make the timescale separation assumption unrealistic. Generally, there have been two categories of studies that have examined the removal of the timescale separation assumption: (1) studies that use intermediate iterates as decisions and assume continuous underlying flows Lin *et al.* (2008); Srikant (2004), and (2) studies that use a multi-timescale approach across different layers of the protocol stack Altman *et al.* (2012); Pham *et al.* (2015). More specifically, the study Lin *et al.* (2008) showed that a β-fairness utility function can be maximized, while guaranteeing system stability, under the assumptions that the number of users per class follows a recurrent Markov Chain. We follow a similar rationale as Lin *et al.* (2008) for the intermediate decisions. Moreover, similarly to the second category of studies, we consider multiple timescales. While the different allocation problems in prior studies had been placed in different layers of the conventional protocol stack, the different allocation problems correspond to different layers of the LayBack architecture in our optimization model.

The Lyapunov drift-plus-penalty method introduced in Georgiadis *et al.* (2006); Neely (2006) has been extensively used in recent years for enforcing constraints in dynamic control. We employ the Lyapunov drift-plus-penalty method to incorporate an economic constraint in the allocation across different operators.

2.2.3 Wireless Backhaul Network Optimization

Judicious usage of the resources in backhaul networks can greatly enhance the wireless services while increasing revenues Ge *et al.* (2019); Luong *et al.* (2019). Generally, the dynamic sharing of installed transmission resources is a promising strategy for enhancing the performance of wireless backhaul networks Bernal-Mor *et al.* (2013); Biermann *et al.* (2012); De Domenico *et al.* (2013); Lakshminarayana *et al.* (2013); Li *et al.* (2019d); Liu *et al.* (2016b,a); Niu *et al.* (2016); Samdanis *et al.* (2016); Semiari *et al.* (2015); Taleb *et al.* (2015). Our specific focus is on expanding the scope of resource sharing by exploiting the centralized control that SDN provides while operating within the hierarchical layer structure of the SDN-based backhaul networks. We note that aside from the general enhancement of resource sharing and use, some recent optimization studies have sought to consider specific objectives, such as to minimize energy consumption Ali *et al.* (2019); Cen *et al.* (2019); Scarpiniti *et al.* (2019), or to optimize for uploading specific content, e.g., video Yang *et al.* (2019a).

Typically, the different hierarchical layers cover geographic regions of different scopes and operate on different timescales, e.g., fast timescales in small localized regions and slow timescales over wide-area regions. To the best of our knowledge, only a few studies have explicitly considered these heterogenous scopes and timescales. Prasad et al. Prasad *et al.* (2014) combined an allocation of users to a set of beam vectors in the backhaul of a heterogeneous wireless network on a slow timescale with a corresponding transmission time slot allocation on a fast timescale. Tang et al. Tang *et al.* (2017) examined interactions between slow timescale resource allocation in a pool of baseband units (BBUs) in a cloud radio access network with a fast-timescale beam-forming in remote radio heads. The related recent study Tang *et al.* (2019) has examined the interactions between the slicing of the upper layers of the communication network stack at a slow timescale with the fast-timescale wireless channel dynamics, while the study Lyu *et al.* (2018) has considered multiple timescales for optimizing a decentralized SDN control structure. Moreover, in the context of computation task scheduling on virtual machines, a collaborative centralized and distributed control approach has recently been examined in Xia *et al.* (2019), while multiple timescales have been examined in Chen *et al.* (2019). We also note that two timescales have been considered for minimizing energy costs for data center computations Yao *et al.* (2014); Yu *et al.* (2015) and for smart grid optimization Wang *et al.* (2018a).

Complementary to these prior studies, we present a case study on the optimal dynamic allocation of an abstract backhaul resource (represented by a bitrate) over a total of four layers operating on four different timescales. A preliminary version of parts of this case study has appeared in Ferrari *et al.* (2018b). This article gives a refined comprehensive presentation of our case study, including the complete set of algorithms for solving the four sub-problems at the considered four layers, whereas only the algorithm for solving one subproblem was worked out in Ferrari *et al.* (2018b). Moreover, this article gives the full details of the evaluation methodology and expanded results.

We note that this case study does not seek to examine theoretical convergence guarantees for multi-layer multi-timescale optimization. Initial steps towards such a theoretical analysis have recently been reported in Karakoc *et al.* (2018). As a complement to and a motivation for detailed theoretical analyses, this present case study seeks to demonstrate the feasibility of the multi-layer optimization with multiple timescales and to showcase performance gains for wireless backhaul.

2.3 Overview of Layered Backhaul (LayBack) Network Architecture

The LayBack network architecture Shantharama *et al.* (2018a) categorizes the backhaul network elements, such as switches, gateways, and core networks, into layers that are broader than the traditional access networks, aggregation networks, and data center networks. The LayBack architecture envisions to homogenize the multitude of networking technologies, such as cable, cellular, and traditional Ethernet through a unifying SDN orchestrator.

2.3.1 Layers in LayBack

We briefly review the layers in the LayBack architecture, which is illustrated in Figure 2.1, focusing mainly on the context of cellular networks. The end-device layer encompasses the heterogeneous mobile wireless end devices. The radio node layer includes the LTE eNBs and Wi-Fi APs. The gateway (GW) layer encompasses the network entities between the radio node layer and the backhaul (core) entities, e.g., entities of the legacy enhanced packet core. For instance, the GW layer may include the gateways of small cell deployments, or the Base Band Units (BBUs) of a cloud radio access network. Similarly, a Cable Modem Termination System (CMTS) Gowdal *et al.* (2018); Granizo Arrabe *et al.* (2018), which serves as a gateway for radio nodes Thyagaturu *et al.* (2018a), belongs to the GW layer. The SDN switching layer consists of SDN switches that flexibly interconnect the radio node layer with the backhaul (core) layer. Radio nodes operating in a non-C-RAN environment (such as macro cell eNBs) process the baseband signals locally and connect directly to the backhaul (core) layer network gateways via the SDN switching layer. The backhaul (core) network layer comprises technology-specific network elements, such as the Evolved Packet Core (EPC) which supports the connectivity of LTE eNBs.

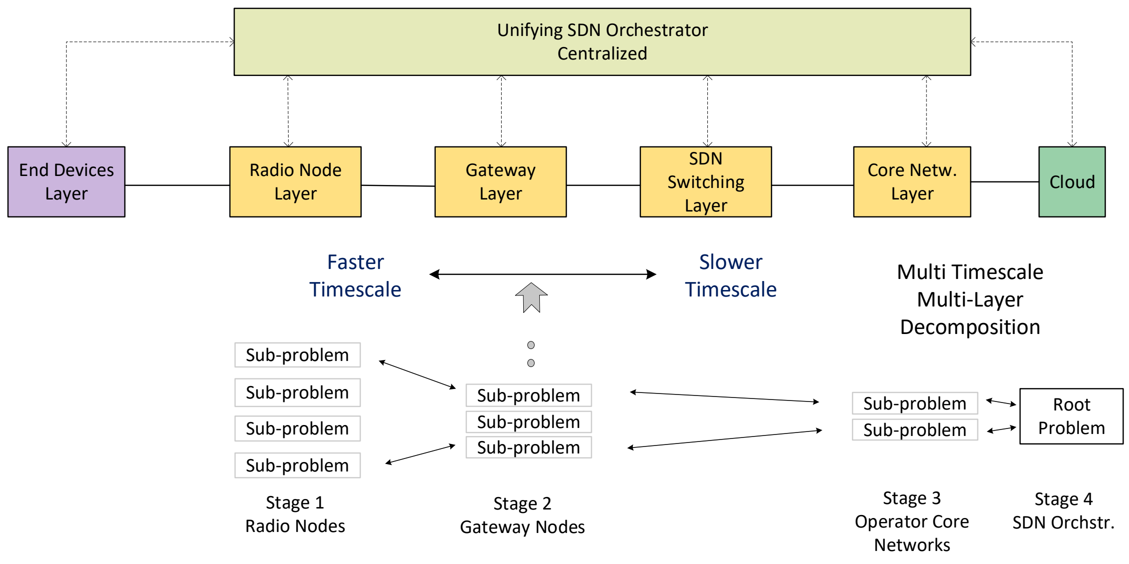


Figure 2.1: Illustration of Layback Architecture and Multi-timescale Optimization Decomposition in Context of Cellular Networks: Layback Partitions the Wireless Backhaul Infrastructure into Radio Node Layer, Gateway layer, Sdn switching Layer, And core Network Layer. The Entire Network Is Controlled by the Central Unifying Sdn Orchestrator. This case Study Decomposes the Optimization of The Sharing of the Backhaul Bitrate of Multiple Operator Core Networks Into Fast-timescale Sub-problems at the Radio Nodes And Progressively Slower Timescale Sub-problems at the Gateways And Operator Core Networks; Whereby All Sub-problems Are Coordinated Through a Root Problem at the Sdn Orchestrator.

2.3.2 Management in LayBack

The unifying SDN orchestrator in LayBack has three main tasks: (1) it creates a common platform for coordinating among all the wireless service operators and heterogeneous network technologies across its layers; (2) it maintains the current topology information of the entire network and tracks the network capabilities; (3) it enables each of the layers to flexibly reconfigure the network by allocating resources in response to their time-varying needs, while maintaining long-term performance requirements that define the service guarantees. The networks maintained by different operators periodically communicate their requirements and reconfiguration capabilities to the SDN orchestrator to enable the SDN orchestrator to fulfill its tasks. Next, we show how these tasks can be combined with an online optimal resource sharing task that leverages our multi-layer multi-timescale NUM framework.

2.4 Layered SDN-Based Optimization Framework

2.4.1 Overview

This section formulates a multi-layer multi-timescale optimization model for the backhaul resource sharing in LayBack. The optimization model is decomposed into multiple layers so that the orchestrator centrally controls the resource sharing among the operators, while distributing the decision-making processes to ensure scalability. The multiple timescales facilitate quick dynamic reactions to the needs of the network end users while accommodating the signaling delays to the central SDN orchestrator.

In our optimization model, we abstract away the actual relationships between the physical layer wireless communication resources (i.e., spectrum and power) at the radio node layer (eNB, Wi-Fi AP) and the corresponding dynamic allocation of the bitrate. We focus on the management of an abstract total backhaul bitrate resource Z, which is indirectly tied to the redistribution of the physical layer wireless communication resources. The SDN-based LayBack architecture maintains a logically separated queue at each radio node. The shared resource Z trickles down from the unifying SDN orchestrator to the operators, from each operator to its gateways (GWs) and, finally, from each GW to its radio nodes.

2.4.2 Model Definitions

We consider a network with O distinct operators, indexed by $o = 1, 2, \ldots, O$. (The main model definitions are summarized in Table 2.1.) Each operator manages a set $\mathcal{G}_o$ of GWs indexed by $g \in \mathcal{G}_o$. In turn, each GW g manages a set of eNBs, indexed by $n \in \mathcal{N}_g$. Let us also define the set $\mathcal{N} \triangleq \bigcup_{o=1}^{O} \bigcup_{g \in \mathcal{G}_o} \mathcal{N}_g$ of all the eNBs and the set $\mathcal{G} \triangleq \bigcup_{o=1}^{O} \mathcal{G}_o$ of all the GWs. The queue at a given eNB $n \in \mathcal{N}$ is denoted by Q_n and its dynamics are

$$Q_n[t+1] = [Q_n[t] - z_n[t]]^+ + a_n[t+1], \tag{2.1}$$

where $a_n[t]$ and $z_n[t]$ represent, respectively, the exogenous packet arrival process and the backhaul service rate that is granted to eNB n during time slot t. Also, $[\cdot]^+$ denotes the projection onto the nonnegative orthant ($[\gamma]^+ = \max(\gamma, 0)$). The service rate $z_n[t]$ represents the backhaul (bitrate) resources allocated for the upstream (eNB to GW) transmission between t and $t+1$ to the specific eNB n.

The multi-layer multi-timescale optimization framework developed in this section is applicable for the wide range of optimal resource allocations to distributed entities. In particular, the developed optimization framework is well suited for scenarios with substantial signaling delays between the distributed entities and a central controller so that purely centralized decisions are impractical. Aside from large-scale wireless access networks, such resource allocation problems arise for instance in supply and demand management Thyagaturu *et al.* (2016b) and in transactive energy markets Nasrallah *et al.* (2018, 2019); Thyagaturu *et al.* (2018b); Alharbi *et al.* (2017); Rehmani *et al.* (2021).

Table 2.1: Summary of Model Notations.

		Values
Parameter	**Notation**	**(for eval. in Section 2.5)**
	Backhaul Netw. Architecture	
# of Operators (indexed $o = 1, \dots, O$)	O	2
# of GWs per oper. o	$\lvert\mathcal{G}_o\rvert$	3
# of eNBs per GW g	$\lvert\mathcal{N}_g\rvert$	10
Total Backhaul Cap. (Mbps)	Z	20
Operator Backhaul Cap. (Mbps)	Z_o	10
eNB-to-GW RTT (ms)	τ_N^G	1
GW to Operator RTT (ms)	τ_G^O	100
Operat. to SDN Orch. RTT (s)	τ_O^S	1
	Resource Allocations	
Cap. alloc. to Oper. o	x_o	
Vector of Oper. alloc.	$\mathbf{x} = \{x_1, \dots, x_O\}$	
Cap. alloc. to GW g	y_g	
Vector of alloc. to GWs at Op. o	$\mathbf{y}_o = \{y_g : g \in \mathcal{G}_o\}$	
Cap. alloc. to eNB n	z_n	
Vector of alloc. to eNBs at GW g	$\mathbf{z}_g = \{z_n : n \in \mathcal{N}_g\}$	

2.4.3 Centralized Queue Length Minimization

Before introducing our timescale decomposition, we start from the centralized optimization we wish to emulate, and the logical steps that decompose the problem into layers via the Lagrange decomposition. If the SDN orchestrator, with full control of the total service rate Z, could allocate rates directly to the eNBs, the optimization would be:

$$\max_{\boldsymbol{z}} \sum_{n\in\mathcal{N}} \mathcal{U}_n(z_n) \text{ s.t.} \sum_{n\in\mathcal{N}} z_n \le Z,\ 0 \le z_n \le Q_n[t]\ \forall n \in \mathcal{N}, \tag{2.2}$$

where we use the QMW policy as objective function with $\mathcal{U}_n(z_n) = Q_n[t]z_n$ for the sake of illustrating the decomposition technique. In this formulation, the first constraint represents the overall backhaul capacity, whereas the second constraint defines the feasible region for optimization variable z_n which is limited by serving all packets in the queue per one time slot. We remark that an alternative optimization case study would be to consider the wireless device queues as the bottom layer queues. For such an alternate optimization model with wireless device queues, the utility should include the state w of the wireless channel which could be incorporated as $f(Q_n[t], w, z_n[t])$, whereby f is a known function of the queue, channel state information w, and service rate $z_n[t]$.

With the QMW policy, the maximization in (2.2) leads to the minimization of the long-term average total queue length, which also results in the minimization of the end-to-end delay in the network (a consequence of Little's theorem Allen (1990) for the simplified scenario of continuous flows and infinite queue backlogs Banirazi *et al.* (2012)).

2.4.4 Operator Resource Constraints

There are two potential problems with solving (2.2): (1) the allocation of network resources at the level of granularity of individual eNBs may result in scalability problems; and (2) without any long-term constraints, some operators may hoard backhaul resources. In order to create multiple layers to distribute the decision-making processes, we rewrite the maximization in (2.2) by introducing variables that for the sake of solving (2.2), are slack variables. As we will see, the additional variables represent actual network decisions in the distributed and time-decomposed implementation of the centralized scheduler.

In particular, let us denote by x_o the portion of the wireless service rate Z that is distributed to operator o and let $\boldsymbol{x} = \{x_1, x_2, \dots, x_O\}$ denote the vector of allocated operator service rates. Each operator o, $o = 1, \dots, O$, redistributes the resources, by giving a portion y_g of x_o to each of its GWs $g \in \mathcal{G}_o$, whereby we denote $\boldsymbol{y}_o = \{y_g : g \in \mathcal{G}_o\}$ for the vector of GW rate allocations of operator o. In turn, each GW g redistributes the resources, by giving a portion z_n of y_g to each of its eNBs $n \in \mathcal{N}_g$, whereby we denote $\boldsymbol{z}_g = \{z_n : n \in \mathcal{N}_g\}$. If all these assignments could happen at the same timescale indexed by t, distributing the constraints at each layer, the optimization could be solved as follows:

$$\max_{\boldsymbol{x}}. \sum_{o=1}^{O} \mathcal{U}_o^\star(x_o; t) \quad \text{s.t.} \sum_{o=1}^{O} x_o \le Z \tag{2.3}$$

with $\mathcal{U}_o^\star(x_o; t)$ being the optimal value of the subproblem:

$$\max_{\boldsymbol{y}_o}. \sum_{g \in \mathcal{G}_o} \mathcal{U}_g^\star(y_g; t) \quad \text{s.t.} \sum_{g \in \mathcal{G}_o} y_g \le x_o \tag{2.4}$$

and $\mathcal{U}_g^\star(y_g; t)$ being the optimal value of

$$\max_{\boldsymbol{z}_g} \sum_{n \in \mathcal{N}_g} Q_n[t] z_n \text{ s.t.} \sum_{n \in \mathcal{N}_g} z_n \le y_g, \; 0 \le z_n \le Q_n[t] \; \forall n \in \mathcal{N}_g. \tag{2.5}$$

It is important, however, to remark that the allocation of $\boldsymbol{x}$ to solve (2.3) needs to respect an "economic" constraint across the operators that defines a contractual service obligation and prevents any operator from gaming the system (i.e., consistently acquiring more resources than what it paid for). This constraint on the long-run average of the decisions $\boldsymbol{x}$ is:

$$\limsup_{\tau \to \infty} \frac{1}{\tau} \sum_{t=0}^{\tau-1} x_o[t] \le Z_o, \tag{2.6}$$

where for consistency of the problem, it is necessary to have $\sum_{o=1}^{O} Z_o \le Z$. At the same time, by having an inequality constraint, we are not forced to assign resources to an operator that would be wasted if there is not sufficient uplink demand.

We use the concept of virtual queues, following the Lyapunov drift-plus-penalty approach Georgiadis *et al.* (2006) to encode the constraint in (2.6), and we modify the objective in (2.3) into:

$$\sum_{o=1}^{O} \mathcal{U}_o^\star(x_o;t) - \frac{1}{V}\sum_{o=1}^{O} \Theta_o[t]x_o. \tag{2.7}$$

After deciding $\boldsymbol{x}[t]$, the virtual queues Θ_o are updated as:

$$\Theta_o[t+1] = \left[\Theta_o[t] + (x_o[t] - Z_o)\right]^+, \tag{2.8}$$

where Z_o is the fixed average maximum resource limitation of operator o.

The parameter V represents the "flexibility" of the constraint in (2.6), e.g., the higher V the more inclined we are to temporarily violate the constraint. The next subsection serves as a basis to tackle the problem at different timescales that are aligned with the network infrastructure, as elaborated in Section 2.4.6. It is however easier to derive them in the *ideal* static case first, given that the expressions in the dynamic case will have the same form, albeit with different meanings.

2.4.5 Iterative Solution via Gradient Descent

We omit the time index t to avoid notational clutter. The dual objective function of the subproblem (2.3) can be written as

$$\Phi_1\Big(y_g, \lambda_{y_g}; \mathbf{Q}\Big) \triangleq \lambda_{y_g} y_g + \max_{\boldsymbol{z}_g} \sum_{n\in\mathcal{N}_g} (Q_n - \lambda_{y_g}) z_n, \tag{2.9}$$

whereby $\mathbf{Q}$ denotes the vector of queue occupancies. We introduce the Lagrangian dual variable λ_Z for the constraint in (2.3), the Lagrangian dual variables $\{\lambda_{x_o} : o = 1,\ldots,O\}$ for the constraints in (2.4), and the Lagrangian dual variables $\{\lambda_{y_g} : g \in \mathcal{G}\}$ for the constraints in (2.5). Then, unfolding all the constraints, we obtain (2.2) and following a cascade of primal dual decompositions (see Palomar and Chiang (2006)), the optimization can be solved via the sequence of projected gradient descent updates:

$$\min_{\lambda_Z} \overbrace{\lambda_Z Z + \sum_{o=1}^{O} \max_{x_o} \overbrace{\left(-\lambda_Z - \frac{\Theta_o}{V}\right) x_o + \min_{\lambda_{x_o}} \overbrace{\lambda_{x_o} x_o + \sum_{g\in\mathcal{G}_o} \max_{y_g} \overbrace{-\lambda_{x_o} y_g + \min_{\lambda_{y_g}} \Phi_1(y_g, \lambda_{y_g}; \mathbf{Q})}^{\Phi_2(\lambda_{x_o}, y_g; \mathbf{Q})}}^{\Phi_3(x_o, \lambda_{x_o}; \mathbf{Q})}}^{\Phi_4(\lambda_Z, x_o; \mathbf{Q})}}^{\Phi_5(\lambda_Z; \mathbf{Q})} \qquad (2.10)$$

Figure 2.2: Optimization Solved Via the Sequence of Projected Gradient Descent Updates.

$$\lambda_Z^{(k+1)} = \left[\lambda_Z^{(k)} - \alpha_1^{(k)}\left(Z - \sum_{o=1}^{O} \underset{x_o}{\operatorname{argmax}}\, \Phi_4\big(\lambda_z^{(k)}, x_o\big)\right)\right]^{+} \qquad (2.10)$$

$$x_o^{(k+1)} = \left[x_o^{(k)} + \alpha_2^{(k)}\left(\underset{\lambda_{x_o}}{\operatorname{argmin}}\, \Phi_3\big(x_o^{(k)}, \lambda_{x_o}\big) - \lambda_Z - \frac{\Theta_o}{V}\right)\right]^{+} \qquad (2.11)$$

$$\lambda_{x_o}^{(k+1)} = \left[\lambda_{x_o}^{(k)} - \alpha_3^{(k)}\left(x_o - \sum_{g\in\mathcal{G}_o} \underset{y_g}{\operatorname{argmax}}\, \Phi_2\big(\lambda_{x_o}^{(k)}, y_g\big)\right)\right]^{+} \qquad (2.12)$$

$$y_g^{(k+1)} = \left[y_g^{(k)} + \alpha_4^{(k)}\left(\underset{\lambda_{y_g}}{\operatorname{argmin}}\, \Phi_1(y_g^{(k)}, \lambda_{y_g}) - \lambda_{x_o}\right)\right]^{+}, \qquad (2.13)$$

where the different α denote step sizes. The bottom layer optimization in (2.9) can be solved with Algorithm 1, while the solution for a general utility is shown in Low and Lapsley (1999). We note that to ensure the convergence of the decomposition, the updates in (2.10)–(2.13) must be read as follows: to reach the optimal λ_Z, the SDN orchestrator needs to perform a sufficient number of iterations in (2.10). However, before computing one iteration of (2.10), the operator layer below should perform a sufficient number of iterations of (2.11) upon receiving the Lagrangian λ_Z, and so on. Unless a value can be computed in closed form in one shot, each update that includes the solution of an optimization problem (i.e., it has an argmax or argmin term in the update) requires a sufficient number of gradient descent updates at the lower level to approximate the solution of the subproblem. Therefore, the indices k in (2.10)–(2.13) are *not* associated with the same timescale. If the computation at each layer and the communication delays among layers were all negligible, we would be in the *timescale separation* regime Palomar and Chiang (2006); Johansson *et al.* (2006). However, this is not possible in a real system, since latencies play a significant role in real networks and the framework we are about to explain explicitly takes these latencies into consideration. We also note that in this decomposition model, there is no sharing of information among the operators, which makes the model more practical. All message passing occurs only between neighboring layers, whereby the lower layer sends the optimal resource allocation and the upper layer sends the dual variable.

Algorithm 1: Solution of (2.9) (at GW g).

```
Input  : $y_g, \{Q_n : n \in \mathcal{N}_g\}$
Output : $\lambda^\star_{y_g}$, $\mathbf{z}_g$
1 if $\sum_{n \in \mathcal{N}_g} Q_n \geq y_g$ then
2     Find the permutation $\boldsymbol{\pi} = \{\pi_i : i = 1, \dots, |\mathcal{N}_g|\}$ to
3     sort the queues $Q_n$ such that $i \geq j \Rightarrow Q_{\pi_i} \leq Q_{\pi_j}$;
4     Find $i^* = \inf\{i : \sum_{j=1}^{i} Q_{\pi_j} \geq y_g\}$;
5     $z_{\pi_j} = Q_{\pi_j}$ for $j < i^*$, $z_{\pi_{i^*}} = y_g - \sum_{j=1}^{i^*-1} Q_{\pi_j}$, $z_{\pi_j} = 0$ for $j > i^*$, $\lambda^\star_{y_g} = Q_{\pi_{i^*}}$;
6 else
7     $z_n = Q_n \; \forall n \in \mathcal{N}_g, \lambda^\star_{y_g} = 0$;
8 end
```

Let us start by considering the optimization at the bottom layer as the one that operates at the minimum latency, i.e., the time difference between the time indexes t and $t+1$ is the Round Trip Time (RTT) between GW and eNB τ_N^G (considered equal, for simplicity, for all GWs and eNBs), since it is the one closest to the devices and to the information regarding traffic. To map all the time instants into integer values of t, it is convenient to normalize all times with respect to τ_N^G (i.e., we set $\tau_N^G = 1$).

Our framework considers that in actual network infrastructures one has constraints that prevent the redistribution of the total resource across the operators (e.g., the decisions $\{x_o : o = 1, \ldots, O\}$) and redistribution of operator resources across the GWs (e.g., the decisions $\{y_g : g \in \mathcal{G}\}$) from changing at the same timescale of the redistribution of GW resources across the eNBs (e.g., the decisions $\{z_n : n \in \mathcal{N}\}$). Therefore, even if a genie could compute the optimal solution of the decomposed problem at each instant t, it might not be possible to implement the decision.

Denoting with $\underline{L}$ and $\underline{P \cdot L}$ the minimum refresh times for the GW decisions $\boldsymbol{y}$ and for the operator's decisions $\boldsymbol{x}$, respectively, time t can be written according to a poly-phase decomposition as

$$t = (mP + p)L + \ell,\ m \in \mathbb{N},\ 0 \leq p \leq P - 1,\ 0 \leq \ell \leq L - 1, \tag{2.14}$$

where $P \cdot L > \underline{P \cdot L}$ and $L > \underline{L}$ are the selected refresh times. We illustrate the multi-timescale dynamics of the optimization framework in Figure 2.3 showing the interactions of eNBs, GWs, operators, and SDN orchestrator.

In the next subsection, to comply with the refresh time limits, the greedy optimization, decoupled at any instant t, is mapped into the stochastic optimization we solve. Changing the objectives from deterministic values to expected values is necessary to capture the uncertainty of the impact of the decisions $\boldsymbol{x}$ and $\boldsymbol{y}$ on the future queues evolving at a faster timescale, e.g., on the effect that a change in higher layers' resources distribution, produces on the lower layers' optimizations.

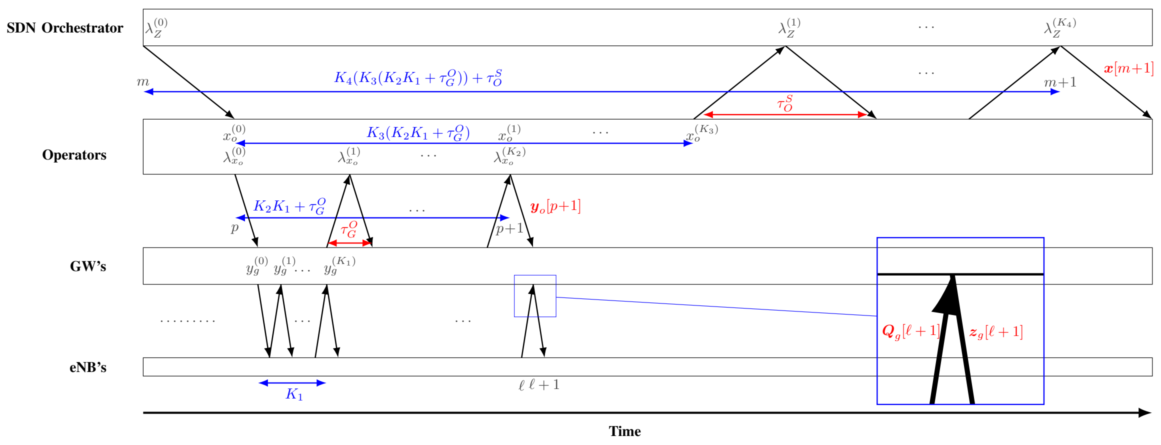


Figure 2.3: Illustration of the Dynamics of the Multi-timescale Optimization Framework Within Context of Layback Infrastructure: The Optimal Policy to Minimize End-to-end Delay Is Decoupled Into Multiple Layers of Sub-problems, With faster Timescales at the Lower Layback Layers The Enbs $n,\ N \in \mathcal{N}_g$, At a Gw g Pass Their Queue Occupancies Each Enb-gw Round-trip Time Rtt τ_n^g to Gw g. Based on the Received Vector of Queue Occupancies $\mathbf{Q}$, Gw g Evaluates The Allocations $\mathbf{Z}_g$ to Its enbs with Algorithm 1. Similarly, The Sdn Orchestrator Evaluates the Allocations $\mathbf{X}$ to The Operators with Algorithms 2 And 3; While Each operator o Evaluates the Allocations $\mathbf{Y}_o$ to Its Gws with Algorithms 4 And 5. (In order to Reduce Clutter, The Enb-to-gw Rtt τ_n^g Has Been Normalized to One In the Illustration, I.E., k_1 in the Illustration Corresponds to $k_1\tau_n^g$ in Actual Time).

Algorithm 2: At the SDN orchestrator.

```
  Input  : $\lambda_Z^{(0)},\ k_4 = 0$
  Output : $\lambda_Z^{(K_4)}$, $\boldsymbol{x}$
1 while $k_4 < K_4$ do
2     Call Algorithm 3 with input $\lambda_Z^{(k_4)}$ to all operators;
3     Receive $\boldsymbol{x}^{(K_3)}\Big(\lambda_Z^{(k_4)}\Big)$ and update $\lambda_Z^{(k_4+1)}$ via (2.10);
4     $k_4 \leftarrow k_4 + 1$;
5 end
6 Decide $\boldsymbol{x}[m+1]$ by projecting $\boldsymbol{x}^{(K_3)}\Big(\lambda_Z^{(K_4-1)}\Big)$ onto the feasible set in (2.2);
7 $m \leftarrow m + 1$;
8 Call Algorithm 2 with input $\lambda_Z^{(0)} \leftarrow \lambda_Z^{(K_4)}$;
```

Algorithm 3: Iterates for x_o (at operator o).

```
  Input  : $\lambda_Z,\ k_3 = 0$, ($x_o^{(0)}$ only if first call)
  Output : $x_o^{(K_3)}$
1 while $k_3 < K_3$ do
2     Call Algorithm 4 with input $x_o^{(k_3)}$;
3     Receive $\lambda_{x_o}^{(K_2)}\left(x_o^{(k_3)}\right)$ and update $x_o^{(k_3+1)}$ via (2.11);
4     $k_3 \leftarrow k_3 + 1$;
5 end
6 $x_o^{(0)} \leftarrow x_o^{(K_3)}$
```

Algorithm 4: Iterates for λ_{x_o} (at operator o).

```
Input  : $x_o, k_2 = 0$, ($\lambda_{x_o}^{(0)}$ only if first call)
Output : $\lambda_{x_o}^{(K_2)}$, $\mathbf{y}_o$
1 while $k_2 < K_2$ do
2     Call Algorithm 5 with input $\lambda_{x_o}^{(k_2)}$;
3     Receive $y_g^{(K_1)}\left(\lambda_{x_o}^{(k_2)}\right)$ and update $\lambda_{x_o}^{(k_2+1)}$ via (2.11);
4     $k_2 \leftarrow k_2 + 1$;
5 end
6 Decide $\boldsymbol{y}\,[(mP + (p+1))L]$ by projecting $\boldsymbol{y}^{(K_1)}\left(\lambda_{x_o}^{(K_2-1)}\right)$ onto the feasible set
  in (2.16);
7 $p \leftarrow p + 1$;
```

Algorithm 5: Iterates for y_g (at GW g).

```
Input  : $\lambda_{x_o}, k_1 = 0$, ($y_g^{(0)}$ only if first call)
Output : $y_g^{(K_1)}$
1 while $k_1 < K_1$ do
2     Call Algorithm 1 with input $y_g$ to solve (2.5);
3     Receive $\lambda_{y_g}^{\star}(y_g^{(k_1)})$ and update $y_g^{(k+1)}$ via (2.13);
4     $k_1 \leftarrow k_1 + 1$;
5 end
```

2.4.6 Stochastic Optimization and Temporal Decomposition

Since the different layers cannot communicate instantaneously, the parameters of the queues change dynamically underneath. Clearly, the objectives of the optimization must be defined in such a way that they stay constant while the bottom layer changes stochastically from one state to the next. The proposed framework can be seen as a special case of *stochastic gradient descent* where the network dynamics, via the evolution of the queues, impose the sequence of training sample updates. In particular, the SDN orchestrator operates its optimization at every time instant $t = mPL$, performing

$$\begin{aligned} &\max_{\boldsymbol{x}} \sum_{o=1}^{O} -\frac{\Theta_o[m]x_o}{V} + \frac{1}{P}\sum_{p=0}^{P-1} \mathbb{E}\left\{\mathcal{U}_o^\star\left(x_o; (mP{+}p)\, L\right)\right\} \\ &\text{s.t.} \sum_{o=1}^{O} x_o \leq Z, \end{aligned} \tag{2.15}$$

with $\mathcal{U}_o^\star\left(x_o; (mP+p)\, L\right)$ equal to the optimal value of the problem solved at the operator layer below:

$$\begin{aligned} &\max_{\boldsymbol{y}_o} \sum_{g\in\mathcal{G}_o} \frac{1}{L}\sum_{\ell=0}^{L-1} \mathbb{E}\left\{\mathcal{U}_g^\star\left(y_g; (mP+p)L+\ell\right)\right\} \\ &\text{s.t.} \sum_{g\in\mathcal{G}_o} y_g \leq x_o, \end{aligned} \tag{2.16}$$

and $\mathcal{U}_g^\star\left(y_g; (mP+p)L+\ell\right)$ being the optimal values of the optimization in (2.5) for $t = (mP+p)L+\ell$. The updates derived in (2.10)–(2.13) will then be used to update the decisions $\boldsymbol{x}$ every PL and the decisions $\boldsymbol{y}$ every L, as if convergence to the solution of a static problem has been achieved in the time horizons of length PL and L, respectively. By introducing K_i as the number of iterations of each update in layer $i = 1, \ldots, 4$, respectively, starting from the bottom, we can derive the following relations:

$$PL \geq \max\left\{K_4\left(K_3\left(K_2K_1\tau_N^G + \tau_G^O\right) + \tau_O^S\right),\ \underline{PL}\right\} \tag{2.17}$$

$$L \geq \max\left\{K_2K_1\tau_N^G + \tau_G^O,\ \underline{L}\right\}, \tag{2.18}$$

where τ_O^S, τ_G^O, and τ_N^G, are, respectively, the RTTs between the SDN orchestrator and the operators, between the operators and the GWs, as well as between the eNBs and the GWs (see also Figure 2.3, where τ_N^G has been normalized to one). The inequalities in (2.17)–(2.18) indicate that if we want to act fast, e.g., reduce P and L (possibly to the minimum refresh times) we need to perform fewer iterations. Vice versa, if we want to perform more iterations, we must be willing to act slower in updating the decisions $\boldsymbol{x}$ and $\boldsymbol{y}$. If we view the static problem as a "surrogate" for the dynamic problem (up to the next decision), increasing the number of iterations and delaying future decisions can guarantee a better accuracy for a static scenario; however, the ability of the algorithm to incorporate new dynamic information is compromised. This trade-off creates another optimization issue which is an important future research direction.

2.5 Numerical Evaluation Results

In this section, we describe the evaluation setup for this numerical optimization case study and discuss the evaluation results obtained with the optimization approach described in the preceding section.

2.5.1 Evaluation Setup

We have implemented the optimization framework described in Section 2.4 in MATLAB to evaluate the allocation of the backhaul bitrate resources in the upstream path in LayBack. The upstream data path consists of eNB, GW, and an operator core network.

LayBack Architecture

Initially, we consider a LayBack network architecture with $O = 2$ network operators, which we index with $o = 1$ and $o = 2$. Each network operator has three GWs for a total of six GWs. Each gateway has ten eNBs for a total of 60 eNBs.

Each operator has an installed backhaul bitrate resource (capacity) of $Z_o =$ 10 Mbps. Assuming that the two operators have agreements to fully share each other's backhaul capacity, the aggregate available backhaul bitrate (capacity) is $Z = 20$ Mbps. The objective of the optimization is to optimally share the available backhaul resource of $Z = 20$ Mbps among all eNBs attached to all the GWs of both operators $o = 1$ and $o = 2$.

The round-trip propagation delays (RTTs) are set to eNB-GW RTT $\tau_N^G = 1$ ms, GW-operator RTT $\tau_G^O = 100$ ms, and operator-SDN orchestrator RTT $\tau_O^S = 1$ s.

Optimization Parameters

The iteration parameters are set to $K_1 = 10$, $K_2 = 5$, $K_3 = 10$, and $K_4 = 1$. Following the lower bounds imposed by the K values in Equations. (2.17) and (2.18), we set $PL = 2500$ and $L = 150$. By default, we set the mean drift-plus-penalty parameter to $V = 1000$. For all the updates, $\alpha = 0.4$ and for numerical stability, the computation of $\lambda^{\star}_{y_g}$ considers the queue normalization $\frac{Q_n}{\sum_{n \in \mathcal{N}_g} Q_n} \frac{|\mathcal{N}_g|}{2}$, which does not alter the solution.

Comparison Benchmark

The baseline in our evaluation is the performance of a no-SDN wireless scheduling framework, i.e., the absence of the LayBack orchestrator to coordinate the scheduling. As a result, each operator o can only occupy its own backhaul bandwidth Z_o, i.e., there is no inter-operator bandwidth sharing. More specifically, in our simulations, the no-SDN benchmark solves only the optimization up to the subproblem Equation (2.4) with the dynamic operator allocation x_o replaced by the static operator capacity Z_o, and the subproblem Equation (2.5) [but not the subproblem Equation (2.3)]. That is, the no-SDN benchmark only optimizes the allocations within each given operator individually, i.e., performs essentially only "intra-operator" optimization. The no-SDN benchmark follows the same multi-timescale behavior with $K_1 = 10$,

$K_2 = 5$, and $K_3 = 10$ as the SDN-based optimization. We report the aggregate of the gateway allocations $\sum_{g \in \mathcal{G}_o} y_g$ for operators $o = 1$ and 2 as the actual allocated operator upstream bitrates of the no-SDN benchmark; whereas, for the SDN-based optimization, we report the operator rate allocations x_o.

We note that an alternate benchmark without any optimization could consider a static allocation of backhaul capacity portions to individual eNBs. Such a static allocation would perform poorly for dynamic bursty traffic models, as specified in Section 2.5.1. The static allocation would incur substantially longer queue lengths than the considered "intra-operator" optimization, which individually independently optimizes the allocations within each operator. Another alternative benchmark could employ conventional two-layer NUM between the eNBs and the operators (with the gateways subsumed by the operators). Such a two-layer benchmark would still perform the intra-operator optimization, but with only two layers compared to the three layers in the considered benchmark. These two benchmarks would generally perform similarly, with differences being influenced by convergence characteristics Karakoc *et al.* (2018). For the present study, we focus on the impact of the sharing of the backhaul resource across operators as quantified by comparing the considered no-SDN "intra-operator" optimization with the full SDN-based optimization involving the central SDN orchestrator.

Traffic Model

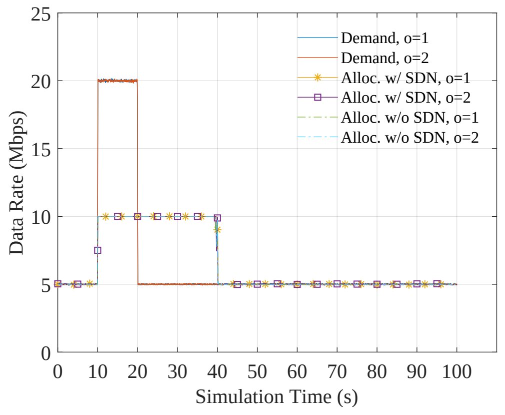
We model the upstream packet traffic generation at a given eNB (which is due to upstream packet arrivals from associated user end devices Bikram Kumar *et al.* (2019)) as an independent Poisson process. We set the eNB Poisson process rates such that the aggregate load from the eNBs at a given operator o results in a base packet traffic load of 5 Mbps, whereby each of the 30 eNBs at a given operator o contributes equally to the aggregate operator load. We conduct simulations of 100 s of backhaul network operation, whereby the Poisson traffic generation occurs over time increments of 0.1 ms, i.e., one simulation run of 100 s corresponds to one Million Poisson packet traffic generation instantiations.

We consider dynamic upstream traffic variations, which can, for instance, be caused by new temporary connection establishment or data connection handovers, e.g., due to user mobility. Specifically, we initially simulate a peak load of 20 Mbps occurring by default at operator 1 from 10 to 20 s of a simulation run and at operator 2 from 50 to 60 s of a simulation run.

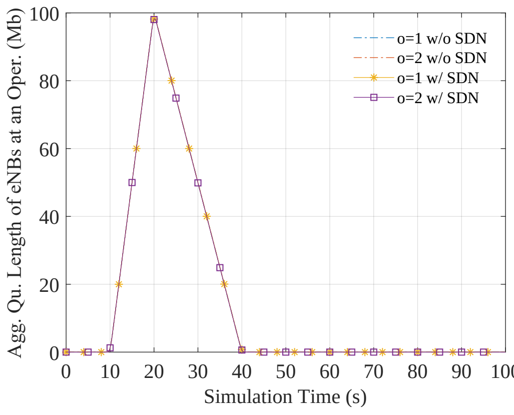
2.5.2 Results

Temporal Spacing of Operator Peak Demands

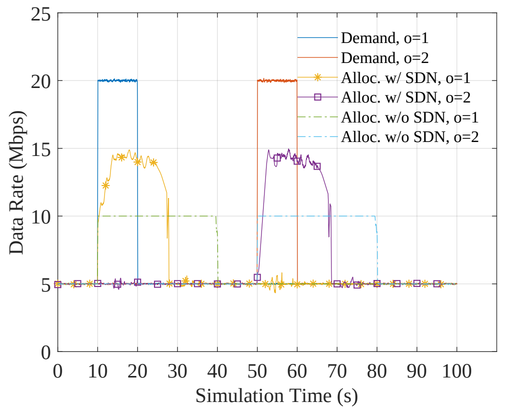
Overlapping Peak Demands We first verify the correct operation of the SDN-based optimization for a scenario that does not permit inter-operator bitrate sharing, specifically, for a scenario where the peak periods of the upstream bitrate demand at the eNBs of the two operators occur simultaneously, as illustrated in Figure 2.4a. Both operators experience a jump of the demanded upstream bitrate from 5 Mbps to 20 Mbps at simulation time 10 s; the 20 Mbps peak load persists for 10 s, and then returns to the 5 Mbps base load level. Note that these load levels correspond to the prescribed Poisson process rates, i.e., the actual load levels vary according to the stochastic characteristics of the Poisson packet generation processes around the prescribed bitrates, as is visible through the slight random "ripples" of the demand bitrates in Figure 2.4a.



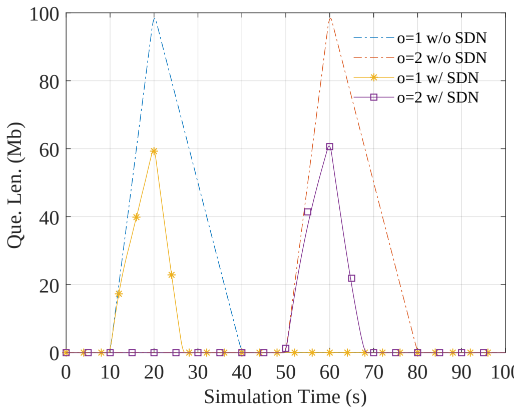
(**a**) Overlap.: Demand, Rate Alloc. to Op.



(**b**) Overlap.: Agg. eNB Queue Length for Op.



(**c**) Sep.: Demand, Rate Alloc. to Op.



(**d**) Sep.: Agg. eNB Queue Length for Op.

Figure 2.4: Upstream Traffic Demands and Corresponding Backhaul Bitrate Allocations to Operators as Well as Aggregated Queue Length of Enbs Associated With a given Operator When Peak Demand Periods of the Two Operators Overlap or Are Separated (Fixed parameter: Mean Drift-plus-penalty Parameter $v = 1000$): For Overlapping Peak Demands (**a**,**B**), Both the Sdn-based Optimization and the Benchmark Without Sdn Allocate to Each Operator Its Maximum Capacity of $z_o = 10$ mbps to Serve The Peak Demands; There Is No Sharing among Operators. For separated Peak Demands (**C**,**D**), The sdn Orchestrator Dynamically Shares The Total Aggregated Backhaul Capacity of $z = 20$ mbps among the Two Operators, Reducing enb Queue Lengths Compared to the Benchmark Without Sdn-based Resource Sharing.

We observe from the curves for the allocated operator rates (x_o with SDN, $\sum_{g\in\mathcal{G}_o} y_g$ without SDN) in Figure 2.4a that both the SDN and no-SDN approaches allocate the maximum operator rate of $Z_o = 10$ Mbps to serve the peak load. Since both operators experience the peak load at the same time, sharing among operators would not be sensible. Rather, each operator o should use its own full upstream bitrate resource Z_o to minimize packet delays. We observe from Figure 2.4a that the SDN-based optimization meets this intuitive optimization goal and gives essentially the same rate allocations as the no-SDN benchmark. In particular, given the equal demands from the eNBs of both operators, the SDN-based optimization strives to allocate an equal share of half of the total upstream backhaul bitrate of $Z = 20$ Mbps to each operator while serving the peak load. Thus, for the entire simulation time duration, the resource allocation with SDN-based optimization follows the resource allocation without SDN.

The no-SDN benchmark solves the optimization up to the subproblem Equation (2.4), whereby the operator upstream transmission capacity is limited to $Z_o = 10$ Mbps with the considered parameter settings. Thus, by solving subproblem (2.4), each operator in the no-SDN benchmark is able to allocate up to $Z_o = 10$ Mbps when a demand burst occurs.

We note that a conventional static allocation of backhaul bitrate (without any dynamic optimization, not even the intra-operator optimization of the no-SDN benchmark) would allocate $Z_o = 10$ Mbps for the entire simulation duration. However, only 5 Mbps out of these 10 Mbps could be used during the time period from 0 to 5 s and from 40 s onwards to the end of the simulation time, thus leading to wasted backhaul bandwidth.

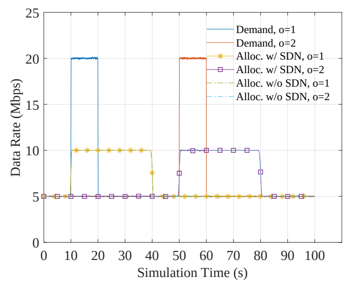
We observe from Figure 2.4b that the queue lengths of the eNBs at both operators linearly increase at a constant rate since both operators experience the same peak load that exceeds their respective available backhaul bitrate Z_o. In particular, the queue lengths increase from 0 to a maximum value corresponding to 10 s $\times(20 - 10)$ Mbps $=$ 100 Mbit while the peak load is feeding into the eNBs from 10 to 20 s simulation time. Subsequently, the queue length decreases down to zero over 20 s as effectively an "extra" backhaul bitrate of 5 Mbps, i.e., the allocated 10 Mbps minus the currently served base load of 5 Mbps, is serving the backlog from 20 s to 40 s simulation time.

Separated Peak Demands Figure 2.4c,d considers the more typical operational scenario when peak demands for the different operators are separated in time, e.g., due to different traffic and mobility patterns of the end users. We observe from Figure 2.4c that the SDN-based optimization with backhaul resource sharing among the two operators allocates up to 15 Mbps to the operator that currently experiences the peak demand (while 5 Mbps continue to serve the other operator); thus fully using the total available backhaul bitrate $Z = 20$ Mbps. In contrast, the benchmark without SDN does *not* share backhaul capacity among operators. Accordingly, without SDN, operator 1 can only serve the peak demand that occurs from 10–20 s with its own 10 Mbps capacity; meanwhile, operator 2 uses only 5 Mbps of its 10 Mbps capacity and the other 5 Mbps are wasted.

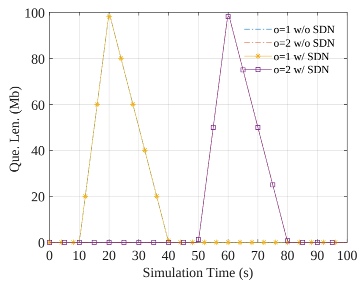
The SDN-based backhaul resource sharing reduces the queue build-up in the eNBs, as observed in Figure 2.4d compared to the benchmark without SDN, implying shorter latencies with SDN-based sharing. The slight variations between the optimization behaviors for the peak demands of operators 1 and 2 are due to the random variations of the actual demands around the prescribed mean Poisson traffic rates.

Impact of Flexibility Parameter V

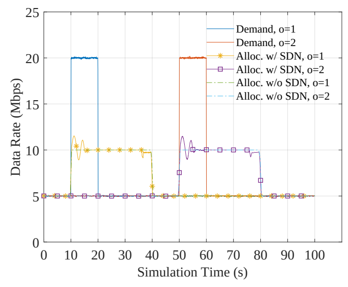
The mean-plus drift parameter V in the optimization framework, see Equation (2.7), relates to the degree of flexibility with which the operators can share the total aggregate backhaul capacity Z beyond their own backhaul capacity Z_o. Figure 2.5 shows the performance of the resource allocation algorithm for increasing values of the flexibility parameter V, namely for $V = 1$, 10, and 100, while for $V = 1000$ we refer to Figure 2.4c,d. Moreover, Figure 2.5g,h shows the optimization performance without the economic constraint (2.6). We observe from Figure 2.5a,c that for small V values, e.g., $V = 1$ and 10, the rate allocation with SDN optimization is nearly equivalent to the no-SDN benchmark. The small differences between the allocations with the SDN optimization and the no-SDN benchmark are mainly some low-amplitude oscillations in the SDN allocations. The allocation oscillations result from the optimization framework striving to adapt to slight random variations in the traffic generation processes. Accordingly, both the SDN optimization and the no-SDN benchmark give essentially the same eNB queue lengths as observed from Figure 2.5b,d. Intuitively, small V values restrict the drift from the mean in the optimization framework, which inherently corresponds to a low degree of flexibility when operators want to share each other's resources.



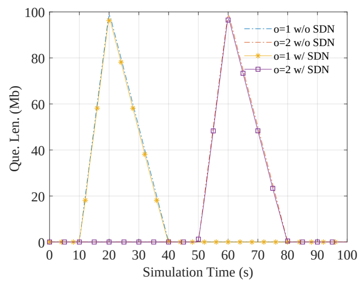
(**a**) $V = 1$ Demand, Rate Alloc. to Op.



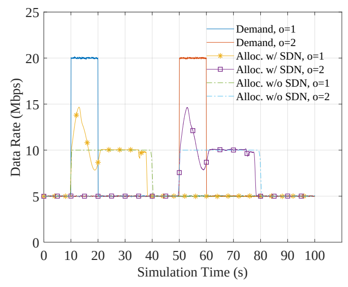
(**b**) $V = 1$ Agg. eNB Queue Length at Op.



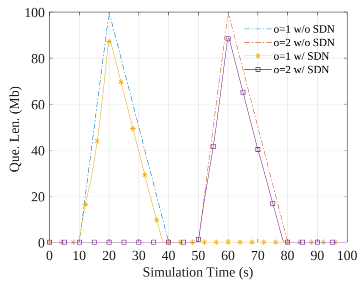
(**c**) $V = 10$ Demand, Rate Alloc. to Op.



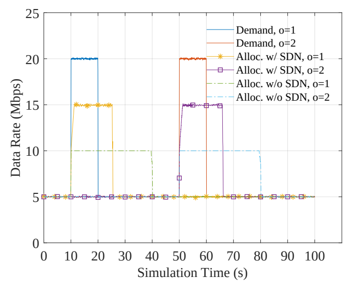
(**d**) $V = 10$ Agg. eNB Queue Length at Op.



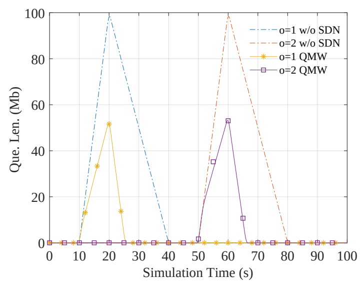
(**e**) $V = 100$ Demand, Rate Alloc. to Op.



(**f**) $V = 100$ Agg. eNB Queue Length at Op.



(**g**) QMW Demand, Rate Alloc. to Op.



(**h**) QMW Agg. eNB Queue Length at Op.

Figure 2.5: Upstream Backhaul Bitrate Allocations and Enb Queue Lengths When Demand Peaks for Operators 1 and 2 Are Spaced Apart: Increasing The "flexibility Parameter" v, See equation (2.7), Increases the Sharing of Backhaul Capacity 2.4.

In contrast, we observe for the higher $V = 100$ and 1000 values in Figure 2.5f and Figure 2.4d that SDN optimization with flexible backhaul resource sharing among operators achieves smaller eNB queue lengths than the no-SDN benchmark. We observe that the queue lengths for $V = 1000$ in Figure 2.4d are nearly as small as the queue lengths in Figure 2.5h for optimization without the long-run rate allocation constraint. Indeed, the rate allocation without the rate allocation constraint in Figure 2.5g is being approximated by the SDN rate allocation in Figure 2.4c. The rate allocation constraint safeguards against persistent unfair backhaul capacity usage by a given operator and is therefore generally recommended for operational networks.

Overall, we observe from Figures 2.4 and 2.5 that the SDN-based optimization of backhaul resource sharing can significantly lower the eNB queue lengths. These lowered eNB queue lengths translate into significantly reduced latencies for the end-user upstream traffic.

Impact of Spacing between Operator Traffic Bursts

While Figures 2.4 and 2.5 considered a fixed 40 s separation of the starting time instants of the data bursts (of 10 s duration) at the two operators, we consider a range of burst separations in Figure 2.6. We observe from Figure 2.6 that a burst separation of zero, which corresponds to the scenario in Figure 2.4a,b does not permit queue reductions through backhaul resource sharing. In contrast, the 40 s separation of the data bursts corresponding to the scenario in Figure 2.4c,d as well as Figure 2.5, does permit the sharing of the backhaul resources of the two operators. Thus, with the large $V = 1000$ flexibility parameter setting, substantial reductions of the average eNB queue lengths can be achieved for both operators for large separations of the data bursts.

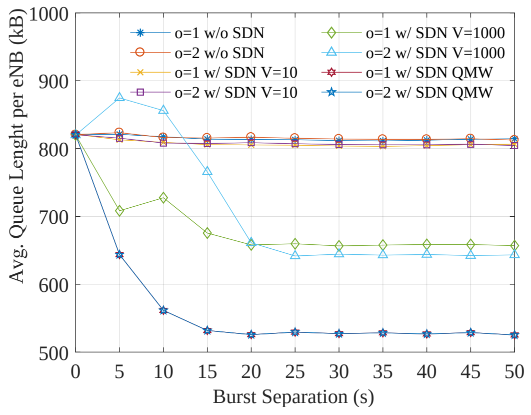


Figure 2.6: Queue Length in Kb at an Enb Averaged over Time and over The Enbs at a given Operator o as A Function of Separation of $o = 1$ and $o = 2$ Data Bursts in S; Figures 2.4 And 2.5 Consider a Burst Separation of 40 s.

In contrast, for the short separation times of 5 s and 10 s, we observe from Figure 2.6 that operator $o = 1$ achieves queue length reductions for the large $V = 1000$ setting, whereas the queue lengths for operator $o = 2$ increase. The eNBs at operator $o = 1$, which receives the earlier data burst, can still achieve queue length reductions by using some of the backhaul capacity of operator $o = 2$ to serve the data burst arriving to the eNBs at operator $o = 1$. However, the use of the $o = 2$ capacity by the $o = 1$ burst when the data burst to the eNBs at operator $o = 2$ arrives, slows down the service for the $o = 2$ burst, resulting in the $o = 2$ queue length increases observed in Figure 2.6. However, we observe from Figure 2.6 that the average of the curves for $o = 1$ and $o = 2$ for the $V = 1000$ setting is slightly below the corresponding queue length averages for operation without SDN or with the inflexible $V = 10$. Thus, the flexible sharing of backhaul capacity with $V = 1000$ does not “harm” the overall system compared to operation without sharing. The QMW benchmark gives yet lower queue length as QMW shares the bandwidths of the two operators without any constraints, i.e., corresponds to V approaching infinity (which would not enforce fair bandwidth allocations to operators).

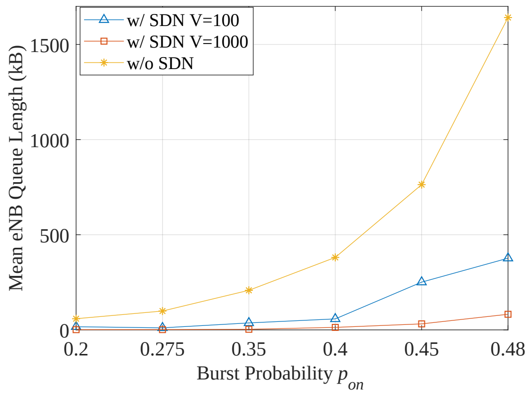
Impact of Random Traffic Bursts at Operators

The traffic model from Section 2.5.1 consisted of eNB Poisson packet traffic, whereby the eNB Poisson traffic rates at a given operator o were set to result in traffic bursts at prescribed times, as examined in Sections 2.5.2 through 2.5.2. We now generalize this traffic model to random traffic bursts as follows. The eNBs continue to generate independent Poisson packet traffic. The eNB Poisson packet rates at a given operator $o,\ o = 1, 2$, follow an independent two-state (on and off) Markov chain. In the on state, the 30 eNBs at a given operator generate an aggregate Poisson traffic rate of 20 Mbps; while in the off state, there is no packet generation. Both states have exponentially distributed random sojourn times with a mean of 10 s. The load is varied by adjusting the steady-state probability p_{on} of being in the on state and each simulation scenario is run for 1000 s.

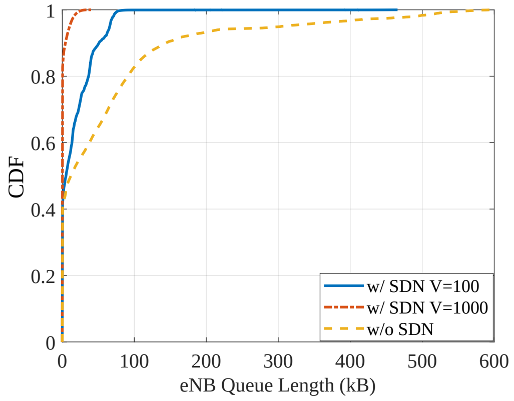
Figure 2.7a shows the mean eNB queue length as a function of the on-state probability p_{on}, i.e., effectively as a function of the load level. We observe from Figure 2.7 that the SDN control achieves eNB queue length reductions across the entire stable load range from a small on-state (burst) probability p_{on} up to a load level near the stability limit, which would be reached for $p_{\text{on}} = 0.5$. The eNB queue length reduction appears initially modest for small p_{on} because the bursts are rare for low p_{on}, i.e., the behavior is similar to the individual burst scenario considered in Figures 2.4 and 2.5 and thus can be cleared relatively quickly, even without SDN control. For increasing p_{on}, the bursts become more frequent, the eNB queue backlogs increase and flexible SDN control with $V = 1000$ achieves substantial queue reductions compared to operation without SDN control and compared to a less flexible SDN control with $V = 100$.

Figure 2.7b–d show the cumulative distribution function (CDF) of the eNB queue occupancy for three load levels, represented by different p_{on}. We observe from Figure 2.7b–d that the CDF curves for SDN control reach the level of one within a much smaller span of eNB queue lengths than the operation without SDN control. For instance, for the medium load level $p_{\text{on}} = 0.35$, the CDF for the SDN control with $V = 100$ reaches one for a queue length of about 90 kB; whereas, operation without SDN control reaches a CDF level of one only for around 760 kB eNB queue length. Thus, the CDF results indicate vastly reduced variability of the eNB queue length with the SDN control as the SDN control reacts to the traffic bursts by actively re-allocating backhaul resources among the $O = 2$ considered operators.

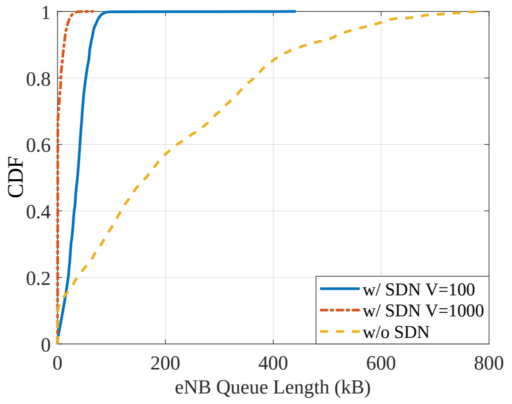
We observe from Figure 2.7d that for the low eNB queue occupancies in the range up to about 200 kB, operation without SDN control achieves higher probabilities of keeping the eNB queue lengths in this low range than SDN control with $V = 100$. This is mainly because the SDN control strives for fairness. If some eNB has a small queue occupancy compared to the other eNBs, the SDN control balances out the eNB queue occupancies via the centrally coordinated backhaul bandwidth allocation. In particular, the CDF curve for SDN control with $V = 100$ indicates that almost all the queue occupancies occur around the 200 to 250 kB range (the larger $V = 1000$ allows for flexible violations of the fairness constraint while sharing the backhaul bandwidth and thus achieves substantially lower queue occupancies). If some services do not want to be subjected to this fairness guided resource allocation and rather want priority service, then the priorities can be implemented through weights for their utilities.



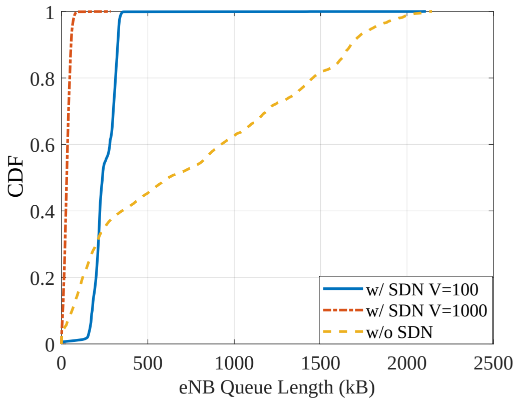
(**a**) Mean eNB queue vs. p_{on}



(**b**) CDF of eNB queue for $p_{\text{on}} = 0.2$



(**c**) CDF of eNB queue for $p_{\text{on}} = 0.35$



(**d**) CDF of eNB queue for $p_{\text{on}} = 0.45$

Figure 2.7: Mean and Cdf of Enb Queue Length in Kb for $o = 2$ Operators With Random Traffic Burst as a Function of Steady-state Probability p_{on} of Burst State.

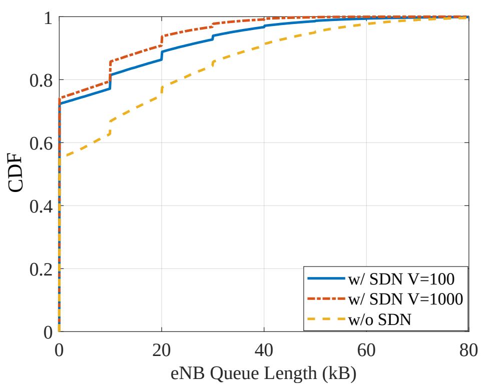
We considered only $O = 2$ operators sharing the overall backhaul resource Z in this section. When a larger number O of operators shares the overall resource, then the performance of the SDN control would further improve in accordance with the classical statistical multiplexing gains for many variable bitrate traffic streams sharing a common resource Smith and Whitt (1981); Liu *et al.* (2016a); Tonini *et al.* (2018). In this and the preceding evaluation scenarios, traffic bursts were generated on a per-operator basis, i.e., an independent Markov chain for each operator o, $o = 1, 2$, determined the

Poisson packet traffic rates (whereby the eNBs at an operator contributed equally to the operator traffic load). This per-operator traffic burst scenario reflects situations where traffic demands shift among operators, e.g., as large groups of users move among different nearby sub-networks, e.g., from lecture halls to restaurants (whereby the lecture halls and the restaurants have different operators) in a campus setting. In the next section, we consider per-eNB Markov chain modulated Poisson packet traffic rates that reflect situations where each eNB generates traffic bursts independently, e.g., when individual users conduct bursty Internet transactions, e.g., upload files.

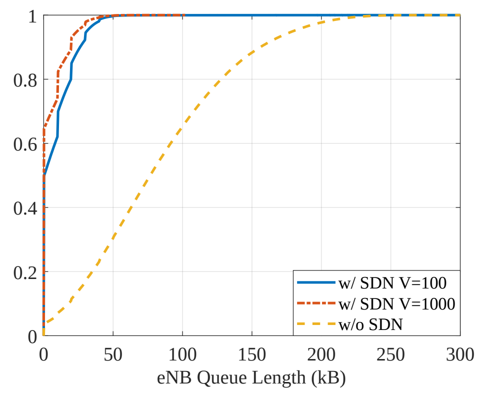
Impact of Random eNB Traffic Bursts

To evaluate the multi-layer multi-timescale approach for a large-scale network with independent eNB traffic bursts we modify the LayBack architecture from Section 2.5.1 as follows. We consider $O = 20$ operators, each with two GWs; each GW has five eNBs, for a total of 200 eNBs. The overall backhaul capacity still equals $Z = 20$ Mbps, but the operator backhaul capacity is $Z_o = 1$ Mbps. We consider this large network for random eNB Poisson packet traffic bursts generated according to an independent two-state (on and off) Markov chain for each of the 200 eNBs. An eNB generates 0.2 Mbps of Poisson packet traffic in the on state (fixed sojourn time of 10 s) and no traffic in the off state (exponentially distributed random sojourn time with mean 20 s for low load, 15 s for medium load, and 12 s for high load). The stationary distribution of visits to the on and off states is kept at 0.5 and 0.5. The resulting long-run traffic load for the medium load scenario is 16 Mbps (= 200·0.2 Mbps·(0.5·10 s)/(0.5·10 s + 0.5·15 s)), while the long-run traffic loads for the low and high load scenarios are 13.3 Mbps and 18.2 Mbps, respectively.

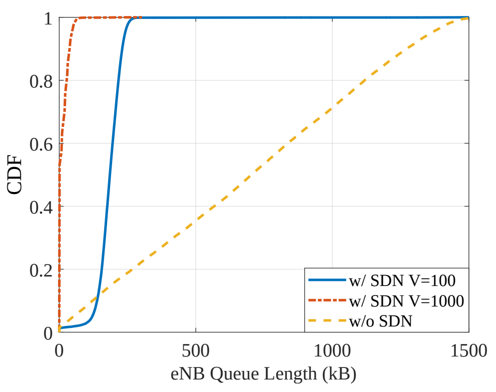
We observe from Figure 2.8a that for the light load scenario the SDN orchestrated backhaul bitrate allocation increases the probabilities for low eNB queue occupancies below 50 kB only relatively slightly compared to the operation without SDN. In contrast, we observe from Figure 2.8b vastly increased probabilities for low eNB queue occupancies below 50 kB with the SDN control compared to operation without SDN. More specifically, SDN control keeps the eNB queue lengths below 50 kB with a probability near one, whereas queue lengths below 50 kB occur only with a probability of about 0.4 without SDN control.



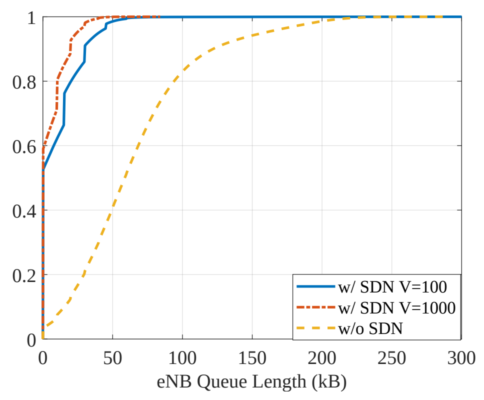
(**a**) Light load, 10 s eNB burst



(**b**) Medium load, 10 s eNB bursts



(**c**) High load, 10 s eNB bursts



(**d**) Medium load, 0.5 s eNB bursts

Figure 2.8: Cumulative Distribution Function (Cdf) of Enb Queue Length For Independent Enb Traffic Bursts with Various Load Levels for Long (10 s) Bursts and Medium Load for Short (.5 s) Bursts; Fixed parameters: $o = 20$ Operators, Each with Two Gateways (Each with Five Enbs).

If five or fewer eNBs at a given operator are in the on (traffic burst of 0.2 Mbps) state, then the aggregate traffic of the ten eNBs at the operator can be accommodated within the operator backhaul capacity of $Z_o = 1$ Mbps. If six or more eNBs at a given operator are in the traffic burst state and another operator has less than five eNBs in the traffic burst state, then the SDN control can share the backhaul resource among the operators. For the light traffic scenario (Figure 2.8a), occurrences of six or more simultaneous eNB traffic bursts at a given operator occur only occasionally; thus, there are relatively few opportunities for SDN control to share backhaul resources.

For the medium and high load levels (Figure 2.8b,c) it becomes increasingly likely that the aggregate load from the ten eNBs at a given operator exceeds the operator backhaul capacity Z_o. At the same time, due to the general Poisson process clumping behaviors, it is likely that the eNB traffic bursts "clump" at a given operator and exceed Z_o, while other operators have spare backhaul capacity. Thus, central SDN control of the backhaul capacity allocation can achieve substantial eNB queue length reductions compared to the operation without SDN.

Regarding the flexibility parameter V, we observe from Figure 2.8 that the benefit of the large $V = 1000$ relative to the smaller $V = 100$ increases as the load increases from the light/medium load to the high load. This is mainly because, $1/V$ is essentially the penalty for using spare bandwidth from other operators. For the light and moderate load levels, there are only relatively rare to moderately frequent occasions of bandwidth sharing; thus, there is no pronounced effect of V. For the high load (which corresponds to a long-run average overall backhaul use of 10/12), the assumption of the Lyapunov optimization is satisfied (i.e., the queues are stable), allowing the increased V to reduce the queue lengths.

Figure 2.8d considers the medium load scenario for short eNB bursts of 0.5 s (with corresponding 0.75 s off state sojourn time). Comparing Figure 2.8b,d, we observe that the operation without SDN achieves shorter eNB queue lengths with the short bursts in Figure 2.8d compared to the long bursts in Figure 2.8b. For example, an eNB queue length under 100 kB is achieved with over 0.8 probability in Figure 2.8d, but only less than 0.65 probability in Figure 2.8b. Intuitively, the shorter eNB traffic bursts create only smaller eNBs queue backlogs that are easier to clear with the limited operator bandwidth Z_o. We also observe from the comparison of Figure 2.8b,d that the gap between SDN control with $V = 100$ and with $V = 1000$ has slightly widened in Figure 2.8d, mainly due to the CDF curve for $V = 100$ reaching only lower values in Figure 2.8d compared to Figure 2.8b. This is primarily because the shorter burst in Figure 2.8d require more flexibility from the SDN control; however, the $V = 100$ control provides only limited flexibility and can therefore not perform quite as well as for the longer bursts in Figure 2.8b. Nevertheless, even though the gap between operation with SDN control vs. operation without SDN control has slightly shrunken in Figure 2.8d compared to Figure 2.8b, the SDN control still achieved substantial eNB queue length reductions.

2.6 Conclusions

This article has presented a multi-timescale approach for optimizing the sharing of backhaul resources in the SDN-based layered backhaul (LayBack) network architecture. Through primal dual decomposition and Lyapunov drift techniques we decomposed the traditionally centralized SDN resource management into a distributed management model. The distributed resource management accommodates realistic signaling propagation delays by conducting optimization computations at the higher gateway and SDN orchestrator layers at slower timescales compared to the fast-timescale operation at the eNB radio nodes. The distributed optimization is also highly scalable as only slow timescale optimizations of the sharing of the backhaul resources among multiple operators are performed at the central SDN orchestrator; the finer-grained faster timescale resource allocations to the individual eNB radio nodes and various gateway nodes are performed at the lower layers of the multi-layered multi-timescale optimization.

Our numerical evaluations for backhaul example networks have quantified the performance characteristics of the described multi-timescale backhaul resource optimization. We found that the SDN controlled sharing of the backhaul resources among operators can significantly reduce the queue lengths at the radio nodes, e.g., the eNBs that serve the upstream traffic from the wireless end users, compared to an optimization without SDN controlled resource sharing.

There are many interesting directions for future research on optimizing the backhaul in wireless networks. This present case study has focused on demonstrating the feasibility of a multi-timescale optimization with a specific example optimization methodology (gradient descent combined with Lyapunov drift-plus-penalty method) in a specific configuration of the LayBack backhaul network architecture. Future research should examine how wireless backhaul network architectures should be dimensioned, e.g., how many layers and how many nodes should be in a given layer for a range of anticipated end-device densities and mobility patterns over the geographic area covered by the wireless backhaul network architecture, so as to best support the optimization processes for resource allocation. With the emergence of multi-access edge computing (MEC) it may become important to widen the scope of resource allocation optimization to cover communication, caching (storage), and computation (e.g., virtual machine compute processing) resources Wang *et al.* (2017c); Xiang *et al.* (2019). Moreover, the suitability of the various types of optimization methodologies for the resource allocation in wireless backhaul networks should be broadly studied and compared. The comparison should consider both the optimization performance as well as the practical operational aspects, e.g., simplicity and computation resource usage. Another important future research aspect is robustness and reliability of the wireless backhaul network. Emerging cyber-physical systems, such as medical devices that provide critical diagnostics and continuous therapeutic interventions to humans going about their daily lives Ogudo *et al.* (2019) as well as networked vehicular systems Arena and Pau (2019); Santa *et al.* (2019); Storck and Duarte-Figueiredo (2019), such as the transportation systems in smart cities, require uninterrupted connectivity with high quality of service levels. Wireless backhaul networks need multiple redundant connectivity paths that provide fail-over functionalities in case of failures Frascolla *et al.* (2019); Gazit and Messer (2018); Tran *et al.* (2018). Future backhaul resource optimization needs to account for and route among these multiple connectivity paths (e.g., with SDN support Guck *et al.* (2018)) and optimally allocate resources during normal operation as well as after various failure scenarios. Furthermore, it would be of interest to implement the SDN controlled backhaul resource sharing in SDN testbeds to verify the resource sharing performance characteristics in real operational networks.

Chapter 3

HARDWARE ACCELERATION

3.1 Introduction

3.1.1 Trend to Run Softwarized Network Functions on General-Purpose Computing (GPC) Platforms

Traditionally, the term "network function (NF)" applied primarily to functions of the lower network protocol layers, i.e., mainly the data link layer (e.g., for the data link layer frame switching NF, virtual local area network NF, and medium access control security NF) and the network layer (e.g., for the datagram routing NF and Internet Protocol firewall NF). These low-level NFs were usually executed in specially designed dedicated (and typically proprietary) networking equipment, such as switches, routers, and gateways. Recently, the definition of an NF has been broadened to describe networking related tasks spanning from low-level frame switching and Internet Protocol (IP) routing to high-level cloud applications Ballani *et al.* (2015); Kozat *et al.* (2020); Mijumbi *et al.* (2015). The area of networking currently undergoes an unprecedented transformation in moving towards implementing NFs as software entities—so-called "softwarized NFs"—that run on General-Purpose Computing (GPC) platforms and infrastructures as opposed to dedicated networking equipment hardware.

In order to motivate this survey on hardware-accelerated platforms and infrastructures for softwarized NFs, we briefly introduce the basic concepts of softwarized NFs, including their computation and management on GPC platforms and infrastructures, in the following paragraphs. We then explain the need for hardware-acceleration of softwarized NFs on GPC platforms and infrastructures in Section 3.1.2, followed by an overview of the contributions of this survey in Section 3.1.3.

Network Functions (NFs) and Network Function Virtualization (NFV)

The term "Network Function (NF)" broadly encompasses the compute operations (both logical [e.g., bitwise AND or OR] and mathematical scalar and vector [e.g., integer and floating point arithmetic]) related either directly or indirectly to data link layer (Layer 2) frames, network layer (Layer 3) datagrams or packets, and network application data (higher protocol layers above Layer 3). For instance, a packet filter is a direct logical NF that compares header data to allow or block packets for further processing, while a jitter and latency estimator function is an example of an indirect arithmetic NF. An NF that requires dedicated processing with a strict deadline, e.g., an NF to verify a medium access control (MAC) frame error through a Cyclic Redundancy Coding (CRC) check, is preferably implemented as a hardware component. On the other hand, an NF with relaxed timing requirements, e.g., TCP congestion control, can be implemented as a software entity.

The push towards "softwarized NFs" is to reduce the hardware dependencies of NFs for function implementation so as to maximize the flexibility for operations, e.g., to allow for the flexible scaling and migration of NF services. Softwarized NFs enable compute operations to be implemented as generic executing programs in the form of applications that can be run on a traditional OS or isolated environments, such as Virtual Machines (VMs) Clayman *et al.* (2014) or containers Anderson *et al.* (2016), on GPC platforms. Analogous to the broad term "Network Function (NF)", the term "Network Function Virtualization (NFV)" broadly refers to NF implementation as a virtualized entity, typically as an application (which itself could run inside a container), and running inside a VM (see Fig. 3.1). Thus, NFV is an implementation methodology of an NF; while the term NF broadly refers to compute operations *related to* general packet processing. Moreover, the term “Virtual Network Function (VNF)” refers to an NF that is implemented with the NFV methodology.

Role of Software Defined Networking (SDN) in the Management of NFs

Software Defined Networking (SDN) Amin *et al.* (2018); Cox *et al.* (2017); Farhady *et al.* (2015); Kaljic *et al.* (2019) is a paradigm in which a logically centralized software entity (i.e., the SDN controller) defines the packet processing functions on a packet forwarding node. The notion of centralized decision making for the function implementation and configuration of forwarding nodes implies that the network control plane (which makes the decisions on the packet processing) is decoupled from the network data plane (which forwards the packets). Extending the principles of SDN from forwarding nodes to the broad notion of compute nodes can achieve more flexibilities in the deployment of NFs on GPC platforms in terms of scalability and management Alberti *et al.* (2019); Li and Chen (2015). More precisely, SDN can be applied for two primary purposes: *i*) macro-scale NF deployments, where the decisions involve selecting a specific platform for NF deployments based on decision factors, such as physical location, capabilities, and availability, and *ii*) micro-scale NF deployments, where the decisions involve reconfiguring the NF parameters during on-going operations based on run-time requirements, such as traffic loads, failures and their restoration, as well as resource utilization.

Compute Nodes for Running NFs

In general, the compute nodes running the NFs as applications (VMs and containers) can be deployed on platform installations ranging from large installations with high platform densities (e.g., cloud and data-centers) to distributed and singular platform installations, such as remote-gateways, clients, and mobile nodes. The cloud-native approach Roseboro (2016) is the most common method of managing the platform installations for the deployment of NFs that are centrally managed with SDN principles. While the cloud-native approach has proven to be efficient for resource management in cloud and data center deployments of NFs, the applicability of the cloud-native approach to remote-gateways, clients, and mobile nodes is yet to be investigated Shah *et al.* (2020).

The wide-spread adoption of Multi-Access Edge Computing (MEC) Abbas *et al.* (2017) with cloud-native management is accelerating the trend towards softwarized NFs, which run on GPC platforms. The MEC aims to deliver low-latency services by bringing computing platforms closer to the users Liu and Zhang (2018); Mehrabi *et al.* (2019); Heuchert *et al.* (2022); Mehrabi *et al.* (2021); Tang and Hu (2020); Xiang *et al.* (2019, 2022); Doan *et al.* (2021); Wang *et al.* (2020); **?**. A key MEC implementation requirement is to inherit the flexibility of hosting a variety of NFs as opposed to a specific dedicated NF. A GPC platform inherently provides the flexibility to implement NFs as software entities that can easily be modified and managed, such as applications, Virtual Machines (VMs), and containers Zhang *et al.* (2018). In a typical MEC node deployment, the GPC platform is virtualized by a hypervisor Dinakar (2019), e.g., Linux Kernel-based Virtual Machine (KVM), Microsoft HyperV, or VMware ESXi, and then NFs are instantiated as a VM or container managed by the hypervisor. The flexibility of an MEC is achieved by the process of migrating applications, VMs, and containers to different locations by an orchestration function Wang *et al.* (2018).

Management of NFs

The NF deployment on a compute node (i.e., physical platform) is typically managed through a logically centralized decision making entity referred to as "Orchestrator". Based on SDN principles, the orchestrator defines and sends orchestration directives to the applications, VMs, and containers to run on compute nodes Alsaeedi *et al.* (2019); Binsahaq *et al.* (2019); Li and Chen (2015); Kellerer *et al.* (2019); Shantharama *et al.* (2018b); Wang *et al.* (2019b); Zilberman *et al.* (2015). OpenStack Kavanagh (2015) and Kubernetes Martí Luque (2019); Kouchaksaraei and Karl (2019) are the mostly commonly adopted dedicated orchestration frameworks in the cloud and data-center management of resources and applications, including VMs and containers. In addition to flexibility, the softwarization and virtualization of NFs can reduce CAPEX and OPEX of the network operator. In particular, the network operator can upgrade, install, and configure the network with a centralized control entity. Thus, MEC and virtualization are seen as key building blocks of future network infrastructures, while SDN enables efficient network service management.

3.1.2 Need for NF Hardware Acceleration on GPC Platform

The NF softwarization makes the overall NF development, deployment, and performance characterization at run time more challenging Li and Chen (2015). Softwarized NFs rely on GPC central processing units (CPUs) to accomplish computations and data movements. For instance, data may need to be moved between input/output (I/O) devices, e.g., Network Interface Cards (NICs), and system memory. However, the GPC platforms, such as the Intel® x86–64 Arafa *et al.* (2019) and AMD® Lepak *et al.* (2017) CPU platforms, are not natively optimized to run NFs that include routine packet processing procedures across the I/O path Cerović *et al.* (2018); Garay *et al.* (2016); Nobach and Hausheer (2015); Ordonez-Lucena *et al.* (2017). The shortcomings of GPC platforms for NF packet processing have motivated the development of a variety of software and hardware acceleration approaches, such as the Data Plane Development Kit (DPDK) Intel Corp. (2014), Field Programmable Gate Array (FPGA), Graphics Processing Unit (GPU), and Application Specific Integrated Circuit

(ASIC) Nurvitadhi *et al.* (2016), to relieve the hardware CPU from compute-intensive tasks generated by the NFs, such as data link layer frame switching, IP look-up, and encryption Pongrácz *et al.* (2013).

The deployment of softwarized NFs on GPC platforms achieves a high degree of flexibility. However, it is important to note that critical NF functionalities can be compromised if the hardware and software functional limitations as well as operational characteristics and capabilities are not carefully considered. Generally, the dynamic CPU characteristics can vary over time. For instance, the cache coherency during memory accesses can introduce highly variable (non-deterministic) latencies in NF packet processing Gallenmüller *et al.* (2015). Moreover, the CPU power and thermal characteristics can vary the base operating frequency, introducing variable processing time behaviors Makineni and Iyer (2003); Emmerich *et al.* (2015); Chou and Bhuyan (2015). Therefore, the softwarization of NFs must carefully consider the various performance implications of NF acceleration designs to ensure appropriate performance levels of NFs deployed on hardware-accelerated GPC platforms. These complex NF performance implications of hardware-accelerated GPC platforms and infrastructures motivate the comprehensive survey of this topic area so as to provide a foundation for the further advancement of the technology development and research on hardware-accelerated platforms and infrastructures for NFs.

3.1.3 Contributions and Organization of this Survey

In order to inform the design of hardware acceleration for the processing of softwarized NFs on GPC platforms, this article comprehensively surveys the relevant existing enabling technologies and research studies. Generally, the processing of a software application task is essentially achieved by a set of hardware interactions. Therefore, understanding hardware features provides a key advantage in the design of software applications. In contrast to a generic software application, an NF involves typically extensive I/O interactions, thus, the NF compute processing largely depends on hardware support to achieve high throughput and short latency for NF packet processing. However, the NF implementation relies not only on I/O interactions for packet transmission and reception, but also requires memory for tunneling and encapsulation, storage for applications (e.g., store-and-forwarding of media), as well as computing (e.g., for cryptography and compression).

This survey provides an authoritative up-to-date survey of the hardware-accelerated platforms and infrastructures that speed up the processing of NF applications. The term "platform" as used in this survey article consolidates all the physical hardware components that can be used to build a complete system to support an Operating System (OS) to drive an application. The platform includes the Basic Input Output System (BIOS), CPU, memory, storage, I/O devices, dedicated and custom accelerators, switching fabric, and power management units. The term "infrastructure" corresponds to the end-to-end connectivity of platforms, such as network components, switches, Ethernet, and wireless links. Platform and infrastructure together constitute a complete hardware framework to support an NF.

Despite the wealth of surveys on NFs and their usage in a wide variety of networking contexts, to the best of our knowledge, this present survey article is the first comprehensive survey of hardware-accelerated platform and infrastructure technologies and research studies for the processing of NFs. We give an overview of the related surveys in Section 3.1.4 and provide background on the processing of NFs in Section 3.2. Section 3.3 comprehensively surveys the relevant enabling technologies for hardware-accelerated platforms and infrastructures for processing NFs, while Section 3.4 comprehensively surveys the related research studies. For the purpose of this survey, we define enabling technologies as designs, methodologies, and strategies that are currently available in the form of a product in the market place; enabling technologies are typically developed by industry or commercially oriented organizations. On the other hand, we define research studies as investigations that are primarily conducted to provide fundamental understanding and insights as well as new approaches and methodologies that aim to advance the overall field; research studies are primarily conducted by academic institutions, such as universities and research labs.

Section 3.3 classifies the enabling technologies according to the relevant hardware components that are needed to support the processing of NFs, namely the CPU, interconnects, memory, as well as custom and dedicated accelerators on the platforms; moreover, Section 3.3 surveys the relevant infrastructure technologies (SmartNICs and Non-Transparent Bridging). Section 3.4 categorizes the research studies into studies addressing the computer architecture, interconnects, memory, and accelerators on platforms; moreover, Section 3.4 surveys infrastructure research on SmartNICs. Section 3.5 summarizes the main open challenges for hardware-accelerated platforms and infrastructures for processing softwarized NFs and Section 3.6 concludes this survey article.

3.1.4 Related Surveys

This section gives an overview of the existing survey articles on topics related to NFs and their processing and use in communication networks. Sections 3.1.4 through 3.1.4 cover topic areas that border on our central topic area, i.e., prior survey articles on topic areas that relate to our topic area in a wider sense. Section 3.1.4 focuses on prior survey articles that cover aspects of our topic area. Section 3.1.4 highlights our original survey coverage of hardware-accelerated platforms and infrastructures for NFs with respect to prior related survey articles

Softwarization of Network Functions (NFs)

The NF softwarization can be achieved in different forms, i.e., an NF can be implemented as a software application, as a Virtual Machine (VM), or as a container image. The concept of implementing an NF as a VM has been commonly referred to as Virtualized Network Function (VNF), and Network Function Virtualization (NFV) as a broader term for the technology of implementing, deploying, and managing the VNFs. In general, the NFV concept has been widely discussed in the survey literature Han *et al.* (2015); Jain and Paul (2013); Rehman *et al.* (2019). The traditional challenges of NFV deployment are associated with the virtualization process of NFs, such as overhead, isolation, resource allocation, and function management Wood *et al.* (2015). Herrera et al. Herrera and Botero (2016) have discussed the resource allocation and placement of applications, VMs, and containers on GPC platforms. More specifically, Herrera et al. Herrera and Botero (2016) have surveyed different schemes for the embedding of virtual networks over a substrate network along with the chaining of NFs.

The deployment of an NF as a software application, VM, or container image in the cloud and public networks poses critical security challenges for the overall NFV service delivery. The security aspects and challenges of NFs have been discussed by Yang et al. Yang and Fung (2016) and Farris et al. Farris *et al.* (2018) for threats against NFs on Internet of Things (IoT) networks; while threat-based analyses and countermeasures for NFV security vulnerabilities have been discussed by Pattaranantakul et al. Pattaranantakul *et al.* (2018). Furthermore, Lal et al.Lal *et al.* (2017) have presented best practices for NFV designs against security threats.

Software Defined Networking (SDN) for NFs

Software Defined Networking (SDN) provides a centralized framework for managing multiple NFs that are chained together to form a network Service Function Chain (SFC) Bhamare *et al.* (2016); Li and Qian (2016); Miotto *et al.* (2019). SDN controllers can be used to monitor the resources across multiple platforms to allocate resources for new SFCs, and to manage the resources during the entire life time of a service. The SDN management strategies for NFs have been summarized by Li et al. Li and Chen (2015). SDN also provides a platform for the dynamic flow control for traffic steering and the joint optimization of resource allocation and flow control for NFV. The main challenges of SDN-based management is to achieve low control overhead and latency while ensuring the security during the reconfiguration Pattaranantakul *et al.* (2019). In contrast to surveys of independent designs of SDN and NFV, Bonfim et al. Bonfim *et al.* (2019) have presented an overview of integrated NFV/SDN architectures, focusing on SDN interfaces and Application Programming Interfaces (APIs) specific to NFV management.

Network Function Virtualization (NFV) and Network Slicing

5th Generation (5G) Ghosh *et al.* (2019); Gupta and Jha (2015); Sharma *et al.* (2020); Rischke *et al.* (2021) is a cellular technology that transforms the cellular infrastructure from hardware-dependent deployment to software-based hardware-independent deployment. 5G is envisioned to reduce cost, lower the access latencies, and significantly improve throughput as compared to its predecessors Nasrallah *et al.* (2018); Parvez *et al.* (2018); Sachs *et al.* (2018). VNFs are an integral part of the 5G infrastructure as NFs that realize the 5G based core network functionalities are implemented as VNFs. In addition to NFV, 5G also adopts SDN for the centralized management of the NFV resources. Yang et al. Yang *et al.* (2015) have presented a survey of SDN management of VNFs for 5G networks, while Nguyen et al. Nguyen *et al.* (2017); Lucani *et al.* (2018); Tasdemir *et al.* (2021); Gabriel *et al.* (2018); Wunderlich *et al.* (2017) have discussed the relative benefits of different SDN/NFV-based mobile packet core network architectures. Bouras et al. Bouras *et al.* (2017) have discussed the challenges that are associated with SDN and NFV based 5G networks, such as scalability and reliability. Costa et al. Costa-Perez *et al.* (2017b) have summarized efforts to homogeneously coordinate resource allocation based on SDN and NFV across both fronthaul and backhaul networks in the 5G infrastructure.

In conjunction with SDN and NFV, the technique of network slicing provides a framework for sharing common resources, such as computing hardware, across multiple VNFs while isolating the different network slices from each other. Afolabi et al. Afolabi *et al.* (2018) have surveyed the softwarization principles and enabling technologies for network slicing. As discussed in the survey by Foukas et al. Foukas *et al.* (2017) for VNFs in 5G, for the design of 5G infrastructure, network slicing provides an effective management and resource allocation to multiple tenants (e.g., service providers) on the same physical infrastructure. A more general survey on network slicing for wireless networks (not specific to 5G wireless networks) has been presented by Richart et al. Richart *et al.* (2016b). The surveys Barakabitze *et al.* (2020); Ben Azzouz and Jamai (2019); Bojkovic *et al.* (2019); Su *et al.* (2019) have discussed network slicing and the management of resources in the context of 5G based on both SDN and NFV.

NFV in Multi-Access Edge Computing (MEC)

In contrast to the deployment of VMs and containers in cloud networks, fog and edge networks bring the network services closer to the users, thereby reducing the end-to-end latency. Yi et al. Yi *et al.* (2015) have presented a survey of NFV techniques as applied to edge networks. Some of the NFV aspects that are highlighted by Yi et al. Yi *et al.* (2015) in the context of fog and edge networks include scalability, virtualization overhead, service coordination, energy management, and security. As an extension of fog and edge networks, Multi-Access Edge Computing (MEC) generalizes the compute infrastructure at the edge of the access network. A comprehensive MEC survey has been presented by Tanaka et al. Tanaka *et al.* (2018), while the role of NFV in MEC has been surveyed by Taleb et al. Taleb *et al.* (2017). The use of both SDN and NFV provides strategies for effective management of MEC resources in edge networks as described by Baktir et al. Baktir *et al.* (2017) and Blanco et al. Blanco *et al.* (2017).

NFV Orchestration

NFV service orchestration involves the management of software applications, VMs, and containers which implement NFs. The NFV management constitutes the storage of VNF images, the allocation of resources, the instantiation of VNFs as runtime applications, the monitoring of the NFV performance, the migration of the VNFs between different hosts, and the shutting down of VNFs at the end of their life time. De et al. de Sousa *et al.* (2019) have presented a survey of various methods for managing NFV services. In contrast to NFV management, the orchestration of service function chaining (SFC) adds more complexity since an SFC involves the management of multiple VNFs for a single network service. The SFC complexities, such as compute placement, resource monitoring, and flow switching have been outlined in a survey article by Mechtri et al. Mechtri *et al.* (2017). Duan et al. Duan *et al.* (2016) have presented a survey on SDN-based orchestration studies for NFV management.

Acceleration of NFs

NFs typically require the routine processing of packets involving intense Input/Output (I/O) activities into and out of the compute platform Pitaev *et al.* (2018). Since GPC platforms are not fundamentally designed for packet processing, GPC platforms require additional acceleration techniques for effective high-speed packet processing Kim and Shao (2018). Linguaglossa et al. Linguaglossa *et al.* (2019) have provided a tutorial introduction to the broad NFV field and the overall NFV ecosystem, including tutorial introductions to software acceleration (inclusive of the related ecosystem of software stacks) and hardware acceleration of NFV. The hardware acceleration section in Linguaglossa *et al.* (2019) focuses mainly on a tutorial introduction to the general concept of hardware offloading, mainly covering the general concepts of offloading to commodity NICs and SmartNICs; an earlier brief tutorial overview of hardware offloading had appeared in Woesner *et al.* (2018). However, a comprehensive detailed survey of specific hardware offloading technologies and research studies is not provided in Linguaglossa *et al.* (2019). Zhang Zhang (2020) has presented an overview of NFV platform designs; whereby, Zhang defines the term "NFV platform" to broadly encompass all hardware and software components involved in providing an NFV service (in contrast, we define the term "platform" to only refer to the physical computing entity). Zhang Zhang (2020) mainly covers the VNF software and management aspects, i.e., Management and Orchestration (MANO) components Yousaf *et al.* (2019), that are involved in NFV deployments. Zhang Zhang (2020) covers hardware acceleration only very briefly, with only about ten references in one paragraph. In contrast to Linguaglossa *et al.* (2019) and Zhang (2020), we provide a comprehensive survey of hardware-accelerated platforms and infrastructures for NF processing. We comprehensively cover the technologies and research studies on the hardware acceleration of CPUs, interconnects, and memory, as well as the accelerator devices on platforms, and furthermore the hardware acceleration of infrastructures (which in our classification encompass SmartNICs) that benefit NF processing.

FPGAs can be programmed with different functions, thereby increasing design flexibility. FPGA-based acceleration in terms of application performance is limited by the transistor-gate density and CPU-to-I/O transactions. Additionally, the FPGA configuration time is relatively longer than running a compiled executable on a GPU or CPU. While GPUs are beneficial for running numerous parallel, yet simple computations, the FPGA advantages include the support for complex operations which can be a differentiating factor for compute-intensive workloads Li *et al.* (2016a). NF applications that require specialized compute-intensive functions, such as security, can achieve superior performance with FPGAs as compared to GPUs and CPUs. Niemiec et al. Niemiec *et al.* (2020) have surveyed FPGA designs for accelerating VNF applications covering the use cases that require compute-intensive functions, such as IPSec, intrusion detection systems, and deep packet inspection Xu *et al.* (2016). The Niemiec et al. survey Niemiec *et al.* (2020) includes FPGA internals, virtualization and resource slicing of FPGA, as well as orchestration and management of FPGA resources specifically for NFV deployments. In contrast, our survey includes FPGAs operating in conjunction with CPUs, i.e., FPGAs as platform capability enhancements, to assist in accelerating general NF applications (that are not limited to NFV deployments, but broadly encompass arbitrary NF applications, including e.g., bare-metal applications).

3.2 Background on NF Implementation

In this section we provide background on Network Functions (NFs), discuss various forms of NF implementation, and common acceleration strategies. An NF is a compute operation on a packet of an incoming traffic stream in a compute host. NF examples

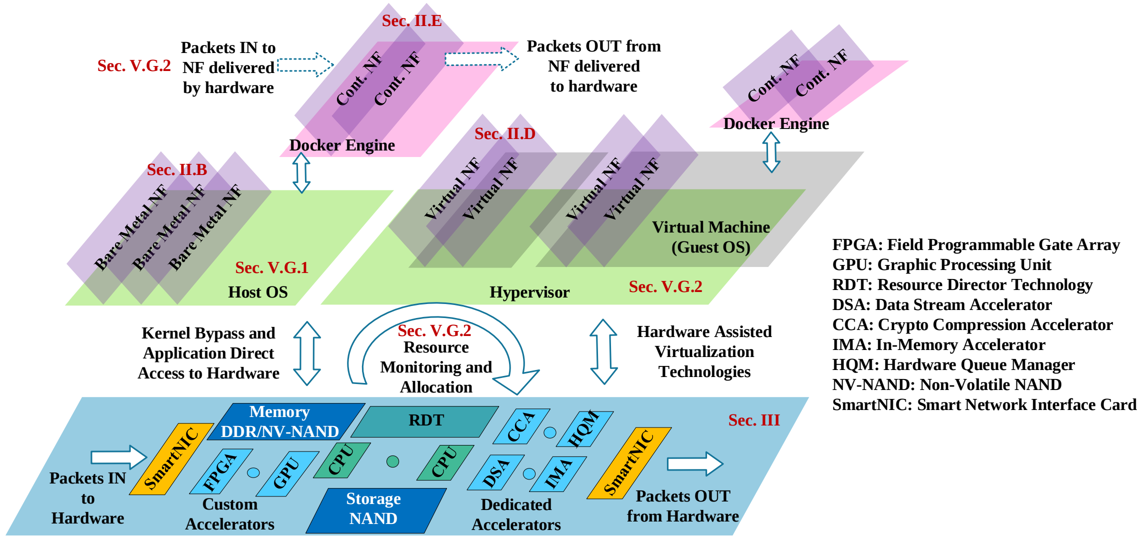

Sec. II.E
Sec. V.G.2
Packets IN to NF delivered by hardware
Cont. NF
Cont. NF
Packets OUT from NF delivered to hardware
Cont. NF
Cont. NF
Docker Engine
Docker Engine
Sec. II.D
Sec. II.B
Bare Metal NF
Bare Metal NF
Bare Metal NF
Virtual NF
Virtual NF
Virtual NF
Virtual NF
Virtual Machine (Guest OS)
Sec. V.G.1
Host OS
Hypervisor
Sec. V.G.2
FPGA: Field Programmable Gate Array
GPU: Graphic Processing Unit
RDT: Resource Director Technology
DSA: Data Stream Accelerator
CCA: Crypto Compression Accelerator
IMA: In-Memory Accelerator
HQM: Hardware Queue Manager
NV-NAND: Non-Volatile NAND
SmartNIC: Smart Network Interface Card
Kernel Bypass and Application Direct Access to Hardware
Sec. V.G.2 Resource Monitoring and Allocation
Hardware Assisted Virtualization Technologies
Memory DDR/NV-NAND
RDT
CCA
HQM
Sec. III
SmartNIC
FPGA
GPU
CPU
CPU
DSA
IMA
SmartNIC
Packets IN to Hardware
Custom Accelerators
Storage NAND
Dedicated Accelerators
Packets OUT from Hardware

Figure 3.1: Illustration of Gpc Platform Hardware to Process Network Functions (Nfs). An Nf Can Be Implemented as a Bare Metal Nf, Application Nf (Not Shown), Virtual Nf (Vnf), or Container Nf (Cnf).

range, for instance, from a simple IP header look-up for packet forwarding to complex operations involving security negotiations of an end-to-end connection. NFs can also be indirect functions, such as statistical analysis of traffic, network port management, and event monitoring to detect a Denial-of Service (DoS) attack. Traditionally, an NF is implemented with dedicated hardware and software components (see Sec. 3.2.1). Recently, with the softwarization of NFs, the trend is towards implementing NFs as software running on General-Purpose Computing (GPC) platforms. A softwarized NF running on a GPC platform can be designed as: a bare-metal (BM) implementation on a native OS (as user application) or as a part of the OS (as kernel module) (see Sec. 3.2.2), as application running on an OS, i.e., as user application, or as kernel module as part of the OS (see Sec. 3.2.3), as Virtual Machine (VM) on a hypervisor (see Sec. 3.2.4), or as container running on a container engine (see Sec. 3.2.5). Brief background on general acceleration strategies for NFs running on GPC platforms is given in Sec. 3.2.6.

Before we delve into the background on NFs, we give a brief overview of the terminology used for structures on GPC processor chips. The term “package” refers to several hardware components (e.g., CPU, memory, and I/O devices) that are interconnected and packed to form a system that is integrated into a single unit with metallic finishing for physical mounting on a circuit board. That is, a package is a typical off-the-shelf product that is available as a full hardware module and that can be plugged into a server-chassis. A package is often a combination of CPU and non-CPU components, such as memory (DRAM modules), I/O devices, and accelerators. A GPC platform consists typically of multiple packages.

Typically, a commercially available “chip”, such as a processor chip or a RAM chip, is a full System-on-Chip (SoC). A GPC processor chip is typically, in the socket form-factor. We may therefore use the terminology “CPU chip” and “socket” interchangeably; synonymous terminologies are “CPU socket” and “CPU slot”. Generally, a package contains only a single CPU socket (plus other non-CPU components). Also, a given CPU socket consists generally of only a single CPU chip, which can contain multiple dies, and each die can consist of multiple CPU cores. In particular, a single CPU chip consists typically of multiple interconnected CPU dies. A die is a single silicon design entity that is etched in one shot during fabrication. On a CPU chip, there can be multiple dies interconnected through silicon vias or metallic wires.

3.2.1 Dedicated Hardware Based NF Implementation

Overview

The traditional implementation of an NF was through the design of dedicated hardware and software, such as off-the-shelf network switches, routers, and gateways Velte and

Velte (2013); Ball *et al.* (1995); Chen *et al.* (2003). Hardware based systems are driven by an embedded software (firmware, microcode), with microprocessor, microcontroller, Digital Signal Processor (DSP), or Application-Specific Integrated Circuit (ASIC) modules. Embedded software for hardware control is generally written in low-level languages, such as C or assembly. The designs are tightly focused on a specific prescribed (dedicated) task. For instance, if the design is to route packets, the embedded hardware and software components are programmed to route the packets. Hence, dedicated hardware NF implementations are fixed implementations that are designed to perform a dedicated task, except for the management configuration of the device and NF.

Benefits

Implementation with dedicated hardware and software achieves the best performance for the dedicated task due to the constrained nature of task processing. As opposed to the processes and task scheduling in an OS, processes running on dedicated hardware use static (fixed) resource allocation, thereby achieving a deterministic packet processing behavior. Dedicated NF hardware units are also energy efficient as no processing cycles are wasted for conversions, e.g., privileges of execution, modes of operation, and address translations, in OSs and hypervisors.

Shortcomings

A main shortcoming of NF hardware implementation is very limited flexibility. Reconfigurations require additional efforts, such as intervention by a network administrator. Moreover, NF hardware (HW) is typically locked into vendors due to proprietary designs, procedures, and protocols. The overall cost of dedicated hardware products could be relatively high since the deployment and maintenance require specialized skills and specialized vendor assistance.

3.2.2 Bare-Metal (BM) NF Implementation

Overview

Hardware resources that directly host software tasks, e.g., applications, for computing and I/O without any additional abstraction (except for the OS that directly hosts the software task) are referred to as Bare-Metal (BM) hardware Hovemeyer *et al.* (2004). In contrast to BM hardware, the other forms of hardware include abstracted hardware (i.e., virtualized HW). In theory and practice, there can be multiple layers of abstraction, achieving nested virtualization Ben-Yehuda *et al.* (2010); Zhang *et al.* (2011). Abstraction of hardware resources reduces the complexity of operating and managing the hardware by the application which can help the application to focus on task accomplishment instead of managing the hardware resources. The BM implementation can provide direct access to hardware for configurability, reducing the overheads for computing and for hardware interactions for I/O. The application performance on BM as compared to abstracted hardware, i.e., on a VM or container, has been examined in Yamato et al. Yamato (2015).

Benefits

The BM implementation of NFs can achieve relatively higher performance as compared to NFs running on virtualized and abstracted environments Yamato (2015). The high BM performance is due to the low overhead during NF compute tasks. The instruction and memory address translations required by abstractions are avoided by BM implementations. The BM implementation also provides direct access to OS resources, such as the kernel, for managing the memory allocation, prioritizing the scheduling processing, and controlling I/O transactions.

Shortcomings

The BM implementation of an NF does not provide a secure and isolated environment to share the hardware resources with other NFs on the same BM. If multiple NFs run on the same BM hardware, multiple NFs can interfere with each other due to the contention for resources, such as CPU, cache, memory, and I/O resources, resulting in non-deterministic behaviors. Running a low number of NFs to avoid interference among NFs can result in low resource utilization. Hence, the management of applications could incur additional computing as well as a higher management cost. NF implementation on BM with hardware-specific dependencies can result in reduced scalability and flexibility.

3.2.3 Application and Kernel Based NF Implementation

Overview

In general, NFs are mainly deployed as applications which implement the overall packet processing functionality. In contrast to the NF implementation as a user-space application, NF tasks can also be embedded into the kernel as a part of the OS. Generally, there are two types of processes that are run by the OS on the CPU: *i*) applications that use the user-space memory region, and *ii*) more restrictive kernel (software) modules that use the kernel-space memory region. However, a kernel-based NF provides little or no control to the user for management during runtime. Therefore, NFs are mainly run as applications in the user-space execution mode in an OS.

The user-space has limited control over scheduling policies, memory allocation, and I/O device access. However, NFs in the user-space are given special permissions through kernel libraries and can access kernel-space resources (i.e., low level hardware configurations). Some NF applications, such as authentication, verification, and policy management, may not always require hardware interactions and special kernel-space access. Therefore, the design of NF applications should consider the hardware requirements based on the nature of the task, i.e., whether an NF is time-sensitive (e.g., audio packets), memory intensive (e.g., database management), or compute intensive (encryption/decryption). Some examples of high level NF applications with low resource dependencies are data validation, traffic management, and user authentication.

Benefits

Application based NFs have simple development, deployment, and management. Most NFs are designed and deployed as user-space application in an OS. User-space applications generally consume lower compute, memory, and I/O resources compared to abstraction and isolation based implementations, such as container and VMs.

Shortcomings

NF applications that are implemented in the user-space are vulnerable to security attacks due to limited OS protection. Also, user-space applications are not protected from mutual interference of other NF applications, thus there is no isolation among tasks, resulting in non-deterministic execution of NF tasks. Moreover, user-space applications fall short for networking tasks that require near real-time reaction as the requests propagate through memory regions and follow traditional interrupt mechanisms through I/O hardware.

3.2.4 Virtual Machine (VM) Based NF Implementation

Overview

To flexibly manage NFs with effective resource utilization and isolation properties, NFs can be implemented as an application running on a Virtual Machine (VM). A VM is typically implemented as a guest OS over a host OS. The host OS abstracts the hardware resources and presents a virtualized hardware to the guest OS. The software entity (which could be part of the host OS) that abstracts and manages the hardware resources is referred to as a hypervisor. An NF can then be implemented as a kernel module or as a user-space application on the guest OS. A host OS/hypervisor can support multiple guest OSs through sliced resource allocation to each guest OS, thus providing a safe virtual environment for the NF execution.

Benefits

VM based NF implementation provides a high degree of flexibility in terms of deploying and managing the NFs. Multiple instances of the same NF can be instantiated through duplication of VM images for scalability and reliability. VM images can also be transported easily over the network for the instantiation at a remote site. Additionally, multiple NFs can be hosted on the same host OS, increasing the effective resource sharing and utilization. A VM is a complete OS, and all the dependent software necessary for the execution of an NF application is built into the VM, which improves the compatibility across multiple host OSs and hypervisors.

Shortcomings

In general, the performance of an NF implemented as a VM is lower than BM and OS based implementation, since virtualization incurs both compute and memory overhead Yamato (2015). Since a VM is also a fully functional OS, the overall memory usage and execution processes are complex to design and manage as compared to a user-application based NF running on an OS without virtualization. NF software implementation issues are complex to trace and debug through multiple layers of abstraction. Deployment cost could be higher due to the need for specialized support for the VM management Strunk (2012).

3.2.5 Container based NF Implementation

Overview

The VM based NF implementation creates a large overhead for simple NFs, such as Virtual Private Network (VPN) authentication gateways. Scaling and migrating VMs requires large memory duplications, which result in overall long latencies for creating and transporting multiple VM instances. The concept of workload containerization originated for application management in data centers and the cloud to overcome the disadvantages of VMs Coutinho *et al.* (2015). Containers have been designed to create a lightweight alternative to VMs. A key difference between a VM and a container is that a container shares the host OS kernel resources with other containers, while a VM shares the hardware resources and uses an independent guest OS kernel. The sharing of host OS resources among containers is facilitated by a Container Engine (CE), such as Docker. NFs are then implemented as a user-space application running on a container Cziva and Pezaros (2017). The primary functions of a CE are:

i) Provides Application Programming Interfaces (APIs) and User Interfaces (UIs) to support interactions between host OS and containers.

ii) Container image management, such as storing and retrieving from a repository.

iii) Configuration to instantiate a container and to schedule a container to run on a host OS.

Benefits

The primary benefits of containerization are the ease of NF scalability and flexibility. Containers are fundamentally designed to reduce the VM management overhead, thus facilitating the creation of multiple container instances and transporting them to different compute nodes. Container based NFs support cloud-native implementation, i.e., to inherently follow the policies applied through a cloud management framework, such as Kubernetes. Containerization creates a platform for NFs to be highly elastic to support scaling based on the demand during run time, resulting in Elastic-Network Functions (ENFs) Szabo *et al.* (2015).

Shortcomings

Critical shortcomings of containerization of an NF are:

i) Containers do not provide the high levels of security and isolation of VMs.

ii) A container can run on BM hardware; whereas, a VM can run both on a hypervisor and on BM hardware.

iii) Only the subset of NF applications that support a modularized software implementation and have low hardware dependencies can be containerized.

iv) Containers do not provide access to the full OS environment, nor access to a Graphic User Interface (GUI). Containers are limited to a web-based user interface that provides simple hypertext markup language (HTML) rendering for applications that require user interactions, e.g., for visualizations and decisions based on traffic analytics.

3.2.6 Acceleration Strategies for NF Implementation

NF softwarization should carefully consider different design strategies as one design strategy does not fit all application needs. In addition to discussed software implementation designs (Sections 3.2.2– 3.2.5), we need to consider acceleration techniques to facilitate the NF application to achieve optimal performance in terms of overall system throughout, processing latency, resource utilization, energy, and cost, while preserving scalability and flexibility. Towards these goals, acceleration can be provided in either software or hardware.

Software Acceleration Methods

Overview Typically, an NF on a GPC infrastructure requires an application running on a traditional OS, such as Linux or Windows, whereby, an application can also be hosted inside a VM or container for abstraction, security, and isolation requirements. However, traditional OSs are not natively designed towards achieving high network performance. For instance, an OS network driver typical operates in interrupt mode. In interrupt mode, a CPU is interrupted only when a packet has arrived at the Network Interface Card (NIC), upon which the network driver process running on the CPU executes a subroutine to process the packet waiting at the NIC. If the CPU is in a power-saving deep sleep state due to inactivity, waking the CPU would take several cycles which severely lengthens the overall packet processing latency. An alternative to the interrupt mode is polling. However, polling of the NIC would significantly reduce the ability of the CPU to perform other tasks. Thus, the interrupt mode incurs relatively long latencies, while keeping the CPU and power utilization low. However, the interrupt mode generally does not maximize the overall throughput (total packets processed by the CPU per second), which requires the batching of packets and is more readily achieved with polling Barach *et al.* (2018).

Some of the examples of software acceleration strategies are:

i) Polling strategies of I/O devices for offloading task completions and I/O requests.

ii) Continuous memory allocation, and reduction in memory copies between processes and threads.

iii) Reduced virtual to physical address translations.

iv) Maintaining cache coherency during memory accesses.

iv) Scheduling strategies for resource monitoring and allocation.

Benefits One of most prominent benefits of software acceleration is the low cost of adoption in the market, which also reduces the development to deployment cycle time. Software acceleration requires only very small or no modifications of the existing infrastructure. Software optimizations also pave the way to an open source architecture model of software development. The overall development and deployment of software acceleration reduces the complexity and need for sophisticated traditional hardware acceleration designs; and maximizes the performance and utilization of existing hardware infrastructures.

Shortcomings Software acceleration may not provide the best possible system throughput as compared to hardware acceleration to fully utilize the system capacity as the software overhead may cause bottlenecks in the system, e.g., for memory and I/O device accesses. Software implementation also increases the overall energy consumption for a given acceleration as the processing is done by the CPU through a generic instruction set. Higher access control (e.g., root privileges) for user-space applications to achieve software acceleration generally does not go well with isolation and has security implications in terms of privacy as multiple applications could interfere with each other Lal *et al.* (2017). Also, additional layers of software abstractions for acceleration add more latency for the overall task processing as compared to hardware acceleration.

Hardware Acceleration Methods

Overview Although software optimizations provide acceleration of NFs, software is fundamentally limited by the CPU availability (i.e., contention with other processes), load (i.e., pending tasks), and utilization (i.e., average idle time) based on the active task computing that the CPU is trying to accomplish. NFs typically require routine tasks, such as IP look-up for network layer (Layer 3) forward routing operations. For data link layer (Layer 2) operations, the MAC look-up and port forwarding that needs to be performed for every frame creates a high I/O bound workload. Similarly, the encapsulation and decapsulation of every packet needed for tunnel-based forwarding constitutes a high memory bound workload. A more CPU intensive type of task is, for instance, encryption and decryption of IP packets for security. In order to maximize the performance, the CPU has to frequently monitor the NIC and has to process the IP packets as part of an NF; both of these actions consume large numbers of CPU cycles. Therefore, hardware based acceleration is critical for NF development and deployments.

Hardware acceleration can be broadly categorized into custom acceleration and dedicated acceleration. Custom acceleration is generic and programmable to application requirements either at run-time or preloaded based on the need. Examples of custom acceleration are Graphic Processing Unit (GPU) and Field Programmable Gate Arrays (FPGA). In contrast, dedicated hardware acceleration is designed and validated in hardware for a defined function, with little or no programming flexibility to change the behavior of hardware at run-time. On the other hand, custom hardware acceleration is cost effective and easy to configure which helps in developing new protocols and behaviors that are adapted to the applications.

Benefits As compared to software acceleration, hardware acceleration provides more robust advantages in terms of saving CPU cycles that execute the NF processing tasks than implementation as a software. Overall, hardware accelerators significantly improve the system throughput and task latency as well as energy efficiency for NF implementations Benini *et al.* (2012).

Shortcomings The main shortcomings of hardware accelerations are:

i) Longer time frame for development cycle than for software acceleration development.

ii) For every hardware component there is an associated software component that needs to be developed and maintained.

iii) Introduction of new technologies, newer specifications and skills to manage the hardware.

iv) Higher cost of implementation and adoption into market.

v) Infrastructure upgrades with new hardware components are difficult

vi) Locked in vendors for hardware and maintenance support.

3.3 Enabling Technologies for Hardware-Accelerated Platforms and Infrastructures for NF Implementation

This section comprehensively surveys the enabling technologies for hardware-accelerated platforms and infrastructures for implementing NFs. This section is

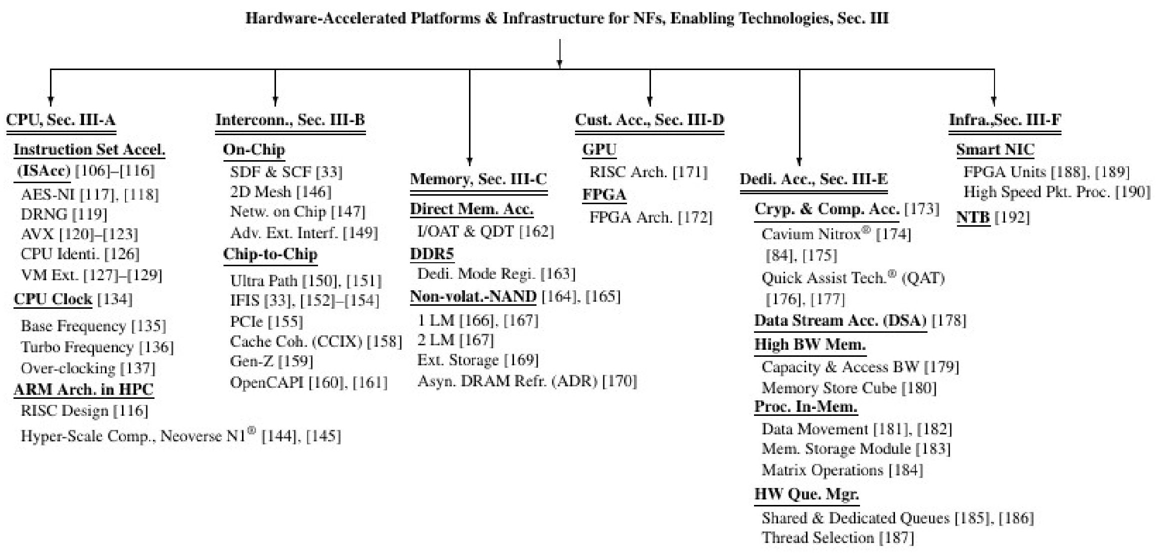


Figure 3.2: Classification Taxonomy of Enabling Technologies For Hardware-accelerated Platforms and Infrastructure for Processing Softwarized Nfs: The Main Platform Related Categories Are Hardware Accelerations for the Cpu, Interconnects, and Memory, as Well As Custom and Dedicated Hardware Accelerators That Are Embedded on The Platform; The Infrastructure Hardware Accelerations Focus on Network Interface Cards and Bridging.

structured according to the classification structure of the enabling technologies in Fig. 3.2, whereby a subsection is dedicated to each of the main categories of enabling technologies, i.e., CPU, interconnects, memory, custom accelerators, dedicated accelerators, and infrastructure.

3.3.1 Central Processing Unit (CPU)

Traditionally in the current deployments, the CPU performs nearly all the computing required by an NF. While most NF computing needs can be met by a CPU, an important question is to decide whether a CPU is the ideal resource to perform the NF tasks. For instance, a polling function only continuously monitors a hardware register or a memory location; a CPU may not be well suited for such a polling function. This section comprehensively surveys the enabling technologies for accelerating the operation of the CPU hardware for processing NFs.

Instruction Set Acceleration (ISAcc)

An *instruction* is a fundamental element that defines a CPU action. A CPU action can be a basic operation to perform an arithmetic or logic operation on two variables, to store or to retrieve data from memory, or to communicate with an external I/O device. The instruction set (IS) is a set of instructions that are pre-defined; the IS comprehensively lists all the CPU operations. In the computing literature, the IS is also commonly referred to as Instruction Set Architecture (ISA); for brevity, we use the terminology “Instruction Set (IS)” and define the acronym “ISAcc” to mean “Instruction Set Acceleration”. The properties of the IS list distinguish the type of CPU, typically as either Reduced Instruction Set Compute (RISC) or Complex Instruction Set Compute (CISC) Blake *et al.* (2009). Generally, RISC has a very basic set of limited operations, while CISC includes a comprehensive set of instructions targeted at complex operations. RISC is power and silicon-space efficient. However, the limited set of RISC operations generates large amounts of translated machine opcodes from a high-level programming language which will reduce performance for complex operations, such as encryption or compression. On the other hand, CISC can implement a complex operation in a single CPU *instruction* which can result is smaller machine opcodes, improving the performance for complex operations. However, CISC generally consumers higher power and requires more silicon-space than RISC.

Tensilica Tensilica (2020) is an example of low-power DSP processor based on the RISC architecture which is optimized for floating point operations Arnold *et al.* (2014b). Tensilica processors are typically used in the design of I/O devices (e.g., NIC)

Table 3.1: Cpu Instruction Set Acceleration (Cpu-isacc) Extensions: Aes-ni, Drng, and Avx-512. Cpu-isacc Optimizes Hardware Implementations of Software Functions, Such as Random Number Generation, Cryptographic Algorithms, and Machine Learning, in Terms Of Power and Performance.

	CPU Instruction	**Acceleration Function**
AES-NI	`AESENC`	One round AES encryp. flow
	`AESNCLAST`	Last round AES encryp. flow
	`AESDEC`	One round AES decryp. flow
	`AESDECLAST`	Last round AES decryp. flow
	`AESKEYGENASSIST`	AES round key generation
	`AESIMC`	AES Inverse Mix Columns
	`PCLMLUQDQ`	Carryless multiply
DRNG	`RDRAND`	Hardw.-gen. random value
	`RDSEED`	Hardw.-gen. random seed value
AVX-512	`VNNI`	Vector Neural Net. Instr.
	`GFNI`	Galois Field New Instr.
	`VAES`	Vector AES Instructions
	`VBMI2`	Vector Byte Manip. Instr. 2
	`BITALG`	Bit Algorithms

and hardware accelerators in the form of new IS definitions and concurrent thread execution to implement softwarized NFs. The IS extensions have been utilized to accelerate hashing NFs Arnold *et al.* (2014a); Pauls *et al.* (2019) and dynamic task scheduling Arnold *et al.* (2012). Similar IS extensions have accelerated the complex network coding function Nguyen *et al.* (2020); Wunderlich *et al.* (2019); Yang *et al.* (2019b) in a hardware design Acevedo *et al.* (2018).

ISAcc Hennessy and Patterson (2019); Yokoyama *et al.* (2019) provides an additional set of instructions for RISC and CISC architectures. These additional instructions enable a single CPU *instruction* to performs a specific part of the computation that is needed by an application in a single CPU execution cycle. The most important CPU instructions that directly benefit NF designs are:

Advanced Encryption Standard-New Instructions (AES-NI) Advanced Encryption Standard-New Instructions (AES-NI) Akdemir *et al.* (2010); Hofemeier and Chesebrough (2012) include IS extensions to compute the cryptography functions of the Advance Encryption Standard (AES); in particular, AES-NI includes the complete encryption and decryption flow for AES, such as AES-GCM (AES-GCM is a type of AES algorithm, and AES-ENC is used internally for GCM encryption). AES-NI has been widely used for securing HTTPS connections needed for end-to-end NFV instances over networks. HTTP uses the Transport Layer Security (TLS) Secure Sockets Layer (SSL) protocol (which incorporates AES) to generate and exchange keys as well as to perform encryption and decryption. SSL implementations, such as OpenSSL, provide the interface and drivers to interact with the AES-NI CPU acceleration instructions.

Digital Random Number Generator (DRNG) The Digital Random Number Generator (DRNG) Cox *et al.* (2011) with the RDRAND instruction can be used for generating public and private cryptographic keys. The RSEED instruction can be used for seeding software-based Pseudorandom Number Generators (PRNGs) used in cryptography protocols. DRNG is also extensively used in modeling, analytics for random selections, large scale system modeling to introduce randomization, natural disturbances, and noises in encryption and control loop frameworks, which are applicable to SDN controller-based NF designs.

CPU IDentification (`CPUID`) The CPU IDentification (`CPUID`) Cloutier (2019) instruction provides the details of CPU specifications, enabling software to make decisions based on the hardware capabilities. A user can write a predefined value to the EAX CPU register with the `CPUID` instruction to retrieve the processor specific information that is mapped to the value indicated by the EAX CPU register. A comprehensive list of CPU specifications can be enumerated by writing values in sequence to the EAX and reading the EAX (read back the same write register), as well as the related EBX, ECX, and EDX CPU registers. For instance, writing `0x00`h to the EAX provides the CPU vendor name, whereas writing `0x07`h gives information about the AVX–512 IS capability of the CPU. NF orchestration can use the `CPUID` instruction to identify the CPU specifications along with the ISAcc capabilities to decide whether an NF can be run on the CPU or not.

Virtual Machine Extensions (VMX) The Virtual Machine Extensions (VMX) Intel Corp. (2020c,d); Linux Assembly (2018) provide advanced CPU support for the virtualization of the CPU, i.e., the support for virtual CPUs (vCPUs) that can be assigned to VMs running on a Virtual Machine Monitor (VMM) Sugerman *et al.*

(2001); Plouffe *et al.* (2014). In the virtualization process, the VMM is the host OS which has direct controlled access to the hardware. VMX identifies an instruction as either a VMX root operation or a VMX non-root operation. Based on the instruction type provided by the VMX, the CPU executes a VMX root operation with direct hardware access, while a VMX non-root operation is executed without direct hardware access. The two most important aspects in virtualization are: *a*) *VM entries*, which correspond to VMX transitions from root to non-root operation, and *b*) *VM exits*, which correspond to VMX transitions from non-root to root operation. NFs implemented on a virtual platform should be aware of the VMX principles and whether an NF requires root operations to take the advantage of performance benefits in root-based operations.

Deep Learning (DL) Boost The Deep Learning (DL) Boost IS acceleration on Intel® CPUs Arafa *et al.* (2019) targets machine learning and neural network computations. The traditional implementation of floating point operations results in extensive Arithmetic and Logic Unit (ALU) computations along with frequent accesses to registers, caches, and memory. DL Boost transforms floating point operations to integer operations, which effectively translates the higher precision floating point multiply and addition operations to lower precision integer calculations. The downside is the loss of computation accuracy. However, for machine learning and neural network computations, a loss of accuracy is often tolerable. DL Boost can transform Floating Point 32 bit (FP32) operations to FP16, INT8, and further down to INT2. DL boost reduces the multiply-and-add operations, which increases system throughput while reducing latency and power consumption. An NF that requires low precision floating operation for prediction, estimation, and machine learning applications can benefit from DL Boot acceleration of the CPU IS.

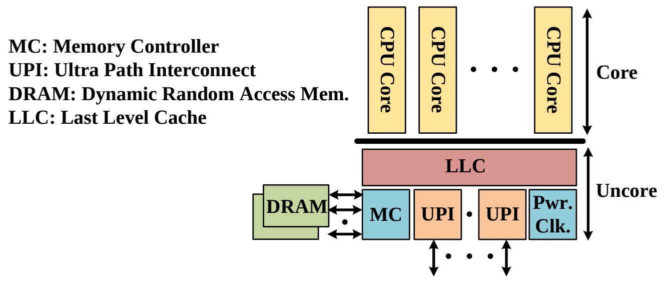


Figure 3.3: Components Inside Processor Chips Are Generally Functionally Separated into Core (I.E., Cpus) and Uncore Elements. Uncore Elements Are Non-core Components, Such as Clock, Memory Controllers, Integrated Accelerators, Interrupt Controllers, And Interconnects.

Cache Hierarchy The cache hierarchy has been commonly organized as follows:

i) The level L1 cache for code is normally closest to the CPU with the lowest latency for any memory access. A typical L1 cache for code has a size of around 64 kilobytes (KB), is shared between two cores, and has 2-way access freedom. The L1 cache for code is commonly used to store opcodes in the execution flow, whereby a block of opcodes inside a loop can greatly benefit from caching.

ii) The level L1 cache for data is a per-core cache which resides on the CPU itself. The L1 data cache typically stores the data used in the execution flow with the shortest access latency on the order of around 3–4 clock cycles.

iii) A typical level L2 cache is shared between two cores and has a size of around 1–2 MB. The access latency is typically around 21 clocks with 1 read for 4 clock cycles and 1 write for 12 clock cycles.

iv) The level L3 cache is generally referred to as shared Last Level Cache (LLC), which is shared across all cores. The L3 cache is typically outside the CPU die, but still may reside inside the processor die. A typical processor die consists of core and uncore elements Gupta *et al.* (2012) (see Fig. 3.3). Uncore elements refer to all the non-CPU components in the processor die, such as clock, Platform Controller Hub (PCH), Peripheral Component Interconnect express (PCIe) root complex, L3 cache, and accelerators.

Data-Direct IO (DDIO) The Data-Direct IO (DDIO) Intel Corp. (2012) is a cache access advancement I/O technology. The DDIO allows I/O devices, such as the PCIe based NIC, GPU, and FPGA, to directly read and write to the L3 shared LLC cache, which significantly reduces the latency to access the data received from and sent to I/O devices. Traditionally, I/O devices would write to an external memory location which would then be accessed by the CPU through a virtual to physical address translation and a page look-up process. NF applications require frequent I/O activities, especially to read and write packets between NIC and processor memory. With DDIO, when a packet arrives at the NIC, the NIC directly writes to the cache location that is indexed by the physical address of the memory location in the shared L3 cache. When the CPU requests data from the memory location (which will be a virtual address for CPU requests), the address is translated from virtual to physical, and the physical address is looked up in the cache, where the CPU finds the NIC packet data. The DIDO avoids the page walk and memory access for this packet read operation. A CPU write to NIC for a packet transmission executes the same steps in reverse. Thus, NF implementations with intense I/O can greatly benefit from the DDIO cache management.

CPU Clock

One of the critical aspects of an NF is to ensure adequate performance when running on a GPC platform. In addition to many factors, such as the transistor density, memory access speeds, and CPU processing pipeline, the CPU operational clock frequency is a major factor that governs the CPU throughput in terms of operations per second. However, in a GPC platform, the CPU clock frequency is typically dynamically scaled to manage the thermal characteristics of the CPU die Cohen *et al.* (2003). The CPU clock frequency directly impacts the total power dissipated as heat on the CPU die.

Base Frequency The base frequency Schone *et al.* (2012) is the normal CPU operational frequency suggested by the manufacturer to guarantee the CPU performance characteristics in terms of number of operations per second, memory access latency, cache and memory read and write performance, as well as I/O behaviors. The base frequency is suggested to achieve consistent performance with a nominal power dissipation to ensure sustainable and tolerable thermal features of the CPU die.

Turbo Frequency The turbo frequency technique Charles *et al.* (2009) *automatically* increases the platform and CPU operational frequency above the base frequency but below a predefined maximum turbo frequency. This frequency increase is done opportunistically when other CPUs in a multi-core system are not active or operating at lower frequencies. The turbo frequency is set according to the total number of cores running on a given CPU die, whereby the thermal characteristic of the CPU die is determined by the aggregated power dissipated across all the cores on the CPU die. If only a subset of the cores on the CPU die are active, then there is an extra thermal budget to increase the operational frequency while still meeting the maximum thermal limits. Thus, the turbo frequency technique exploits opportunities for *automatically* increasing the CPU core frequencies for achieving higher performance of applications running on turbo frequency cores.

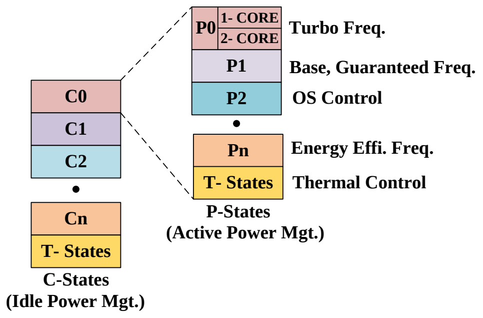


Figure 3.4: Processor States Are Broadly Classified as Cpu States (c-states) Which Indicate the Overall Cpu State; Additionally, When the Cpu Is Active (I.E., In $c0$), Then Core-specific Power States (p-states) Indicate the Operational Frequencies of The Cores That Are Actively Executing Instructions.

Over-clocking Over-clocking Jang *et al.* (2012) manually increases the CPU clock frequency above and beyond the manufacturer's suggested maximum attainable frequency, which is typically, higher than the maximum turbo frequency. Over-clocking changes the multipliers of the fundamental CPU clock frequency. A clock multiplier on the uncore part of the CPU die generally converts the lower fundamental frequency into the operating base and turbo frequencies. Over-clocking manually alters the multipliers of the clock frequency to reach the limits of thermal stability with an external cooling infrastructure. The thermal budget of the CPU die is forcefully maintained through a specialized external cooling infrastructure (e.g., circulating liquid nitrogen) that constantly cools the CPU die to prevent physical CPU damage from overheating. The highest CPU performance can be achieved through successful over-clocking procedures; however, the cost and maintenance of the cooling infrastructure limit sustained over-clocked operations. Hence only few applications can economically employ over-clocking on a consistent basis.

ARM Architectures in High Performance Computing (HPC)

RISC and CISC compute architectures with ISAcc support have recently been merging their boundaries to achieve the benefits from both architectures. The demand for low power consumption while achieving high performance has prompted RISC architectures to support High Performance Computing (HPC) capabilities. For instance, the ARMv7 RISC architecture contains the `THUMB2` extensions for 16-bit instructions similar to CISC, and the x86 ISAcc performs micro-operation translations that are similar to RISC. Yokoyama et al. Yokoyama *et al.* (2019) have surveyed the state-of-the-art RISC processor designs for HPC computing and compared the performance and power consumption characteristics of the ARMv7 based server platforms to the Intel server platforms. The results from over 400 workload executions indicate that the state-of-the-art ARMv7 platform is 2.3-fold slower than the Sandy Bridge (Intel), 3.4-fold slower than Haswell (Intel), and nearly 7% faster than Atom (Intel). However, the Sandy Bridge (Intel) platform consumes 1.2-fold more power than the ARMv7.

Figure 3.5 presents an overview of the Neoverse N1 Pellegrini *et al.* (2020) CPU architecture targeted for edge and cloud infrastructures to support hyper-scale computing. The N1 platform can scale from 8 to 16 cores per chip for low computing needs, such as networking, storage, security, and edge compute nodes, whereas, for server platforms the architecture supports more than 120 cores. For instance, a socket form factor of N1 consists of 128 cores on an 8×8 mesh fabric. The chip-to-chip connectivity (e.g., between CPU and accelerator) is enabled by the CCIX® (see Sec. 3.3.2) through a Coherent Mesh Network (CMN) interfacing with the CPU. The latency over the CMN is around 1 clock cycle per Mesh Cross Point (XP) hop. The N1 supports 8 DDR channels, up to 4 CCIX links, 128 MB of L3 cache, 1 MB of private cache along with 64 kB I-cache and 64 kB D-cache. The performance improvements of N1 as compared to the predecessor Cortex-A72 are: 2.4-folds for memory allocation, 5-folds of object/array initializations, and 20-folds for VM initiation. The Neoverse N1 has been commercially deployed on Amazon Graviton Amazon Web Services, Inc. (2020) servers, where the workload performance per-vCPU shows an improvement of 24% for HTTPS load balancing with NGNIX and 26% for X.264 video encoding as compared to the predecessor M5 server platforms of Amazon Graviton.

Summary of CPU

In summary, the CPU provides a variety of options to control and enable the features and technologies that specifically enhance the CPU performance for NF applications deployed on GPC platforms. In addition to the OS and hypervisors managing the CPU resources, the NF application designers can become aware of the CPU capabilities through the CPU instruction `CPUID` and develop strategies to run the NF application processes and threads on the CPU cores at desired frequency and power levels to achieve the performance requirements of the NF applications. In general, a platform consists of both CISC and RISC computing architectures, whereby CISC architectures (e.g., x86 and AMD) are predominantly used in hyper-scale computing operations, such as server processors, and RISC architectures are used for compute operations on I/O devices and hardware accelerators.

The CPU technologies discussed in Sec. 3.3.1 along with the general CPU technology trends in support of diverse application demands Sengupta *et al.* (2020); Datta *et al.* (2020) enable increasing numbers of cores within a given silicon area such that the linear scaling of CPU resources could—in principle—improve the overall application performance. However, the challenges of increasing the core density (number of cores per die) include core-to-core communication synchronization (buffering and routing of messages across interconnects), ensuring cache coherency across L3 caches associated with each core, thread scheduling such that the cache coherency is maximized and inter-core communication is minimized. Another side effect of the core-density increase is the higher thermal sensitivity and interference in multi-core computing, i.e., the load on a given core, can impact the performance and capacity of adjacent cores. Therefore, in a balanced platform, the compute (processes and threads) scheduling across different cores should consider several external aspects in terms of spatial scheduling for thermal balancing, cache coherency, and inter-core communication traffic.

3.3.2 Interconnects

An interconnect is a physical entity for a point-to-point (e.g., link) connection between two hardware components, or a point-to-multi-point (e.g., star, mesh, or bus) connection between three or more hardware components. Commonly, an interconnect, which can exist within a given chip (i.e., on-chip) or between multiple chips (i.e., chip-to-chip), is a physical path between two discrete entities for data exchanges. On the other hand, an interface is a logical stateful connection between two components following a common protocol, such as the Universal Serial Bus (USB) or PCIe protocol, to exchange data among each other. (Interfaces have mainly been defined for point-to-point; the PCIe has some point-to-multi-point broadcast messages, however only for control and enumeration of devices by the OS.) More specifically, an interface is the logical stateful connection, e.g., a time slot structure, that exists on a physical path (i.e., the interconnect) between two discrete physical components. For instance, there exists a USB interface on a physical USB interconnect; similarly, there exists a logical PCIe interface (e.g., slot structure) on a PCIe interconnect Gerszberg *et al.* (2000).

Physical interconnects between hardware components often limit the maximum achievable performance of the entire system due to bottlenecks, e.g., the memory transaction path limits the access of applications to shared resources. The NF design should pay close attention to interconnects and interfaces since NF application can easily saturate an interconnect or interface between hardware components, limiting the NF performance. Several interconnect and interface technologies can connect different components within a die, i.e., on-chip, and connect components die-to-die, i.e., external to the chip.

On-Chip Interconnects

On-chip interconnects, which are also referred to as on-die interconnects, connect various hardware components within a chip, such as core, accelerator, memory, and cache, that are all physically present inside the chip. On-die interconnects can be broadly categorized into core-to-core, core-to-component, and component-to-component, depending on the end-point use cases. The typical design of an on-die interconnect involves a mesh topology switching fabric built into the silicon die, which allows multiple components to simultaneously communicate with each other. The mesh topology switching fabric achieves high overall throughout and very low latency.

Scalable Data Fabric (SDF) & Scalable Control Fabric (SCF) The Infinity Scalable Data Fabric (SDF) and Scalable Control Fabric (SCF) Lepak *et al.* (2017) (see Fig. 3.6) are the AMD® proposed switching fabrics for on-die component communications. SDF and SCF are responsible for the exchange of data and control messages between any endpoint on the chip. The separation of data and control paths allows the fabric to prioritize the control communications. The SCF functions include thermal and power management on-die, built-in self-tests, security, and interconnecting external hardware components (whereby a hardware component is also sometimes referred to as a hardware Intellectual Property (IP) in this field). SDF and SCF are considered as a scalable technology supporting large numbers of components to be interconnected on-die. Similarly, Infinity Fabric On-Package (IFOP) provides die-to-die communication within a CPU socket i.e., on the same package.

2D Mesh The Intel® 2D mesh Park *et al.* (2012) (see Fig. 3.7) interconnects multiple core components within a socket. A core component along with a Cache Homing Agent, Last Level Cache (LLC), and Snooping Filter corresponds to a "Tile" in the CPU design. A tile is represented as a rectangular block that includes a core, CHA, and SF as illustrated in the Xeon® CPU overview in Fig. 3.7. The 2D mesh technology implements a mesh based interconnect to connect all the cores on a given die, i.e., single CPU socket.

In previous Intel® core architecture generations, the Home Agent (HA) was responsible for the cache management. In the current generation, each mesh stop connects to a tile, enumerated as logical number, i.e., as tile0/CHA0, tile1/CHA1, and so on; thereby effectively moving from a centralized HA to distributed CHA agents. When a memory address is accessed by the CPU, the address is hashed and sent for processing by the LLC/CHA/SF residing at the active mesh stop that is directly connected to the tile that makes the memory request. The CHA agent then checks the address hash for data presence in an LLC cache line, and the Snoop Filter (SF) checks the address hash to see if the address is cached at other LLC locations. In addition to cache line and SF checks, the CHA makes further memory read/write requests to the main memory and resolves address conflicts due to hashing.

In summary, the Infinity Fabric SDF and SCF (Fig. 3.6), and the 2D mesh (Fig. 3.7) are part of core-to-core and core-to-component designs which directly interact with the CPU on-die. On the other hand, most accelerator hardware components are external to CPUs and come as discrete components that can be (*i*) embedded on the CPU die (on-chip), but are (*ii*) externally connected to the CPU through I/O interfaces, such as PCIe.

Network on Chip (NoC) A Network on Chip (NoC) Kumar *et al.* (2002) (see Fig. 3.8) implements an on-die communication path similar to the network switching infrastructure in traditional communication networks. On-die communications over a switching fabric uses a custom protocol to package and transport data between endpoints; whereas, the NoC uses a common protocol for the transport and physical communication layer transactions. The data is commonly packetized, thus supporting variably bit-widths through serialization. An NoC provides a scalable and layered architecture for flexible communication among nodes with a high density on a given die area. An NoC has three layers: *i*) transaction, which provides load and store functions; *ii*) transport, which provides packet forwarding, and *iii*) physical, which constitutes wires and clocks. A pitfall to avoid is excessive overhead due to high densities of communication nodes on the NoC which can impact the overall throughput performance due to overhead. Additionally, an NoC can pose a difficult challenge to debug in case of a transaction error.

Advanced eXtensible Interface (AXI) The Advanced eXtensible Interface (AXI) as defined in the ARM® Advanced Micro-controller Bus Architecture (AMBA) AXI and AXI-Coherency Extension (ACE) specification ARM Holdings (2019) provides a generic interface for on-chip communication that flexibly connects various on-die components (see Fig. 3.9). The AXI interconnect provides master and slave based end-to-end connections; operations are initiated by the master, and the slaves respond to the requested operation. As opposed to operations, transfers on AXI can be mutually initiated. Dedicated channels are introduced for multiple communication formats, i.e., address and data. Each channel is essentially a bus that is dedicated to send the message of similar type: *i*) Address Write (AW), *ii*) Address Read (AR), *iii*) Write Data (W), *iv*) Read Data (R), and *v*) Write Response (R). These dedicated channels provide an asynchronous data transfer framework that allows concurrency in read and write requests simultaneously between master and slave. If there are multiple components with caches associated with each IP, ACE provides an extension to AXI that provides cache coherency between multiple IPs (i.e., components on-die) by maintaining coherence across multiple caches. Cache coherency is only applied to components that act as the master in the AXI transactions.

Chip-to-Chip

While on-chip interconnects provide connectivity between hardware components inside a chip or a die, chip-to-chip interconnects extend physical interconnects outside the chip for extending communication with an external IP component, i.e., hardware block present on another chip.

Ultra Path Interconnect (UPI) The Intel® Ultra Path Interconnect (UPI) Meng *et al.* (2018); Tam *et al.* (2018) implements a socket-to-socket interconnect that improves upon its predecessor, the Quick Path Interconnect (QPI). The UPI allows multiple processors to access shared addresses with coordination and synchronization, which overcomes the QPI scalability limitations as the number of cores increases. In coordination with the UPI, a Caching and Home Agent (CHA) maintains the coherency across the cores of multiple sockets, including the management of snoop requests from cores with remote cache agents Thus, the UPI provides a scalable approach to support high socket densities on a platform while supporting cache coherency across all the cores. The UPI supports 10.4 Giga Transfers per second (GT/s), which is effectively 20.8 GB/s. The UPI can interconnect processor cores over multiple sockets in the form of 2-way, 4-way, and 8-way Symmetric Multiprocessing (SMP), with 2 or 3 UPI interconnects on each socket, as illustrated in Fig. 3.10 for Intel® Skylake processors.

Infinity Fabric InterSocket (IFIS) The Infinity Fabric InterSocket (IFIS) Beck *et al.* (2018); Lepak *et al.* (2017); AMDl (2020); Teich (2017) of AMD® implements package-to-package (i.e., socket-to-socket) communication to enable two-way multi-core processing. A typical IFIS interconnect has 16 transmit-receive differential data lanes, thereby providing bidirectional connectivity with data rates up to 37.93 GBs. IFIS is implemented with a Serializer-Deserializer (SerDes) for inter-socket physical layer transport whereby data from a parallel bus of the on-chip fabric is serialized to be transported over IFIS interconnect; the deserializer then parallelizes the data for the on-chip fabric. One key IFIS property is to multiplex data from other protocols, such as PCIe and Serial AT Attachment (SATA), which can offer transparent transport of PCIe and SATA packets over multiple sockets.

Due to their high physical complexity and cost, UPI and IFIS are only employed for inter-socket communication between CPU sockets. However, the vast majority of the compute pipeline hardware components, such as memory and I/O devices, could lie outside of the CPU socket chip, depending on the compute package design of the GPC platform. Therefore, it is critical for NF performance to consider general chip-to-chip interconnects beyond CPU sockets. The dominant general state-of-the-art hardware chip-to-chip interconnects are the Peripheral Component Interconnect express (PCIe) and Compute eXpress Link (CXL) which are summarized below.

Peripheral Component Interconnect express (PCIe) The Peripheral Component Interconnect express (PCIe) McGinnis (2017) (see Fig. 3.12) is a chip-to-chip interconnect and interface protocol that enables an external system-on-chip component,

Table 3.2: Summary of Pcie Lane Rates Compared Across Technology Generations from Gen 1.1 Through Gen 5: The Raw Bitrate Is In Giga Transfers per Second, and the Total Bandwidth in Giga Byte Per Second Is given for 16 Parallel Lanes in Both Directions For Application Payload (Without the Pcie Transaction, Link Layer, And Physical Layer Overheads).

PCIe Gen.	**Raw Bitrate (GT/s)**	**BW per lane per direc. (App.) (GB/s)**	**Total BW for 16-lane Link (App.) (GB/s)**
1.1	2.5	0.25	8
2.0	5	0.50	16
3.0	8	1	32
4.0	16	2	64
5.0	32	4	128

e.g., the PCIe enables a non-CPU chip (such as NIC or disk) to connect to a main CPU socket. The PCIe can connect almost any I/O device, including FPGA, GPU, custom accelerator, dedicated accelerator (such as ASIC), storage device, and networking device (including NIC). The current PCIe specification generation is 5.0 which offers a 4 GB/s speed for each directional lane, and an aggregated total throughput over 16 lanes of 128 GB/s, as shown in Table 3.2.

The PCIe follows a transactional protocol with a top-down tree hierarchy that supports serial transmissions and unidirectional links running in either direction of the PCIe link. The PCIe involves three main types of devices: Root Complex (RC): A RC is a controller that is responsible for direct memory access (DMA), address look-up, and error management; End Point (EP): An endpoint is a device that connects to the PCIe link; and Switch: A switch is an extension to the bus to which an endpoint (i.e., device) can be connected. The system BIOS enumerates the PCIe devices, starting from the RC, and assigning identifiers referred to as "Bus:Device:Function" or "BDF" for short, a 3 tuple to locate the device placement in the PCIe hierarchy. For instance, a system with a single root complex could have the identifier of `00:00:1`, with bus ID `00`, device ID `00`, and function `1`.

The PCIe does not support sideband signaling; hence, all the communication has to be conducted in a point-to-point fashion. The predecessor of the PCIe was the PCI, which had lower throughput due to skew across the parallel bus width; however, to maintain backward compatibility, the PCIe allows PCI devices to be connected via a PCIe-to-PCI bridge. There are almost no PCI devices in the recent platforms, as the PCIe provides both cost efficiency and performance benefits. However, the OS recognizes PCIe switches as bridges to keep backward compatibility with the software drivers and hence can be seen in the enumeration process of the PCIe. Essentially, every switch port is a bridge, and hence appears so in the OS listing of all PCIe devices.

CCIX®: Cache Coherent Interconnect for Accelerators One factor that limits the hardware accelerator performance in accelerating softwarized NFs is the memory transaction bottleneck between system memory and I/O device. Data transfer techniques between system memory and I/O device, such as DDIO (see Sec. **??**), utilize a system cache to optimize the data transactions between the system memory and I/O device. For I/O transactions, a cache reduces the latencies of memory read and write transactions between the CPU and system memory; however, there is still a cost associated with the data transactions between the I/O device and system memory. This cost can be reduced through a local device-cache on the I/O device, and by enabling cache coherency to synchronize between the CPU-cache and the device-cache.

While the CXL/PCIe based protocols define the operations supporting cache coherency between the CPU and I/O devices, the CXL/PCIe protocols define strict rules for CPU/core and I/O device endpoint specific operations. The Cache Coherent Interconnect for Accelerators (CCIX®) CCIX® Consortium Incorp. (2020) (pronounced "See 6") is a new interconnect design and protocol definition to seamlessly connect computing nodes supporting cache coherency (see Fig. 3.14(a)).

Another distinguishing CCIS feature (with respect to CXL/PCIe) is that the CCIX defines a non-proprietary protocol and interconnect design that can be readily adopted by processors and accelerator manufacturers. The CCIX protocol layer is similar to the CXL in terms of the physical and data link layers which are enabled by the PCIe specification; whereas, the transactions layer distinguishes between CCIX and PCIe transactions. While the cache coherency of the CXL protocol is managed by invoking `CXL.cache` instructions, the CCIX protocol *automatically* synchronizes the caches such that the operations are driver-less (no software intervention) and interrupt-less (i.e., no CPU attention required). The automatic synchronization reduces latencies and improves the overall application performance. The CCIX version 1.1 supports the maximum bandwidth of the PCIe 5.0 physical layer specification of up to 32 Giga Transactions per second (GT/s). Figure 3.14(a) illustrates the protocol layer operations in coexistence with the PCIe, and shows the different possible CCIX system topologies to flexibly interconnect processors and accelerators.

Generation-Z (Gen-Z) The Gen-Z Consortium **?** (see Fig 3.14(b)) has proposed an extensible interconnect that supports on-chip, chip-to-chip, and platform-to-platform communication. As opposed to the CXL and CCIX, Gen-Z has defined: *i*) direct connect, *ii*) switched, and *iii*) fabric technologies for homogeneously connecting compute, memory, and I/O devices. For cross-platform connections, Gen-Z utilizes networking protocols, such as InfiniBand, to enable connections via traditional optical

Ethernet links. More specifically, Gen-Z supports DRAM memory extensions through persistent memory modules with data access in the form of byte addressable load/store, messaging (put/get), and I/O block memory. Gen-Z provides management services for memory disaggregation and pooling of shared memory, allowing flexible resource slicing and allocations to the OS and applications. In contrast to other interconnects, Gen-Z inherently supports data encryption as well as authentication for access control methods to facilitate the long-haul of data between platforms. Gen-Z preserves security and privacy through Authenticated Encryption with Associated Data (AEAD), whereby AEAD encryption is supported by the AES-GCM-256 algorithm. To support a wide range of connections, the Gen-Z interconnect supports variable speeds ranging from 32 GB/s to more than 400 GB/s.

Open Coherent Accelerator Processor Interface (OpenCAPI) The Open Coherent Accelerator Processor Interface (OpenCAPI) Stuecheli *et al.* (2018) (see Fig. 3.14(c) and (d)) is a host-agnostic standard that defines procedures to coherently connect devices (e.g., hardware accelerator, network controller, memory module, storage controller) with the host platform. A common protocol is applied across all the coherently connected device memories to synchronize with the system memory to facilitate accelerator functions with reduced latency. In addition to cache coherency, OpenCAPI supports direct memory access, atomic operations to host memory, messages across devices, and interrupts to the host platform. High frequency differential signaling technology Zhang *et al.* (2009) is employed to achieve high bandwidth and low latency connections between hardware accelerators and CPU. The address translation and coherency cache access constructs are encapsulated by OpenCAPI through serialization which is implemented on the platform hardware (e.g., CPU socket) to minimize the latency and computation overhead on the accelerator device. As compared to the

CXL, CCIX, and Gen-Z, the transaction as well as link and physical layer attributes in OpenCAPI are aligned with high-speed Serializer/Deserializer (SerDes) concept to exploit parallel communication paths on the silicon. Another aspect of OpenCAPI is the support for virtual addressing, whereby the translations between virtual to physical addresses occur on the host CPU. OpenCAPI supports speeds up to 25 Gbps per lane, with extensions up to 32 lanes on a single interface. The CXL, CCIX®, and OpenCAPI interconnects are compared in Table **??**.

Summary of Interconnects and Interfaces

Interconnects provide a physical path for communication between multiple hardware components. The characteristics of on-chip interconnects are very different from chip-to-chip interconnects. NF designers should consider the aspects of function placement, either on the CPU die or on an external chip. For instance, an NoC provides a scalable on-chip fabric to connect the CPU with accelerator components, and also to run a custom protocol for device-to-device or device-to-CPU communication on top of the NoC transport and physical communication layers. The PCIe provides a universal physical interconnection system that is widely supported and accepted; whereas, the CXL provides cache coherency functionalities if needed at the device (i.e., accelerator component).

One of the key shortcomings of existing interconnects and interfaces is the resource reservation and run-time reconfiguration. As the density of platform hardware components, such as cores, memory modules (i.e., DRAM), and I/O devices, increases, the interconnects and interfaces that enable physical connections are multiplexed and shared to increase the overall link utilization. However, shared links can cause performance variations at run-time, and can result in interconnect and interface resource saturation during high workloads. Current enabling technologies do not provide a mechanism to enforce Quality-of-Service (QoS) for the shared interconnect and interface resources. Resource reservation strategies based on workload (i.e., application) requirements and link availability should be developed in future work to provide guaranteed interconnect and interface services to workloads.

3.3.3 Memory

Although the expectation with high-speed NICs, large CPU compute power, as well as large and fast memory is to achieve improved network performance, in reality the network performance does not scale linearly on GPC platforms. The white paper Intel Corp. (2019e) has presented a performance bottleneck analysis of high-speed NFs running on a server CPU. The analysis has identified the following primary reasons for performance saturation: *i*) interrupt handling, buffer management, and OS transitions between kernel and user applications, *ii*) TCP stack code processing, and *iii*) packet data moves between memory regions and related CPU stalls. Towards addressing these bottlenecks, factors that should be considered in conjunction with memory optimizations that relate to data transfers between I/O devices and system memory are: *a*) interrupt moderation, *b*) TCP checksum offloading and TCP Offload Engine (TOE), and *c*) large packet transfer offloading. We proceed to survey efficient strategies for memory access (i.e., read and write) which can mitigate the performance degradations caused by packet data moves.

Direct Memory Access (DMA)

Memory transactions often take many CPU cycles for routine read and write operations from or to main memory. The Direct Memory Access (DMA) alleviates the problem of CPU overhead for moving data between memory regions, i.e., within a RAM, or between RAM and an I/O device, such as a disk or a PCIe device (e.g., an accelerator).

The DMA offloads the job of moving data between memory regions to a dedicated memory controller and engine. The DMA supports the data movement from the main system memory to I/O devices, such as PCIe endpoints as follows. The system configures a region of the memory address space as Memory Mapped I/O (MMIO) region. A read or write request to the MMIO region results in an I/O read and write action; thereby supporting the I/O operations of write and read to and from external devices.

I/O Acceleration Technology (I/OAT) The Intel® I/O Acceleration Technology (I/OAT), as part of the Intel® QuickData Technology (QDT) Nagaraj and Gianos (2015), advances the memory read and write operations over I/O, specifically targeted for NIC data transfers. I/OAT provides the NIC direct access to the system DMA for read write access in the main memory region. When a packet arrives to the NIC, traditionally, the packet is copied by the NIC DMA to the system memory (typically at the kernel space). Note that this DMA is present on the I/O device/endpoint (an external entity) and then an interrupt is sent to the CPU. The CPU then copies the packet into application memory, which could be achieved by initiating a second DMA request, this time on the system DMA, for which the packet is intended. With the proposed QDT, the NIC can request that the system DMA further copies the data onto the application memory without CPU intervention, thus reducing a critical bottleneck in the packet processing pipeline. DMA optimizations have also been presented as part of the Intel® QuickData Technology (QDT) Nagaraj and Gianos (2015).

Dual Data Rate 5 (DDR5)

As technologies that enable NFs, such as NICs, increase their network connectivity data speeds to as high as 100–400 Gbps, data processing by multiple CPUs requires very

Table 3.3: Summary of Double Data Rates (Ddr) Synchronous Data Random Access Memory (Sdram) Rates. The Buffer Size Indicates The Multiplying Factor to the Single Data Rate Sdram Prefetch Buffer Size. The Chip Density Corresponds to the Total Number Of Memory-cells per Unit Chip Area, Whereby Each Memory Cell Can Hold A Bit. The Ddr Rates Are in Mega Transfers per Second (Mt/S). For Ddr4 and Ddr5, the Access to Dram Can Be Performed in the Group Of Memory Cells Which Are Logically Referred to as Memory Banks. That Is, a Single Read/Write Transaction to Dram Can Access the Entire Data Present in a Memory Bank.

DDR Ver.	**DDR**1	**DDR**2	**DDR**3	**DDR**4	**DDR**5
Release Date	2000	2003	2007	2012	2019
Vol. (V)	2.5	1.8	1.5	1.2	1.1
Buffer Size	2	4	8	8	16
Chip Den. (Gb)	0.128–1	0.128–4	0.512–8	2–16	8–64
Data Rate (MT/s)	200–400	400–800	800–2133	1600–3200	3200–6400
Bank Groups	0	0	0	4	8

fast main memory access. Synchronous Dynamic Random Access Memory (SDRAM) enables a main system memory that offers high-speed data access as compared to storage I/O devices. SDRAM is a volatile memory which requires a clock refresh to keep the stored data persistently in the memory. The Dual Data Rate (DDR) improves the SDRAM by allowing memory access on both the rise and fall edges of the clock, thus doubling the data rate compared to the baseline SDRAM. The DDR 5th Generation is the current technology of DDR-SDRAM that is optimized for low latency and high bandwidth, see Table 3.3. The DDR5 addresses the limitations of the DDR4 mainly on the bandwidth per core, as multiple cores share the bandwidth to the DDR.

The higher DDR5 data rate is achieved through several improvements, including improvements of the Duty Cycle Adjuster (DCA) circuit, oscillator circuit, internal reference voltages, and read training patterns with dedicated mode registers Rooney and Koyle (2019). The DDR5 also increases the total number of memory bank groups to twice of the DDR4, see Table 3.3. Overall, the DDR5 maximum data rate is twice the DDR4 maximum data rate, see Table 3.3. The DDRs are connected to a platform in the form of Dual In-line Memory Module (DIMM) cards with 168-pins to 288-pins. In addition to memory modules, DIMMs are a common form of connectors for high speed storage modules to CPU cores.

Non-Volatile NAND (NV-NAND)

In general, memory (i.e., DRAM) is expensive, provides fast read/write access by the CPU, and offers only small capacities; whereas, storage (i.e., disk) is relatively cheap, offers large capacities, but only slow read/write access by the CPU. Read/write access by the CPU to DRAM is referred to as memory access; while disk read/write access follows the procedures of I/O mechanisms requiring more CPU cycles. The slow disk read/write access introduces an I/O bottleneck in the overall NF processing pipeline, if the NF is storage and memory intensive. Some NF examples that require intensive memory and storage access are Content Distribution Networks (CDN) and MEC applications, such as Video-on-Demand and Edge-Live media content delivery.

The Non-Volatile NAND (NV-NAND) technology Weiland *et al.* (2018) strives to address this bottleneck through so-called Persistent Memory (PM), whereas NV-RAM is a type of Random Access Memory (RAM) that uses NV-NAND to provide data-persistence. In contrast to DRAM, which requires a synchronous refresh to keep the memory active (persistent) on the memory cells, NV-NAND technology retains the data in the memory cells in the absence of a clock refresh. Therefore, NV-NAND technology has been seen as solution to growing demand for larger DRAM and faster access to disk storage. Non-Volatile DIMMs (NVDIMMs) in conjunction with the 3D crosspoint technology can create NAND cells with high memory cell density in a given package Burr *et al.* (2014), achieving memory cell densities that are many folds higher as compared to the baseline 2D NAND layout design. PM can be broadly categorized into: *i*) Storage Class Memory (SCM) 1 Level Memory (1LM), i.e., PM as a linear extension of DRAM, *ii*) Storage Class Memory (SCM) 2 Level Memory (2LM), i.e., PM as main memory and DRAM as cache, *iii*) Application-Direct mode (DAX), i.e., PM as storage in NVDIMM form, and *iv*) PM as external storage, i.e., disk.

NVDIMMs can operate as both modes of memory, i.e., DRAM and storage, based on the application use. As opposed to actual storage, the Storage Class Memory (SCM) is a memory featured in NVDIMMs that provides the DRAM class operational speeds at storage size. SCM targets memory-intensive applications, such as Artificial Intelligence (AI) training and media content caching. The memory needs could further differ in terms of use, for instance, AI applications are transactions-driven due to CPU computations, while media content caching is storage driven. Therefore, SCM is further categorized into 1LM and 2LM.

1 **Level Memory (1LM)** In the 1LM memory Ray *et al.* (2017); Intel Corp. (2019b) operational mode, the OS sees NVDIMM PM memory as an available range of memory space for reads and writes. The CPU uses normal load and store instructions that are used for DRAM-access to access the PM NVDIMM memory. However, the data reads and writes over the PM are significantly slower compared to the DDR DRAM access.

2 **Level Memory (2LM)** In the 2LM Intel Corp. (2019b) mode (see Fig. 3.15), the DRAM is used as cache which only stores the most frequently accessed data, while

the NVDIMM memory is seen as larger capacity alternative to the DRAM with the Byte-Addressable Persistent Memory (B-APM) technique. The caching operation and management are provided by the memory controller of the CPU unit. Although data stored in NVDIMM is persistent, the memory controller invalidates the memory upon power loss or at an OS restart while operating in memory mode. 2LM technologies are also the type of Storage Class Memory (SCM) that is used for data-persistent storage usage of memory, as they provide the large capacity of disks while operating at close to memory speeds.

External Storage In contrast to PM, NVM express (NVMe) is also NAND based storage which exists in a PCIe form factor and has an on-device memory controller along with I/O DMA. Since NVMe operates as an external device to the CPU, the OS has to follow the normal process of calling kernel procedures to read the external device data Xu *et al.* (2015). Therefore, storage devices in the NVDIMM form factors outperform NAND based Solid State Disks (SSDs) because of utilizing the DDR link instead of the standard PCIe based I/O interface, as well as the proximity of the DIMMs to the CPU cores.

Asynchronous DRAM Refresh (ADR) Asynchronous DRAM Refresh (ADR) Han *et al.* (2018) is a platform feature in which the DRAM content can be backed up within a momentary time duration powered through super capacitors and batteries just before and after the power state is down on the system platform. The ADR feature targets DDR-SDRAM DIMMs to save the last-instant data by flushing the data present in buffers and cache onto SDRAM and putting the SDRAM on self-refresh through power from batteries or super capacitors. The ADR is an OS-aware feature, where the data is recovered for the analysis of a catastrophic error which brought down the system, or to update the data back to the main memory when the power is restored by the OS. There types of data need to be saved in case of a catastrophic error or power outage are: *i*) CPU cache *ii*) data in the memory controller, and *iii*) I/O device cache, which will be saved to the DRAM during the ADR process. In case of NVDIMMs, the DRAM contents can be flushed to PM storage such that the data can be restored even after an extended power-down state.

Summary of Memory

The networking workloads that run on GPC platforms depend on memory for both compute and storage actions. The overall NF performance can be compromised due to saturation on the memory I/O bus and high read/write latencies. Therefore, in this section we have surveyed state-of-art strategies that directly improve the NF performance that directly improve the memory performance so as to aid NFs. DMA strategies help haul packets that arrive at the NIC (an external component) to memory, and DDR memory offers DIMMs based high-speed low-latency access to the CPU for compute actions on the packet data. For storage and caching based network applications, the PM based NVDIMM can offer very large memory for storage at close to DRAM speeds.

The pitfalls that should be considered in the NF design are the asymmetric memory latency speeds between DRAM and NVDIMM PM. Also, the 2LM memory mode of operations needs to be carefully considered, when there is no requirement for caching, but a need for very low latency transactions.

The shortcomings of memory enabling technologies include asymmetric address translation and memory read latencies arising from the non-linear characteristics of address caching (Translation Lookahead Buffers [TLB]) and data caching (e.g., L3). The asymmetric read and write latencies cause over-provisioning of DRAM and cache resources (for VM deployments) to ensure a minimum performance guarantee. In addition, the memory controller is commonly shared among all the cores on a die, whereby the read/write requests are buffered to operate and serve the requester (CPU or I/O devices) at the DDR rates. Hence, as an enhancement to current enabling technologies, there is a need for memory controller based resource reservation and prioritization according to the workload (application) requirements.

3.3.4 Custom Accelerators

This section surveys hardware accelerator devices that are embedded on the platforms or infrastructures to speed up NF processing; typically, these hardware accelerators relieve the CPU of some of the NF related processing tasks. The major part of the NF software still runs on the CPU, however, a characteristic, i.e., a small part of the NF (e.g., compression or cryptography) is offloaded to the hardware accelerator, i.e., the hardware accelerator implements a small part of the NF as a characteristic. In a custom accelerator, a software program is typically loaded on a GPU or FPGA to perform a specific acceleration function (e.g., a cryptography algorithm), which is a small part of the overall NF software.

Accelerator Placement

Hardware accelerator devices (including GPU and FPGA) can be embedded on the platforms and infrastructures with various placements based on the design requirements. The hardware design of an acceleration device includes an Intellectual Property of the Register Transistor Logic (RTL) logic circuit, processors (e.g., RISC) for general purpose computing, along with firmware and microcodes to control and configure the acceleration device, as well as internal memory and cache components. In general, all the components that realize an acceleration function in a hardware acceleration device are commonly referred to as “acceleration IP”.

The acceleration IP (a blue print of the hardware accelerator device) can be embedded on a silicon chip with different placements: *i*) on-core, *ii*) on-CPU-die, *iii*) on-package (socket chip), *iv*) on-memory, or *v*) on-I/O device (e.g., PCIe or USB), as illustrated in Figure 3.16. The on-core, on-CPU-die, and on-package accelerator placements are referred to as an “integrated I/O device”. Regardless of the accelerator device placement, the CPU views the hardware accelerator as an I/O device (during OS enumeration of the accelerator function) to maintain the application and software flexibility.

The placement of a hardware accelerator is governed by *i*) the original ownership of the acceleration IP, and *ii*) the IP availability and technical merit to the CPU and memory manufacturers to have an integrated device embedded with the CPU or memory module. The placement of an accelerator I/O device as an external component to the CPU has the disadvantages of longer latencies and lower bandwidths as compared to the on-core, on-die, on-package, or on-memory placement of a hardware acceleration device as an integrated I/O device. On the other hand, the integrated I/O device requires area and power on the core, die, or package.

Graphic Processing Unit (GPU)

CPUs have traditionally been designed to work on a serial set of instructions on data to accomplish a task. Although the computing requirements of most applications fit the computation method of CPUs. i.e., the serial execution of instructions, some applications require a high degree of parallel executions. For instance, in graphic processing, the display rendering across the time and spatial dimensions are independent for the display data for each pixel. Serialized execution of instructions to perform computations on each independent pixel would be inefficient, especially in the time dimension.

Therefore, a new type of processing unit, namely, the General-Purpose Graphic Processing Unit (GP-GPU) was introduced to perform a large number of independent tasks in parallel, for brevity, we refer to a GP-GPU as a "GPU". A GPU has a large a number of cores, supported by dedicated cache and memory for a set of cores; moreover, a global memory provides shared data access, see Fig. 3.17. Each GPU core is equipped with integer and floating point operational blocks, which are efficient for arithmetic and logic computations on vectored data. CPUs are generally classified into RISC and CISC in terms of their IS features. In contrast, GPUs have a finite set of arithmetic and logic functions that are abstracted into functions and are not classified in terms of RISC or CISC. A GPU is generally considered as an independent type of computing device.

To get a general idea of GPU computing, we present an overview of the GPU architecture from Nvidia NVidia Fermi (2009) (see Fig. 3.17) which consists of Streaming Multiprocessors (SMs), Compute Unified Device Architecture (CUDA) Core, Load/Store (LD/ST) units, and Special Function Units (SFUs). A GPU is essentially a set of SMs that are configured to execute independent tasks, and there exist several SMs (e.g., 16 SMs) in a single GPU. An SM is an individual block of the execution entity consisting of a group of cores (e.g., 32 cores) with a common register space (e.g., 1024 registers), and shared memory (e.g., 64KB) and L1 cache. A core within an SM can execute multiple threads (e.g., 48 threads). Each SM has multiple (e.g., 16) Load/Store (LD/ST) units which allow multiple threads to perform LD/ST memory actions per clock cycle. A GPU thread is an independent execution sequence on data. A group of threads is typically executed in a thread block, whereby the individual threads within the group can be synchronized and can cooperate among themselves and with a common register space and memory.

For GPU programming, the CPU builds a functional unit called “kernel” which is then sent to the GPU for instantiation on compute blocks. A kernel is a group of threads working together to implement a function, and these kernels are mapped to thread blocks. Threads within a block are grouped (e.g., 32 threads) into warps and an SM schedules these warps on cores. The results are written to a global memory (e.g., 16 GB per GPU) which can be then copied back to the system memory.

Special Function Units (SFUs) execute structured arithmetic or mathematical functions, such as sine, cosine, reciprocal, and square root, on vectored data with high efficiency. An SFU can execute only one function per clock cycle, per thread, and hence should be shared among multiple threads. In addition to SFUs, a Texture Mapping Unit (TMU) performs application specific functions, such as image rotate, resize, add distortion and noise, and performs 3D plane object movements.

Packet processing is generally a serialized execution process because of the temporally ordered processing of packets. However, with several ongoing flows whereby each flow is an independent packet sequence, GPUs can be used for parallelized execution of multiple flows. Therefore, NF applications which operate on large numbers of packet flows that require data intensive arithmetic and logic operations can benefit from GPU acceleration.

Traditionally, GPUs have been connected through a PCIe interface, which can be a bottleneck in the overall system utilization of the GPU for parallel task computing Li *et al.* (2019a). Therefore, Nvidia has proposed a new NVlink interconnect to connect multiple GPUs to a CPU. Additionally, the NVSwitch is a fabric of interconnects that can connect large numbers of GPUs for GPU-to-GPU and GPU-to-CPU communication.

Field Programmable Gate Arrays (FPGA)

CPUs and GPUs provide a high degree of flexibility through programming frameworks and through executing compiled executable code at run-time. To support such programming frameworks, CPUs and GPUs are built to perform general-purpose computing. However, in certain applications, in addition to programming flexibility there is a greater requirement for performance which is typically achieved by dedicated hardware. Field Programmable Gate Array (FPGA) architectures attempt to address both requirements of programmability and performance Farooq *et al.* (2012). As illustrated in Fig. 3.18, the main architectural FPGA blocks are: *i*) logic blocks, *ii*) routing units, and *iii*) I/O blocks. Logic blocks are implemented as Compute Logic Blocks (CLBs) which consist of Look-up Tables (LUTs) and flip-flops. These CLBs are internally connected to form a matrix of compute units with a programmable switching and routing network which eventually terminates at the I/O blocks. The I/O blocks, in turn, connect to external system interconnects, such as the PCIe, to communicate with the CPU and other system components.

The FPGA programming technology determines the type of device and the relative benefits and disadvantages. The standard programming technologies are: *i*) Static RAM, *ii*) flash, and *iii*) anti-fuse. Static-RAM (SRAM) is the most commonly implemented and preferred programming technology because of its programming flexibility and CMOS silicon design process for the FPGA hardware. In SRAM based FPGA, static memory cells are arranged as an array of latches which should be programmed on power up. The SRAM FPGAs are volatile and hence the main system must load a program and configure the FPGA computing block to start the task execution.

The flash technique employs non-volatile memory cells, which do not require the main system to load the configuration after a power reset. Compared to SRAM FPGAs, flash-based FPGAs are more power efficient and radiation tolerant. However, flash FPGAs are cost ineffective since flash does not use standard CMOS silicon design technology.

In contrast to the SRAM and flash techniques, the anti-fuse FPGA can be programmed only once, and offers lower size and power efficiency. Anti-fuse refers to the programming method, where the logic gates have to be burned to conduct electricity; while "fuse" indicates conduction, anti-fuse indicates the initial FPGA state in which logic units do not exhibit conduction.

The programmable switching and routing network inside an FPGA realizes connectivity among all the involved CLBs to complete a desired task through a complex logic operation. As illustrated in Fig. 3.18, the FPGA switching network can be categorized into two basic forms: *i*) island-style routing (Fig. 3.18(b)), and *ii*) hierarchical routing (Fig. 3.18(c)). In island-style routing, Switch Boxes (SBs) configure the interconnecting wires, and connect to a Connection Box (CB). CBs connect CLBs, whereas SBs connect CBs. In a hierarchical network, multiple levels of CLBs connect to a first level of SBs, and then to second level in a hierarchical manner. For better performance and throughput, the island-style is commonly used. State-of-the-art FPGA designs have transceiver I/O speeds above 28 Gbps, RAM blocks, and Digital Signal Processing (DSP) engines to implement signal processing routines for packet processing.

NFs can significantly benefit from FPGAs due to their high degree of flexibility. An FPGA can be programmed to accelerate multiple protocols or part of a protocol in hardware, thereby reducing the overall CPU load. However, the data transactions between the FPGA, NIC, and CPU need to be carefully coordinated. Importantly, the performance gain from FPGA acceleration should exceed the overhead of packet movement through the multiple hardware components.

Summary of Custom Accelerators

Custom accelerators provide the flexibility of programmability while striving to achieve the hardware performance. Though there is gap in the degree of flexibility and performance, technological progress has produced hybrid solutions that approach the best of both worlds.

The GPU implementation Lee *et al.* (2010a) of NF applications is prudent when there are numerous independent concurrent threads working on independent data. It is important to keep in mind that GPU implementation involves a synchronization overhead when threads want to interact with each other. A new GPU compute request involves a kernel termination and the start of a new kernel by the CPU which can add significant delays if the application was to terminate and restart frequently, or regularly triggered for each packet event.

FPGA implementation provides a high degree of flexibility to define a custom logic on hardware. However, most FPGAs are connected to the CPU through the PCIe, which can be a bottleneck for large interactive computing between host CPU and FPGA Koehler *et al.* (2008). The choice of programming technology, I/O bandwidth, compute speed, and memory requirements of the FPGA determines which NF applications can be accelerated on an FPGA to outperform the CPU.

A critical shortcoming of current custom accelerator technologies is their limited effective utilization of GPUs and FPGAs on the platform during the runtime of application tasks resulting from the heterogeneous application requirements. The custom accelerators that are programmed with a characteristic (small part of an overall NF) to assist the NF (e.g., TCP NF acceleration) are limited to perform the programmed acceleration until they are reprogrammed with a different characteristic (e.g., HTTPS NF acceleration). Therefore, static and dynamic reconfigurations of custom accelerators can result in varying hardware accelerator utilization. One possible solution is to establish an open-source marketplace for the acceleration libraries, software-packages, and application-specific binaries, to enable programmable accelerators which can be reconfigured at runtime to begin acceleration based on dynamic workload demands. One effort in this direction are the FPGA designs to support dynamic run-time reconfiguration through binary files which are commonly referred to as *partial reconfiguration* Intel Corporation (2020c) for run-time reconfiguration processes, and *personas* Vipin and Fahmy (2018) for binary files. A further extension of partial reconfiguration and personas is to enable applications to dynamically choose personas based on application-specific hardware acceleration requirements for both FPGAs and GPUs, and to have common task scheduling between CPUs and custom accelerators.

3.3.5 Dedicated Accelerators

Custom GPU and FPGA accelerators provide a platform to dynamically design, program, and configure the accelerator functionalities during the system run-time. In contrast, the functionalities of dedicated accelerators are fixed and built to perform a unique set of tasks with very high efficiency. Dedicated accelerators often exceed the power efficiency and performance characteristics of CPU, GPU, and FPGA implementations. Therefore, if efficiency is of highest priority for an NF implementation, then the NF computations should be offloaded to dedicated accelerators. Dedicated hardware accelerators are implemented as an Application Specific Integrated Circuit (ASIC) to form a system-on-chip. ASIC is a general technology for silicon design which is also used in the FPGA silicon design; therefore, ASICs can be categorized as: *i*) full-custom, which has pre-designed logic circuits for the entire function acceleration, and *ii*) semi-custom, where only certain logic blocks are designed as an ASIC while allowing programmability to connect and configure these logic blocks, e.g., through an FPGA.

A dedicated accelerator offers no programming flexibility due to the hardware ASIC implementation. Therefore, dedicated accelerators generally implement a set of characteristics (small parts of overall NFs) that can used by heterogeneous applications. For instance, for hardware acceleration of the AES-GCM encryption algorithm, this specific algorithm can be programmed on an FPGA or GPU; in contrast, on a dedicated accelerator there would be a list of algorithms that are supported, and we select a specific algorithm based on the application demands.

A wide variety of dedicated hardware accelerators have been developed to accelerate a wide range of general computing functions, e.g., simulations Xiao *et al.* (2019) and graph processing Gui *et al.* (2019). To the best of our knowledge, there is no prior survey of dedicated hardware accelerators for NFs. This section comprehensively surveys dedicated NF hardware accelerators.

Cryptography and Compression Accelerator (CCA)

Cryptography encodes clear (plain-text) data into cipher-text with a key such that the cipher-text is almost impossible to decode into clear data without the key. As data communication has become an indispensable part of everyday living (e.g., medical care and business activities), two aspects of data protection have become highly important: *i*) privacy, to protect data from eavesdropping, and to protect the sender and receiver information; and *ii*) data integrity to ensure the data was not modified by anyone other than sender or receiver. One of the most widely known cryptography applications in NF development is HTTPS Scheitle *et al.* (2018) for securing transmissions of content between two NFs, such as VNF to VNF, Container Network Function (CNF) to CNF, and CNF to VNF. While cryptography mechanisms address privacy and integrity, compression addresses the data sparsity in binary form to reduce the size of data by exploiting the source entropy. Data compression is widely used from local storage to end-to-end communication for reducing disk space usage and link bandwidth usage, respectively. Therefore, cryptography and compression have become of vital importance in NF deployment. However, the downside of cryptography and compression are the resulting computing requirement, processing latency, and data size increase due to encryption.

Cavium Nitrox® Nitrox Marvell (2020) is a hardware accelerator from Cavium (now Marvell) that is external to the CPU and connects via the PCIe to the CPU for accelerating cryptography and compression NFs. The acceleration is enabled through a software library that interfaces via APIs with the device driver and applications. The APIs are specifically designed to support application and network protocol specific security and compression software libraries, such as OpenSSL, OpenSSH, IPSec, and ZLib. In a typical end-to-end implementation, an application makes a function call (during process/thread execution on CPUs) to an application-specific library API, which then generates an API call to the accelerator-specific library, which offloads the task to the accelerator with the help of an accelerator-device driver on the OS. Nitrox consists of 64 general-purpose RISC processors that can be programmed for different application-specific algorithms. The processor cores are interconnected with an on-chip interconnect (see Sec. 3.3.2) with several compression engine instances to achieve concurrent processing. Nitrox acceleration per device achieves 40 Gbps for IPsec, 300K Rivest-Shamir-Adleman (RSA) Operations/second (Ops/s) for 1024 bit keys, and 25 Gbps for GZIP/LZS compression along with support for single root input/output virtualization (SR-IOV) **?**Pitaev *et al.* (2018) virtualization.

Intel® Quick Assist Technology® Similarly, to address the cryptography and compression computing needs, the Intel® Quick Assist Technology® (QAT) Intel Corp. (2020b) provides a hardware acceleration for both cryptography and compression specifically focusing on network security, i.e., encryption and decryption, routing, storage, and big data processing. The QAT has been specially designed to perform symmetric encryption and authentication, asymmetric encryption, digital signatures, Rivest-Shamir-Adleman (RSA), Diffie-Hellman (DH), and Elliptic-curve cryptography (ECC), lossless data compression (such as DEFLATE), and wireless standards encryption (such as KASUMI, Snow3G and ZUC) Intel Corp. (2019c). The QAT is also used for L3 protocol accelerations, such as IPSec, whereby the packet processing for encryption and decryption of each packet is performed by the QAT. A key differentiation of the QAT from Nitrox is the QAT support for CPU on-die integrated device acceleration, such that the power efficiency and I/O performance can be higher with the QAT as compared to the CPU-external Nitrox accelerator.

Data Streaming Accelerator (DSA)

The management of softwarized NF entities depends mainly on the orchestration framework for the management of softwarized NFs. The management of softwarized NFs typically includes the instantiation, migration, and tear-down (termination) of NFs on GPC infrastructures. These NF management tasks are highly data driven as the management process involves the movement of an NF image in the form of an application executable, Virtual Machine (VM) image, or a container image from a GPC node to another GPC node. Such an NF image movement essentially results in a memory transaction operation on a large block of data, such as copy, duplicate, and move, which is traditionally performed by a CPU. Therefore, to assist in these CPU intensive memory operations, a dedicated hardware Data Streaming Accelerator

Table 3.4: Summary of Data Stream Accelerator (Dsa) Opcodes.

Operations	Type	Description
Move	**Memory**	Transfer data from src. to dst. (range: main memory or MIMO)
	CRC Generation	Generate CRC checksum on the transferred data
	DIF	Data Integrity Field (DIF) check DIF insert, strip or update while data transfer
	Dualcast	Copy data simultaneously to two destination locations
Compare	**Memory**	Two source buffers and return whether the buffers are identical
	Delta Record Creator	Contains the difference between the original and modified buffers
	Delta Record Merge	Merge delta record with the original source buffer to produce a copy of the modified buffer at the destination location
	Pattern/Zero Detect	Special case of compare where instead of the second input buffer, an 8-byte pattern is specified.
Flush	**Cache**	Evict all lines in given address range from all levels of CPU caches

(DSA) Intel Corp. (2019a) has been introduced. The DSA functions are summarized in Table 3.4, and the internal DSA blocks have been illustrated in Fig. 3.20.

The DSA functions that are most relevant for NF management are:

i) The memory move function helps with moving an NF image from one memory location to another within the DRAM of a system, or on an external disk location.

ii) The dualcast function helps with simultaneously copying a NF image on memory to multiple locations, for instance, for scaling up of VMs or containers to multiple locations for load balancing.

iii) The memory compare function compares two memory regions and provides feedback on whether the two regions match or not, and where (memory location) the first mismatch occurs. This feature is useful for checking if a VM or container image has been modified or updated before saving or moving the image to a different location.

iv) The delta record creator function creates a record of differences between two memory regions, which helps with capturing the changes between two VM images. For instance, the delta record function can compare a running VM or container with an offline base image on a disk. The offline base image will be made to run by the OS, which has the running context. Then, we can save the VM or container as a new "base" image, so as to capture changes during run-time to be used later.

v) The delta record merge function applies the delta-record generated by the delta record create function consisting of differences between two memory regions to equate two of the involved memory regions. This function helps with VM and container migration, whereby the generated delta-record can be applied to the VM/Container base image to equate between running image at one node/location to another, essentially migrating a VM/container.

High Bandwidth Memory (HBM)

The memory unit (i.e., DDR) is the closest external component to the CPU. The memory unit typically connects to the CPU with a very high speed interconnect as compared to all other external interconnects (e.g., PCIe) on the platform. While the scaling of computing by adding more cores is relatively easy to design, the utilization of larger memory hardware is fundamentally limited by the memory access speed over the interconnect. Therefore, increasing the bandwidth and reducing the latency of the interconnect determines the effective utilization of the CPU computing capabilities. High Bandwidth Memory (HBM) Macri (2015) has been introduced by AMD® to increase the total capacity as well the total access bandwidth between the CPU and memory. For instance, the DDR5 with two memory channels supports peak speeds of 51.2 GB/s per DRAM module; whereas, the latest HBM2E version is expected to reach peak speeds of 460 GB/s. The increase in memory density and speed is achieved through vertical DRAM die-stacking, up to 8 DRAM dies high. The resulting 3D memory store cube is interconnected by a novel Through-Silicon Vias (TSVs) Jeddeloh and Keeth (2012) technology.

Hardware Queue Manager (HQM)

The normal OS and application operations involve interactions of multiple processes and threads to exchange information. The communication between processes and threads involves shared memory, queues, and dedicated software communication frameworks. NF applications share the packet data between multiple threads to process multiple layers of the networking protocol stack and applications. For instance, the TCP/IP protocol functions are processed by one process, while the packet data is typically exchanged between these processes through dedicated or shared queues. Dedicated queues require large memory along with queue management mechanisms. On the other hand, shared queues require synchronization between multiple threads and processes while writing and reading from the shared queue. Allocating a dedicated queue to every process and thread is practically impossible; therefore, in practice, despite the synchronization requirement, shared queues are extensively used because of their relatively easy implementation and efficient memory usage. However, as the number of threads and processes accessing a single shared queue increases, the synchronization among the threads to write and read in sequence incurs significant delays and management overhead.

The Hardware Queue Manager (HQM) accelerator Power *et al.* (2019); Wang *et al.* (2017b) proposed by Intel® implements the shared and dedicated queues in hardware to exchange data and information between threads and processes. The HQM implements hardware queue instances as required by the applications such that multiple producer threads/processes write to queues, and multiple consumer threads/processes read from queues. Producer threads/processes generate the data that can be intended for multiple consumer threads/processes. The HQM delivers the data then to the consumer threads for data consumption following policies that optimize the consumer thread selection based on power utilization McDonnell *et al.* (2019), workload balancing, and availability. The HQM can also assist in the scheduling of accelerator tasks by the CPU threads and processes among multiple instances of hardware accelerators.

Summary of Dedicated Accelerators

Dedicated accelerators provide the highest performance both in terms of throughput and latency along with power savings due to the efficient ASIC hardware implementation as compared to software execution. The common downsides of hardware acceleration are the cost of the accelerator support and the lack of flexibility in terms of programming the accelerator function.

A critical pitfall of dedicated accelerators is the limitation of hardware capabilities. For instance, a dedicated cryptography and compression accelerator only supports a finite set of encryption and compression algorithms. If an application demands a specific algorithm that is not supported by the hardware, then acceleration has to fallback to software execution which may increase the total execution cost even with the accelerator.

Another key pitfall is to overlook the overhead of the hardware offloading process which involves memory transactions from the DRAM to the accelerator for computing and for storing the result. If the data computation that is being scheduled on an accelerator is very small, then the total overhead of moving the data between the accelerator and memory might outweigh the offloading benefit. Therefore, an offload engine has to determine whether it is worthwhile to use an accelerator for a particular computation.

Dedicated accelerators perform a finite set of operations very efficiently in hardware as opposed to software implementations running on the CPU. Therefore, the limitations of current dedicated accelerators are: *i*) acceleration support for only a finite set of operations, and *ii*) finite acceleration capacity (i.e., static hardware resources). One way to address these limitations is to design heterogeneous modules within a dedicated hardware accelerator device to support a large set of operations. Also, the dedicated hardware accelerator device should have increased hardware resources; however, the actually utilized hardware modules (within the device) should be selected at run-time based on the application requirements to operate within supported I/O link capacities (e.g., PCIe).

3.3.6 Infrastructure

SmartNIC

The Network Interface Card (NIC, which is also referred to as Network Interface Controller) is responsible for transmitting and receiving packets to and from the network, along with the processing of the IP packets before they are delivered to the OS network driver for further processing prior to being handed over to the application data interpretation. Typical network infrastructures of server platforms connect a GPC node with multiple NICs. The NICs are external hardware components that are connected to the platform via the PCIe interfaces. NICs implement standard physical (PHY, Layer 1), data link (MAC, Layer 2), and Internet Protocol (IP, Layer 3) protocol layer functions. The IP packets are transported from the local memory of the PCIe device to the system memory as PCIe transactions in the network downlink direction (i.e., from the network to the application).

If there is an accelerator in the packet processing pipeline, e.g., for decrypting an IP Security (IPSec) or MAC Security (MACSec) packet, the packet needs to be copied from the system memory to the accelerator memory once the PCIe DMA transfer to the system memory is completed. The system memory to accelerator memory copying adds an additional memory transfer step which contributes towards the overhead in the overall processing pipeline. Embedding an acceleration function into the NIC allows the packets to be processed as they arrive from the network at the NIC while avoiding this additional memory transfer step, thereby improving the overall packet processing efficiency.

A Smart-Network Interface Controller (SmartNIC) Le *et al.* (2017); Eran *et al.* (2019) not only implements dedicated hardware acceleration at the NIC, but also general-purpose custom accelerators, such as FPGA units, which can be programmed to perform user defined acceleration on arriving packets. FPGAs on SmartNICs can also be configured at run-time, resulting in a dynamically adaptive packet processing engine that the responsive to application needs. An embedded-Switch (eSwitch) is another acceleration function that implements a data link layer (Layer 2) switch function on the SmartNIC to forward MAC frames between NIC ports. This method of processing the packets as they arrive at the NIC is also termed “in-line” processing, whereas the traditional method with the additional memory transfer to the accelerator memory is termed “look-aside” processing. In addition to programmability, the current state-of-the-art SmartNICs are capable of very high-speed packet processing on the order of 400 Gbps Choi *et al.* (2019b) while supporting advanced protocols, such as

Infiniband and Remote-DMA (RDMA) Chen *et al.* (2016).

Non-Transparent Bridge (NTB)

A PCIe bridge (or switch) connects different PCIe buses and forwards PCIe packets between buses, whereby buses are typically terminated with an endpoint. As opposed to a PCIe bridge, a Non-Transparent Bridge (NTB) Regula (2004) extends the PCIe connectivity to external platforms by allowing two different platforms to communicate with each other. The “Non-Transparent” properties are associated with the NTB in that CPUs that connect to an NTB appear as endpoints to each other More specifically, for the regular bridge, all components, e.g., memory, I/O devices, and system details, on either side of the regular bridge are visible to either side across the regular bridge. In contrast, with the non-transparent bridge, one side can only interact with the CPU on other side; CPUs on either side do not see any I/O devices, nor the Root Ports (RPs) at the other side. However, the “non-transparent bridge” itself is visible to the OS running on either side.

A PCIe memory read or write instruction translates to a memory access from a peer node, thereby enabling platform-to-platform communication. The NTB driver on an OS can be made aware to use doorbell (i.e., interrupt) notifications through registers to gain the remote CPU’s attention. A set of common registers are available to each NTB endpoint as shared memory for management.

The NTB benefits extend beyond the support of the PCIe connectivity across multiple platforms; more generally, NTB provides a low-cost implementation of remote memory access, can seek CPU attention on another platform, can offload computations from one CPU to another CPU, and gain indirect access to remote peer resources, including accelerators and network connectivity. The NTB communication over the underlying PCIe supports higher line-rate speeds and is more power efficient than traditional Ethernet/IP connectivity enabled by NICs; therefore NTB provides an economical solution for short distance communication via the PCIe interfaces. One of the key application of NTB for NF applications is to extend the NTB to support RDMA and Infiniband protocols by running as a Non-Transparent RDMA (NTRDMA).

Summary of Infrastructure

Infrastructure enables platforms to communicate with external computing entities through Ethernet/IP, SmartNIC, and NTB connections. As NF applications highly dependent on communication with other nodes, the communication infrastructure should be able to flexibly reconfigure the communications characteristics to the changing needs of applications. The SmartNIC is able to provide support for both NIC configurability and acceleration to offload CPU computations to the NIC. However, the SmartNIC should still be cost efficient in improving overall adaptability. The programmability of custom acceleration at the NIC should not incur excessive hardware cost to support a wide range of functions ranging from security to switching, and to packet filtering applications

In contrast to the SmartNIC, the NTB is a fixed implementation that runs on the PCIe protocol which supports much higher bandwidth than point-to-point Ethernet connections; however, the NTB is limited to a very short range due to the limited PCIe bus lengths. Additional pitfalls of acceleration at the SmartNIC include misconfiguration and offload costs for small payloads.

Traditionally, infrastructure design has been viewed as an independent development domain that is decoupled from the platform components, mainly CPU, interconnects, memory, and accelerators. For instance, SmartNIC design considerations, such as supported bandwidth and protocol technologies (e.g., Infiniband), traditionally do not consider the CPU architectural features, such as Instruction Set Acceleration

(ISAcc, Section 3.3.1), or system memory capabilities, such as NV-NAND persistent memory (Section 3.3.3). As a result, there is a heterogeneous landscape of platform and infrastructure designs, whereby future infrastructure designs are mainly focused on programmable data paths and supporting higher bandwidth with lower latencies. An interesting future development direction is to exploit synergies between platform component and infrastructure designs to achieve cross-component optimizations. Cross-component design optimizations, e.g., in-memory infrastructure processing or ISAcc for packet and protocol processing, could potentially improve the flexibility, latency, bandwidth, and power efficiencies.

3.3.7 Summary and Discussion

In Sec. 3.3 we have surveyed enabling technologies for platform and infrastructure components for the deployment of NFs on GPC infrastructures. A critical pitfall of NF softwarization is to overlook strict QoS constraints in the designs; QoS constraints are critical as software entities dependent on OSs and hypervisors for resource allocation to meet performance demands. OSs are traditionally designed to provide best effort services to applications which could severely impede the QoS of NF applications in the presence of saturated workloads on the OS.

CPU strategies, such as ISAcc, CPU pinning, and CPU clock frequency speed-ups enable NFs to achieve adequate performance characteristics on GPC platforms. Along with CPU processing enhancements, memory access to load and store data for processing by the CPU can impact the overall throughput and latency performance. Memory access can be improved with caching and higher CPU-to-memory interconnect bandwidth. Cache coherency is a strategy in which caches at various locations, such as multiple cache levels across cores and PCIe device caches, are updated with the latest updates of modified data across all the caches. Cache coherency across multiple cores within the same socket is maintained by 2D mesh interconnects (in case of Intel®) and Scalable Data Fabric (SDF) (in case of AMD®). Whereas, coherency across sockets is achieved through UPI interconnects, and for I/O devices through AXI ACL or CXL links.

The DDR5 and PCIe Gen5 provide high bandwidths for large data transactions to effectively utilize compute resources at CPUs as well as custom and dedicated accelerators. NV-NAND technology provides cost effective solutions for fast non-volatile memory that can be used as an extension to DRAM, second-level memory for DRAM, or as a storage unit assisting both CPU and accelerators in their computing needs. In-Memory accelerators extend the memory device to include accelerator functions to save the data transfer time between accelerator and memory device. A custom accelerator GPU provides programmability for high performance computing for concurrent tasks, while an FPGA provides close to hardware level performance along with high degrees of configurability and flexibility. In contrast to custom accelerators, dedicated accelerators provide the best performance at the cost of reduced flexibility. Based on all the enabling technologies offered on a platform, an NF function design should comprehensively consider the hardware support to effectively run the application to achieve the best performance.

3.4 Research Studies on Hardware-Accelerated Platforms and Infrastructures for NF Implementation

This section surveys the research studies on hardware-accelerated platforms and infrastructures for implementing NFs. While the enabling technologies provide the underlying state-of-the-art techniques to accelerate NFs, we survey the enhancements to the enabling technologies and the investigations of the related fundamental trade-offs in the research domain in this section. The structure of this section follows our classification of the research studies as illustrated in Fig. 3.23.

3.4.1 Computing Architecture

The computing architecture advances in both CISC and RISC directly impact the execution of software, such as applications and VMs that implement NFs. The CISC architecture research has mainly focused on enhancing performance, while the RISC architecture research has mainly focused on the power consumption, size of the chip, and cost of the overall system.

CISC

Generally, computing architecture advances are driven by corporations that dominate the design and development of computing processors, such as AMD®, Intel®, and ARM®. One such enhancement was presented by Clark et al. of AMD® Clark (2016); **?** who designed a new Zen computing architecture to advance the capabilities of the x86 CISC architecture, primarily targeting Instruction Set (IS) computing enhancements. The Zen architecture aims to improve CPU operations with floating point computations and frequent cache accesses. The Zen architecture includes improvements to the core engine, cache system, and power management which improve the instruction per cycle (IPC) performance up to 40%. Architecturally, the Zen architecture core comprises one floating point unit and one integer engine per core. The integer clusters have six pipes which connect to four Arithmetic Logic Units (ALUs) and two Address Generation Units (AGUs), see Fig. 3.24.

The ALUs collaborate with the L1-Data (L1D) cache to perform the data computations, while the Address Generation Units (AGUs) collaborate with the L1-Instruction (L1-I) cache to perform the address computations. Table 3.5 compares the cache sizes and access ways of different state-of-the-art x86 CISC architectures. The enhancements of the Zen architecture are applied to the predecessor family of cores referred to as AMD® Bulldozer; the Zen implements address computing to access system memory based on AGUs with two 16-byte loads and one 16-byte store per cycle via a 32 KB 8-way set associative write-back L1D cache. The load/store cache operations in the Zen architecture have exhibited lower latency compared to the AMD® Bulldozer cores. This unique Zen cache design allows NF workloads to run in both high precision and low precision arithmetic based on the packet processing computing needs. For instance, applications involving low precision computations, such as packet scheduling, load balancing, and randomization can utilize the integer based ALU; while high precession computing for traffic shaping can run on the floating point ALUs.

RISC

In contrast to the CISC architectures which focus typically on large-scale general-purpose computations, e.g., for laptop, desktop, and server processors, the RISC architectures have typically been adopted for low-power processors for applications that run on hand-held and other entertainment devices. Concomitantly, the RISC architecture has typically, also been adopted for small auxiliary computing units for module controllers and acceleration devices. The RISC architecture provides a supportive computing framework for designing acceleration computing units that are traditionally implemented as custom accelerators, such as the Intel® QAT® and DSA (see Sec. 3.3.5), due to the power and space efficient RISC architectural characteristics.

Typically, network applications involve direct packet processing at the NIC to support line-rate speeds. To address the present needs of NFs, specifically with the proliferation of Software Defined Networking (SDN), reconfigurable compute hardware is almost a necessity. However, reconfigurable computing infrastructures reserve a fraction of the hardware resources to support flexibility while dedicated computing

Table 3.5: Cache Technologies Directly Impact the Memory Access Times Which Are Critical for Latency-sensitive Networking Applications As Well as for Delivering Ultra Low Latencies (Ull) as Outlined in The 5g Standards. The State-of-art Enhancements to Cache Technologies Are Compared in the Table, Whereby Larger Cache Sizes and Larger Cache Access Ways, Improve the Capabilities of the Processor To Support Low Latency Workloads. The L1 Instruction (L1i) Cache Allows the Instructions That Correspond to Nf Application Tasks To Be Fetched, Cached, and Executed Locally on the Core, While the L1 Data (L1d) Cache Supports the Corresponding Data Caching.

Cache Level	**Bulldozer® FX-8150**	**ZEN®**	**Broadwell-E® i7-6950X**	**Skylake® i7-6700K**
L1l	64 KB 2-Way per module	64 KB 4−Way	32 KB 8−way	32 KB 8−way
L1D	16 KB 2-Way Write Through	32 KB 2-Way Write Back	32 KB 8-Way Write Back	32 KB 8-Way Write Back
L2	2 MB 16-Way per module	512 KB 8-way	256 KB 8-way	256 KB 4-way
L3	1 MB/core 64-way	1/2 MB/core 16-way	2.5 MB/core 16/20-way	2 MB/core 16-way

infrastructures (i.e., proprietary networking switches and gateways) utilize the entire hardware resources for computing purposes. To address this challenge of retaining flexibility to reconfigure as well as achieving effective hardware resource utilization, Pontarelli et al. Pontarelli *et al.* (2019) have proposed a Packet Manipulation Processor (PMP) specifically targeting line-rate processing based on the RISC architecture. The RISC compute architecture is adapted to perform fast match operations in an atomic way, while still being able to reconfigure (update) the matching table, thus allowing programmability of routing and forwarding functions. Fig. 3.25 illustrates the RISC based PMP processor functional blocks tailored to perform packet processing. A given packet is parsed and passed through several matching tables before finally being processed by the PMP array to be transmitted over the link. The PMP array feeds back the criteria for matching and selection to the ingress input mixer.

Moving routine software tasks, such as NF packet processing, from the CPU to dedicated hardware lowers overheads and frees up system resources for general-purpose applications. However, large scale distributed applications, such as big data analysis and data replications, are considered as user space applications, and decoupled from the packet processing framework (e.g., Ethernet, switches and routers). As a result, the replication of data across a large number of compute and storage network platforms would consume large amounts of network bandwidth and computing resources on the given platform involved in data replication, storage, and processing tasks. To address this problem, Choi et al. Choi *et al.* (2019a) have proposed a data-plane data replication technique that utilizes RISC based processors to perform the data replication. More specifically, a SmartNIC consisting of 56 RISC processors implements data plane functions to assist in the overall end-to-end data-replication at the application layer. The proposed framework involves three components: *i*) a master node that requests replications using store and retrieve, *ii*) a client node that assists in maintaining connections, and *iii*) data plane witnesses that store and retrieve the actual data. The RISC computations are optimized to perform the simultaneous operations of replication, concurrent with packet parsing, hashing, matching, and forwarding. A testbed implementation showed significant benefits from the RISC based SmartNIC approach as compared to software implementation: the data path latency is reduced to nearly half and the overall system throughput is increased 6.7-fold.

Focusing on validation and function verification of NF application hardware architectures, Herdt et al. Herdt *et al.* (2019) have proposed a framework to test software functions (which can be extended to NFs) on RISC architectures. The proposed Concolic Testing Engine (CTE) enumerates the parameters for the software functions which can be executed over an instruction set simulator on a virtual prototype emulated as a compute processor. The evaluations in Herdt *et al.* (2019) employed the FreeROTS TCP/IP network protocol layer stack for NF testing to effectively identify security vulnerabilities related to buffer overflows.

NFs are supported by OS services to meet their demands for packet processing. As a result, NF applications running on computing hardware (i.e., a CPU) rely on OS task scheduling services. However, as the number of tasks increases, there is an increased overhead to align the tasks for scheduling to be run on CPU based on scheduling policies, especially in meeting strict latency deadlines for packet processing. Some of the mitigation techniques of scheduling overhead involve using simple scheduling strategies, such as round robin and random selection, or to accelerate the scheduling in hardware. While hardware accelerations are promising, the communication between the CPU and the acceleration component would be a limiting factor. One way to reduce the communication burden between the CPU and the acceleration component is to enable the CPU to implement scheduling using Instruction Set (IS) based accelerations as proposed by Morais et al. Morais *et al.* (2019). Morais et al. Morais *et al.* (2019) have designed a RISC based CPU architecture with a custom instruction as part of the IS to perform scheduling operations for the tasks to be run by the OS on the CPU. A test-bed implementation demonstrated latency reductions to one fifth for an 8-core CPU compared to serial task executions. NF applications typically run in containers and VMs on a common infrastructure that require highly parallel hardware executions. The proposed IS based optimization of task scheduling can help in enforcing time critical latency deadlines of tasks to run on CPUs with low overhead.

Multi-Core Optimization

Most systems that execute complex software functions are designed to run executions in concurrent and parallel fashion on both single and multiple computing hardware components (i.e., multi-core processors). A key aspect of efficient multi-core systems is to effectively schedule and utilize resources. Optimization techniques are necessary for the effective resource allocation based on the system and application needs. On a given single core, Single Instruction Multiple Data (SIMD) instructions within a given compute architecture (i.e., RISC or CISC) allow the CPU to operate on multiple data sets with a single instruction. SIMD instructions are highly effective in the designs of ultra-fast Bloom filters which are used in NF applications, such as matching and detecting operations relevant to the packet processing Lu *et al.* (2018). Due to the nature of multiple data sets in the SIMD instruction, the execution latency is relatively longer compared to single datasets.

In an effort to reduce the execution latency, Zhou et al. Zhou *et al.* (2018) have proposed a latency optimization for SIMD operations in multi-core systems based on Ant-Colony Optimization (ACO). The Zhou et al. Zhou *et al.* (2018) ACO maps each core to an ant while the tour construction is accelerated by vector instructions. A proportionate selection approach named Vector-based Roulette Wheel (VRW) allows the grouping of SIMD lanes. The prefix sum for data computations is evaluated in vector-parallel mode, such that the overall performance execution time can be reduced across multiple cores for SIMD operations. The evaluations in Zhou *et al.* (2018) indicate 50-fold improvements of the processing speed in comparison to single-thread CPU execution. NF applications can greatly benefit from SIMD instructions to achieve ultra-low latency in packet processing pipelines.

Latencies in multi-core systems affect the overall system performance, especially for latency-critical packet processing functions. In multi-core systems, the processing latencies typically vary among applications and cores as well as across time. The latencies in multi-core systems depend strongly on the last level cache (LLC). Therefore, the LLC design is a very important issue in multi-core systems. Wang et al. Wang *et al.* (2016b) have proposed a latency sensitivity-based cache partitioning (LSP) framework. The LSP framework, evaluates a latency-sensitivity metric at runtime to adapt the cache partitioning. The latency-sensitivity metric considers both the cache hit rates as well as the latencies for obtaining data from off-chip (in case of cache misses) in conjunction with the sensitivity levels of applications to latencies. The LLC partitioning based on this metric improves the overall throughput by an average of 8% compared to prior state-of-the-art cache partitioning mechanisms.

Core Power and Performance

While it is obvious that multi-core systems consume higher power compared to single-core systems, the system management and resource allocation between multiple cores often results in inefficient power usage on multi-core systems. Power saving strategies, such as power gating and low power modes to put cores with no activity into sleep states, can mitigate energy wastage. NF applications require short response times for processing the incoming packets. Short response times can only be ensured if the processing core is in an active state to immediately start the processing; whereas, from a sleep state, a core would have to go through a wake-up that would consume several clock cycles.

The energy saving technique proposed by Papadimitriou et al. Papadimitriou *et al.* (2017) pro-actively reduces the voltage supplied to the CPUs (specifically, ARM® based cores) of a multi-core system without compromising the operational system characteristics. In the case of too aggressive reduction of the voltage level supplied to CPUs, uncorrectable system errors would lead to system crashes. Therefore, a sustainable level of voltage reduction just to keep the core active at all times even when there is no application processing can be an identified by analyzing the system characterizations. The evaluations in Papadimitriou *et al.* (2017) based on system characterizations show that energy savings close to 20% can be achieved, and close to 40% savings can be achieved if a 25% performance reduction is tolerated.

A more robust way to control the power characteristics is through dynamic fine-grained reconfiguration of voltage and frequency. However, the main challenge in dynamic reconfiguration is that different applications demand different power scaling and hence the requirements should be averaged across all applications running on a core. Dynamic runtime reconfiguration of voltage and frequency is typically controlled by the OS and the system software (i.e., BIOS, in case of thermal run-off). On top of reconfiguration based on averaged requirements, there would still be some scope to improve the overall voltage and frequency if the run-time load can be characterized in advance before the processes are scheduled to run on the cores. Bao et al. Bao *et al.* (2016) have proposed several such techniques where the power profile is characterized specifically for each core, which is then used for voltage and frequency estimations based on the application needs. Subsequently, Bao et al. Bao *et al.* (2016) have evaluated power savings based on profiling of both core power characterizations and the application run-time requirements. The evaluations have shown significant benefits compared to the standard Linux power control mechanism.

More comprehensive search and select algorithms for the optimal voltage and frequency settings for a given core have been examined by Begum et al. Begum *et al.* (2016). Begum et al. Begum *et al.* (2016) have broadly classified the algorithms into: *i*) search methods: exhaustive and relative, and *ii*) selection methods: best performance and adaptive. Exhaustive search sweeps through the entire configuration space to evaluate the performance. Relative search modifies the current configuration and monitors the relative performance changes with the overall goal to incrementally improve the performance. In the best performing selection, the configuration is tuned in a loop to identify the configuration that results in the best performance; whereas, in adaptive selection, the tuning is skipped, and configuration values are applied to achieve a performance within tolerable limits. NF applications can utilize these techniques based on the application needs so as to meet either a strict or a relaxed deadline for packet processing.

Other strategies to support the power and performance characteristics of NF applications, in addition to dynamic voltage and frequency include CPU pinning, as well as horizontal and vertical scaling. CPU pinning corresponds to the static pinning of applications and workloads to a specific core (i.e., no OS scheduling of process). Horizontal scaling increases the resources in terms of the number of allocated systems (e.g., number of allocated VMs), while vertical scaling increases the resources for a given system (e.g., VM) in terms of allocated CPU core, memory, and storage. Krzywda et al. Krzywda *et al.* (2018) have evaluated the relative power and performance characteristics for a deterministic workload across voltage, frequency, CPU pinning, as well as horizontal and vertical scaling. Their evaluations showed a marginal power improvement of about 5% for dynamic voltage and frequency in underloaded servers; whereas on saturated servers, 20% power savings can be achieved at the cost of compromised performance. Similarly, CPU pinning was able to reduce the power consumption by 7% at the cost of compromised performance. The horizontal and vertical scaling reduced latencies, however only for disproportionately large amounts of added resources. Krzywda et al. also found that load balancing strategies have a relatively large impact on the tail latencies when horizontal scaling (i.e., more VMs) are employed.

Power and performance is a critical aspect to NF applications in meeting the latency demands, and therefore should be carefully considered while balancing between power savings and achieving the highest performance. Aggressive power saving strategies can lead to system errors due to voltage variations, which will cause the system to hang or reboot. Allowing applications to control the platform power can create isolation issues. For instance, a power control strategy applied by one application, can affect the performance of other applications. This vulnerability could lead to catastrophic failures of services as multiple isolated environments, such as containers and VMs, could fail due to an overall system failure.

CPU-FPGA

Reconfigurable computing allows compute logic to be modified according to the workload (application) needs to achieve higher efficiency as compared to instruction set (IS) based software execution on a general-purpose processor. Matthews et al. Matthews and Shannon (2017) (see Fig. 3.26) have proposed a design enhancement called Taiga for the RISC-V (pronounced "RISC-Five") architecture, an open source RISC design. In their design enhancement, the IS processor core is integrated with programmable custom compute logic (i.e., FPGA) units, which are referred to as reconfigurable function units. The processor supports a 32 bit base IS capable of multiply and divide operations. Reconfigurable function units can be programmed to have multiple functions that can be defined during run time, and can then be interfaced with the main processors. This approach can lead to a high degree of Instruction Level Parallelism (ILP) supported by a fetch logic and load store unit that are designed with translation look-aside buffers (TLBs) and internal cache support. Different variants have been proposed, e.g., a full configuration version which has

1.5× the minimum configuration version resources based on the overall density of Look Up Tables (LUTs), hardware logic slices, RAM size, and DSP blocks. The evaluations in Matthews and Shannon (2017) successfully validated the processor configurations and identified the critical paths in the design: The write-back data path is the critical path in the minimum configuration system, and the tag hit circuit for address translations through the TLB is the critical path for the full configuration version.

An FPGA component can be interfaced with the main compute component (i.e., core) in the CPU through multiple interfaces. If the FPGA is placed on the fabric that connects to the core, then the applications can benefit from the data locality and cache coherency with the DRAM and CPU. If the FPGA is interfaced with the compute component (core) through a PCIe interface, then there is a memory decoupling, i.e., the device-specific FPGA internal memory is decoupled from CPU core memory (system memory DRAM). Hence, there are significant latency implications in each model of FPGA interfacing with the CPU based on FPGA presence on the core-mesh fabric or I/O interfaces, such as PCIe and USB. Choi et al. Choi *et al.* (2016) have quantitatively studied the impact of the FPGA memory access delay on the end-to-end processing latency. Their study considers the Quick Path Interconnect (QPI) as FPGA-to-core communication path in case of FPGA presence on the processor die (coherent shared memory between CPU and FPGA) and the PCIe interface (private independent memory for both CPU and FPGA) for the external (FPGA) device connectivity. Their evaluations provide insights into latency considerations for meeting application demands. In summary, for the PCIe case, the device to CPU DMA latency is consistently around 160 μs. For the QPI case, the data access through the (shared) cache results in latencies of 70 ns and 60 ns for read and write hits, respectively. The read and write misses correspond to system memory accesses which result in

355 ns and 360 ns for read and write miss, respectively. The latency reduction from 160 μs down to the order of 70 to 360 ns is a significant improvement to support NF applications, especially NF applications that require ultra-low latencies on the order of sub-microseconds.

Abdallah et al. Abdallah *et al.* (2019) (see Fig. 3.27) have proposed an interesting approach to commonly schedule the tasks among heterogeneous compute components, such as CPU and FPGA. This approach allows a software component to use the compute resources based on the relative deadlines and compute requirements of the applications. Genetic algorithms, such as chromosome assignment strategies and a Modified Genetic Algorithm Approach (MGAA), have been utilized to arrive at combinatorial optimization solutions. The goal of the optimization is to allocate tasks across Multi-Processor SoC (MPSoC) for maximizing the resource utilization and minimizing the processing latency of each task. Their evaluations show that common scheduling across heterogeneous compute processors not only improves the application performance, but also achieves better utilization of the computing resources. Their work can be extended to different types of computing resources other than FPGA, such as GPU and ASICs.

NF applications are particularly diverse in nature with requirements spanning from high throughput to short latency requirements; effectively utilizing the heterogeneous computing resources is a key aspect in meeting these diverse NF demands. For instance, Owa et al. Owaida *et al.* (2019) have proposed an FPGA based web search engine hardware acceleration framework, which implements the scoring function as a decision tree ensemble. A web search engine involves processing pipelined functions of computing, scoring, and ranking potential results. The optimization of these pipelines involves reducing intermediate data transfers and accelerating processes through hardware. Evaluations based on optimizations on FPGA based hardware accelerations show a two-fold performance improvement compared to CPU solutions.

In another example, Kekely et al. Kekely *et al.* (2020) proposed an FPGA based packet classification (matching) hardware acceleration to increase the system throughput. Typically, the packet processing pipelines are implemented in parallel to match several packets in one clock cycle so as to decrease the process latency. However, parallel computations require dedicated resources when accelerating on FPGA, decreasing the overall system throughput. Therefore, Kekely et al. Kekely *et al.* (2020) have implemented a hashing based exact match classification on FPGA which can match packets in parallel while utilizing less resources (e.g., memory). As compared to the baseline FPGA implementation, the results show up to 40% memory savings while achieving 99.7% of the baseline throughput.

The performance of an end-to-end application running on an FPGA accelerated system depends on both software and hardware interactions. The overall performance is dictated by the bottlenecked functions which may exist in both software and hardware sub-components. Since it is challenging to run an application and then profile the performance metrics across various processing stages, Karandikar et al. Karandikar *et al.* (2020) have proposed FirePerf, an FPGA-Accelerated hardware simulation framework. FirePerf performs a hardware and software performance profiling by inserting performance counters in function pipelines such that processing hot spots can be identified so as to find the system bottleneck. FirePerf is an out-of-band approach in which the actual simulation process does not impact the running application. The capabilities of FirePerf were demonstrated for an RISC-V Linux kernel-based optimization process which achieved eight-fold improved network bandwidth in terms of application packet processing.

CPU-GPU

Similar to the study by Abdallah et al. Abdallah *et al.* (2019), Nie et al. Nie *et al.* (2019) (see Fig. 3.28) have proposed a task allocation strategy to schedule tasks between heterogeneous computing resources, specifically, CPU and GPU. While a GPU is a general-purpose compute processor designed to execute parallel threads that are independent of each other, not all workloads (application requirements) are suited for parallel execution. For instance, NF applications that simultaneously perform relatively simple operations on multiple packet flows can run in parallel processing threads entirely on GPUs Yi *et al.* (2017). However, the performance characterizations by Yi et al. Yi *et al.* (2017) considered the performance of a GPU alone to show the benefits in comparison to a CPU (but not the performance of the GPU in conjunction with a CPU).

Generally, a given workload cannot be categorized into either fully parallel threaded or fully single threaded in a strict sense. Therefore, there is a scope for task partitioning into parallel and single-threaded sub-tasks Mardani *et al.* (2013); whereby a given task is split into two different task types, namely task types suitable for GPU (parallel threaded) execution and task types for CPU (single-threaded) execution. The evaluation of the task partitioning method proposed in Mardani *et al.* (2013) considers adaptive Sparse Matrix-Vector multiplication (SpMV). A given task is divided into multiple slave processes and these slave processes are scheduled to run either on a CPU or on a GPU depending on the needs of these slave processes. The task computing on the GPU is limited by the data movement speeds between CPU system memory (DRAM) and GPU global memory. To overcome this limitation, the proposed architecture involves double buffering in either direction of the data flow (into and out of the GPU) as well as on either side of the memory regions, i.e., CPU DRAM and GPU global memory. The evaluations indicate 25% increases in the total number of (floating point) operations. Sparse matrix computations are widely used in NF applications, specifically for anomaly detection in traffic analysis Mardani *et al.* (2013) which is applied in packet filtering and DoS attack mitigation.

Summary of Computing Architectures

The computing architecture of a platform defines its computing performance for given power characteristics. Some applications, such as data collection and storage, can tolerate some performance degradations (resulting from CPU load) and are not latency sensitive; whereas, other applications, e.g., the sensor data processing for monitoring a critical event, are both latency and performance sensitive. Generally, the power constraints on the platform are decoupled from the applications. More specifically, the platform initiatives, such as changes of the CPU characteristics, e.g., reduction of the CPU operational frequency to conserve battery power, are generally not transparent to applications running on the CPU. As a result, the applications may suffer from sudden changes of the platform computing performance without any prior notifications from the platform or the OS. Future research should make the platform performance characteristics transparent for the application such that applications could plan ahead to adapt to changing platform characteristics.

Typically, the platform cores are designed following a homogeneous computing architecture type, i.e., either CISC or RISC. Accordingly, the applications are commonly compiled to run optimally on a specific architecture type. Several studies Venkat *et al.* (2019); Venkat and Tullsen (2014); Pan and Naeemi (2015) have investigated heterogeneous architectures that combine both CISC and RISC computing in a single CPU, resulting in a composite instruction set architecture CPU. While heterogeneous architectures attempt to achieve the best of both the RISC (power) and CISC (performance) architecture types, identifying threads based on their requirements and scheduling the threads appropriately on the desired type of core is critical for achieving optimal performance. Therefore, multi-core optimizations should consider extensions to heterogeneous CPUs, as well as GPUs and FPGAs.

3.4.2 Interconnects

Interconnects allow both on-chip and chip-to-chip components to communicate with short latencies and high bandwidth. To put in perspective, the I/O data rate per lane on the DDR1 was 1 Gbps and for the DDR5 it is 5 Gbps (see Table 3.3), whereby there are 16 lanes per DDR chip. These data rates are scaled significantly with 3D stacking of memory [as in the case of High Bandwidth Memory (HBM), see Section 3.3.5]; for example, the total bandwidth scales up to 512 Gbps for a 4 stack die with 128 Gbps bandwidth per die Lee *et al.* (2014). Therefore, the support for these speeds on-chip and chip-to-chip in an energy-efficient manner is of utmost importance. Towards this goal, Mahajan et al. Mahajan *et al.* (2019b,a) have proposed a Embedded Multi-Die Interconnect Bridge (EMIB) to support high die-to-die interconnect bandwidth within a given package. The key differentiator of EMIB is the confined interconnect area usage inside the package. EMIB allows interconnects to be run densely between silicon endpoints, enabling very high data rates (i.e., aggregated bandwidth). EMIB uses thin pieces of silicon with multi-layer Back-End-Of-Line (BEOL) which could be embedded within a substrate to enable localized dense interconnects. NF applications benefit from highly efficient interconnects in supporting both high throughput and short latencies. For instance, Gonzalez et al. Gonzalez *et al.* (2017) have adapted PCIe links to flexibly interface the accelerators with the compute nodes (25 Gb/s) to support NF applications, such as cognitive computing.

Reconfigurable Interconnect

Existing interconnect designs do not support configurability, mainly due to performance issues and design complexities. The compiler complexity increases when translating programs onto reconfigurable functional units (FUs) on an underlying static fabric (which imposes constraints on the placement of inter-communicating FUs). Karunaratne et al. Karunaratne *et al.* (2017) have proposed HyCUBE, a reconfigurable multi-hop interconnect, see Fig. 3.29. HyCUBE is based on a Coarse-Grained Reconfigurable Array (CGRA), which consists of a large array of function units (FUs) that are interconnected by a mesh fabric Ansaloni *et al.* (2010). An interconnect register based communication, in place of buffer queues, can provide single cycle communication between distant FUs. HyCUBE achieves 1.5× the performance-per-watt as compared to a standard NoC and 3× as compared to a static CGRA. The reconfigurability of the interconnect in HyCUBE allows application-based interconnect design between the FUs to be captured through the compiler and scaled according to the NF application needs.

One way to improve the reconfigurable computing efficiency of FPGAs is to effectively manage the data flow between the FUs on the FPGAs. Jain et al. Jain *et al.* (2016a) have proposed a low-overhead interconnect design to improve the data movement efficiency. The design reduces overheads by re-balancing the FU placement based on Data Flow Graph (DFG) strategies. Also, the design exploits interconnect flexibility (i.e., programmability) to effectively counter the data move inefficiencies, e.g., by funneling data flows through linearized processing layers, i.e., in a single direction, either horizontal or vertical, with a minimum number of hops. The proposed design has been applied to develop a DSP compute acceleration methodology, namely a DSP-based efficient Compute Overlay (DeCO). DeCO evaluations indicate up to

96% reduced Look Up Table (LUT) requirements as compared to standard DSP based FPGA implementation, which translates to reduced interconnect usage between FUs. Most NF applications that involve data processing, such as traffic analysis, event prediction, and routing path computation, would require DSP operations. Therefore, DSP function acceleration is an important aspect of NF application deployment.

Yazdanshenas et al. Yazdanshenas and Betz (2018) have studied the impact of interconnect technologies in the case of virtualization of FPGAs in data centers. NF applications in cloud-native deployments use FPGAs in virtualized environments, therefore understanding the relative interconnect performances helps in designing virtualized NF deployments on FPGA based computing nodes with desired interconnect features. Typical challenges in the virtualization of FPGAs are the inherent FPGA features, such as board-specific characteristics, system-level integration differences, and I/O timing, which should be abstracted and hidden from the applications. Towards this end, a shell based approach abstracts all the FPGA component, except the FUs and interconnect fabric, which results in an easy and common interface for virtualization and resource allocation to applications. More specifically, a shell consists of components, such as external memory controller, PCIe controller, Ethernet, power and subsystem management units. Several interconnect technologies, such as soft (i.e. programmable) NoC and hard (i.e., non-programmable) NoC, have been considered in the performance evaluation of shell virtualization in Yazdanshenas and Betz (2018). The evaluations show that shell based virtualization of the traditional bus-based FPGA interconnects results in a 24% reduction of the operating frequency and a 2.78× increase of the wire demand as well as significant routing congestion. With the soft NoC, the operating frequency can be increased compared to the traditional bus-based implementation, but the increased wire demand and routing congestion remain. However, the hard NoC system outperforms both the soft NoC and the bus-based FPGA implementation. The hard NoC is therefore recommended for data center deployments.

3D On-Chip Interconnect

3D chip design allows for the compact packaging of SoCs to effectively utilize the available chip area. However, the higher density of chip components in a SoC comes at the cost of complex interconnect designs. Through Silicon Vias (TVS) is an interconnect technology that runs between stacked chip components. Using TVS technology, Kang et al. Kang *et al.* (2016b) have proposed a new 3D Mesh-of-Tree (MoT) interconnect design to support the 3D stacking of L2 cache layers in a multi-core system, see Fig. 3.30. The 3D MoT switches and interconnects are designed to be reconfigurable in supporting the power-gating (i.e., turn off/on voltage supply to the component) of on-chip components, such as cores, memory blocks, and the routing switches themselves. The adaptability of 3D MoT allows the on-chip components (e.g., L2 cache) to be modulated as the application demands vary with time. The evaluations in Kang *et al.* (2016b) demonstrate that the reconfigurable 3D MoT interconnect design can reduce the energy-delay product by up to 77%. As with the dynamic nature of traffic arrivals for the NF processing, the hardware scaling of resources as the demand scales up and the power gating of components as demand falls can provide an efficient platform to design power-efficient NF processing strategies.

NoC

As the core count of the traditional computing nodes and Multiprocessor System on Chips (MPSoCs) increases to accommodate higher computing requirements of the applications, the interconnects pose a critical limiting path for overall performance increases. Typically, the core-to-core communication is established through high-bandwidth single- and multi-layer bus architecture interconnects. The present state-of-the-art core-to-core communication involves mesh architecture-based interconnects. However, for mesh interconnects, core-to-core communications have not been specifically designed to support other computing components, such as memory, cache, and I/O devices (e.g., GPU and FPGA). A Network-on-Chip (NoC) is able to support both core-to-core communications and other computing components through standard interconnects and switches.

The cost-efficient design of NoC interconnects has been comprehensively discussed in Coppola et al. Coppola *et al.* (2018), where the programmability has been extended to interconnects in addition to compute units, resulting in an Interconnect Programming Unit (IPU). However, traditional NoCs have static bandwidth for the interconnects which can cause performance bottlenecks. Addressing this issue, Tsai et al. Tsai *et al.* (2012) have proposed a Bi-directional NoC (BiNoC) architecture which supports dynamically self-reconfigurable bidirectional channels. The BiNoC architecture involves common in-out ports fed by data in either direction with a self-loop-path through the internal crossbars while the input flow is supported by an input buffer. This BiNoC design allows the traffic to loop-back within the same switch and port. For a given workload, the bandwidth utilization over the BiNoC is typically significantly lower than over a traditional NoC. NF applications that require high data-rate processing can benefit from the high data-rate I/O through the compute components provided by the BiNoC.

Goehringer et al. Goehringer *et al.* (2011) have proposed an adaptive memory access design to facilitate data movements between multiple FPGA compute processors (cores). Typically, memory access to the system memory is serialized, resulting in increased memory read and write latencies when many clients try to simultaneously access the memory. In the adaptive memory access design, the adaptive memory-core manages the resource allocation to each FPGA core. Each FPGA core (client) is allocated a priority, whereby the priority of each processor can be changed dynamically. Additionally, the number of processors connected to the adaptive memory-core can vary based on the application demands. The adaptive memory-core separates the memory into regions that are core-specific individually accessed by the NoC fabric. Also, the adaptive memory-core maintains a separate address generator for each core, thereby allowing multiple FPGA cores to simultaneously access memory regions.

3D NoC

Traditional NoCs connect compute nodes on a 2D planar routing and switching grid, thus limiting the total number of compute notes that can be supported for a given surface area. A 3D NoC extends the switching network to the third dimension, thus supporting several 2D planar girds for a given surface area dimension, increasing the density of the total number of compute nodes. However, one of the challenges of the 3D NoC design is the performance degradation over time due to the aging of circuits primarily from Bias Temperature Instability (BTI) causing gate-delay degradation. Furthermore, continued operations of a 3D NoC with higher gate-delays could result in the failure of the interconnect fabric. A potential solution to retain the 3D NoC performance is to increase the voltage; however, an increased voltage accelerates the circuit aging process. In addition to an increased voltage, electro-migration (gradual movement of charged particles due to momentum transfer) on the 3D Power Delivery Network (PDN) also reduces the chip lifetime. Raparti et al. Raparti *et al.* (2017) have evaluated the aging process of the interconnect circuit as well as PDN network, and proposed a run time framework, ARTEMIS, for application mapping and voltage-scaling to extend the overall chip lifetime. Typically, the use of an 3D NoC is asymmetric due to uneven scheduling of computing tasks, leading to uneven aging of the 3D NoC, as illustrated in Fig. 3.31. ARTEMIS enables the application to use 3D NoC symmetrically through an optimization process, thereby spreading out the aging process evenly throughout the 3D NoC grid. ARTEMIS evaluations show that the chip lifetime can be extended by 25% as compared to uneven aging of 3D NoC.

Similar to the uneven aging of circuits and PDN network, a single transistor failure in a 3D NoC impacts the performance of the entire chip due to the tight coupling of networks in a 3D NoC. Therefore, a 3D NoC design should include resilient features for a large number of transient, intermittent, and permanent faults in the 3D NoC grid. To this end, Ahmed et al. Ahmed and Abdallah (2016) have presented a novel routing algorithm and an adaptive fault-tolerant architecture for multi-core 3D NoC systems. More specifically, the architecture proposed by Ahmed et al. Ahmed and Abdallah (2016) implements a Random-Access-Buffer mechanism to identify the faulty buffers on the switching network and to isolate them in a routing algorithm that avoids invalid paths. Though the reliability of the 3D NoC is improved, the design costs 28% in terms of area and 12.5% in power overhead.

Wireless NoC

A Heterogeneous System Architecture (HSA) allows different computing components, such as CPU, GPU, and FPGA, to co-exist on the same platform to realize a single system. These heterogeneous components require heterogeneous connectivities. Also, the run-time interconnect requirements typically change dynamically with the load. Moreover, when the distance (number of hops in mesh) between two heterogeneous components increases, the communication latency often increases. Gade at al. Gade and Deb (2016) have proposed a Hybrid Wireless NoC (HyWin), as illustrated in Fig. 3.32, to address the latency and flexibility of NoC interconnects for an HSA. The

HyWin architecture consists of sandboxed (i.e., inside a securely isolated environment) heterogeneous sub-networks, which are connected at a first (underlying) level through a regular NoC. Processing subsystems are then interconnected through a second level over millimeter (mm) wave wireless links. The resource usage of a physical (wired) link at the underlying level avoids conflicts with the wireless layer. The wireless link is especially helpful in establishing long-range low-latency low-energy inter-subsystem connectivity, which can facilitate access to system memory and lower level caches by the processing subsystems. The CPU-GPU HSA testbed evaluations in Gade and Deb (2016) show application performance gains of 29% and latency reductions to one half with HyWin as compared to a baseline mesh architecture. Moreover, HyWin reduces the energy consumption by approximately 65% and the total area by about 17%. A related hybrid wireless NoC architecture has been proposed in Bahrami *et al.* (2019), while other recent related studies have examined scalability Mnejja *et al.* (2020), low-latency Ouyang *et al.* (2020), and energy efficiency Catania *et al.* (2017).

Similarly, for planar interconnected circuits (commonly used for chip-to-chip packaging), Yu et al. Yu *et al.* (2016) have proposed a wide-bandwidth G (millimeter) band interconnect with minimized insertion loss. The proposed interconnect design is compatible with standard packaging techniques, and can be extended to THz frequencies supported by a low insertion loss of 4.9 dB with a 9.7 GHz frequency and 1 dB bandwidth. Further advances in millimeter wave NoCs have recently been reviewed in Gade *et al.* (2019).

A common pitfall for wireless NoC design is to not consider the wireless errors, as the errors can increase the end-to-end latency over wireless links, resulting from retransmissions. More specifically, the protocols to correct the transmission errors beyond the forward error corrections require higher layer flow control, with acknowledgment mode operations (e.g., Automatic Repeat Request protocols or TCP). The reporting of errors back to the source and receiving the retransmissions would increase the overall memory-to-memory transactions (moves or copies) of data through a wireless NoC.

Software Defined NoC (SD-NoC)

Software Defined Networking (SDN) separates the control plane from the data plane of routing and forwarding elements. The control plane is further (logically) centralized to dynamically evaluate the routing policies Andreades *et al.* (2019). The extension of the SDN principles to an NoC is referred to as Software Defined NoC (SD-NoC). Application needs can be captured by the control plane of the NoC routers, which then program the data-plane routing policies across the interconnects between the compute components. One of the bottlenecks in SDN designs is the control plane complexity when there are many routing elements.

In the case of Chip Multi-Processors (CMP) with several thousand cores, the SD-NoC design becomes particularly challenging. Additionally, when the threads running on each of these cores try to exchange data with each other, the interconnect usage can saturate, reducing the overall CMP benefits. Addressing this problem, Scionti et al. Scionti *et al.* (2016, 2018) have presented an SD-NoC architecture based on data-driven Program eXecution Models (PXMs) to reconfigure the interconnect while retaining the hard-wired 2D mesh topology. More specifically, virtual topologies, such as local and global rings Li *et al.* (2019c), are generated and overlayed on the hard-wired 2D mesh to support changing application demands. This approach has resulted in power savings of over 70% while the chip area was reduced by nearly 40%. A related SD-NoC based on the Integrated Processing NoC System (IPNoCSys) execution model Fernandes *et al.* (2009a,b) has been examined in Nunes and Kreutz (2019).

Generally, the SD-NoC designs are configured to be specific to an application in use, and cannot be reused across multiple applications. To overcome this limitation, Sandoval et al. Sandoval-Arechiga *et al.* (2016) have proposed an SD-NoC architecture that enables on-the-fly reconfiguration of the interconnect fabric. This on-the-fly reconfiguration design can be adapted to other applications with minimal changes, reducing the non-recurring engineering cost. The main feature of their architecture is configurable routing which is achieved through a two-stage pipeline that can buffer and route in one clock cycle, and arbitrate and forward in the other cycle. The controller and switch were designed to support flow-based routing with flow IDs. Global average delay, throughput, and configuration time were evaluated for various simple routing algorithms and a wide range of packet inject rate patterns. Deterministic/fixed routing between processing elements was shown to perform better than adaptive routing. Deterministic/fixed routing has a map of the routing path between every source and destination pair; the routing paths are programmed into the NoC fabric and remain active for the entire system life time. In contrast, fully adaptive routing dynamically adapts the packet routing based on the injection rates. For high packet inject rates, the path evaluations select longer and disjoint paths to effectively spread the packets throughout the fabric so as to accommodate the increasing traffic; which may not result in a efficient end-to-end path for packet flow. In both cases, deterministic/fixed routing and adaptive routing, the on-the-fly reconfiguration enables the NoC to be programmed, i.e., the fabric logic to change according to the traffic demands, so that even the deterministic/fixed paths are reconfigured based on need. A distributed SDN architecture for controlling the reconfigurations in an efficient scalable manner has been examined in Ruaro *et al.* (2019).

These advanced reconfigurations of on-chip interconnects allow NF applications to adapt to varying networking loads in order to achieve desired processing responses latencies for arriving packet while employing restrictive resource usage to save power and improve overall efficiency.

Optical Interconnects

Interconnects based on Silicon Photonic (SiPh) technologies achieve—for the same power consumption—several orders of magnitude higher throughput than electrical interconnects. Therefore, optical interconnects are seen as a potential solution for meeting the demands of applications requiring large data transactions between computing elements Bashir *et al.* (2019); Reza (2017); Yahya *et al.* (2020). SiPh offers solutions for both on-chip and chip-to-chip interconnects. For instance, Hsu et al. Hsu *et al.* (2018) have proposed a 2.6 Tbits/sec on-chip interconnect with Mode-Division Multiplexing (MDM) with a Pulse-Amplitude Modulation (PAM) signal. To achieve the speeds of 2.6 Tbits/sec, 14 wavelengths in three modes supporting 64 Gbps are aggregated with hard decision forward-error-correction threshold decoding.

Gu et al. Gu *et al.* (2017) have proposed a circuit-switched on-chip Optical NoC (ONoC) architecture providing an optical interconnect grid with reuse of optical resources. As compared to a traditional NoC, an ONoC does not inherently support buffers within routers to store and forward; therefore, the transmissions have to be circuit switched. The ONoC disadvantages include high setup-time overhead and contention for the circuit-switched paths. Gu et al. Gu *et al.* (2017) have proposed a Multiple Ring-based Optical NoC (MRONoC) design which uses ring based routing, as well as redundant paths to re-use the wavelength resources without contentions. (A related circuit-switched ONoC with a hierarchical structure based on a Clos-Benes topology has been examined in Yao and Ye (2020).) The MRONoC thus enables ultra-low cost, scalable, and contention-free communication between nodes.

Wavelength Division Multiplexing (WDM) allocates different modulated wavelengths to each communicating node to reduce the contention. Hence, in general, an ONoC system based on WDM is limited by the number of wavelengths; the wavelength reuse in MRONoC mitigates this limitation. The simulation evaluations in Gu *et al.* (2017) indicate a 133% improvement of the saturated bandwidth compared to a traditional mesh ONoC. Related statistical multiplexing strategies for ONoC channels have been investigated in Wang *et al.* (2019a), while NoC wavelength routing has been studied in Huang *et al.* (2020). Moreover, recent studies have explored the thermal characteristics of ONoCs Ye *et al.* (2020), the interference characteristics in optical wireless NoCs Dehkordi and Tralli (2019), and the SDN control for optical interconnects Chandna *et al.* (2019).

Further evolutions of integrated photonics and optical interconnects have been applied in quantum computing technologies. Wang et al. Wang *et al.* (2016a) have developed a novel chip-to-chip Quantum Photonic Interconnect (QPI) which enables the communication between quantum compute nodes. The QPI meets the demands of very high speed interconnects that are beyond the limits of single-wafer and multi-chip systems offered by state-of-the-art optical interconnects. The main challenge that is overcome in QPI is to maintain the same path-entangled states on either chip. To achieve this, a two-dimensional grating coupler on each chip transports the path-entangled states between the communicating nodes. The simulation evaluations show an acceptable stability of the QPI on quantum systems with a high degree of flexibility. As NF applications are ready to exploit quantum technologies capable of very large computations, the research efforts on interconnects enable platform designers to build heterogeneous systems that exploit the benefits of diverse hardware infrastructures.

Summary of Interconnects

In conjunction with computing architecture advancements of CPUs and I/O devices, whereby both the core numbers and the processing capacities (operations per second) have been increasing, the interconnects and interfaces that establish communication among (and within) I/O devices and CPUs play an important role in determining the overall platform performance Zhezlov *et al.* (2020). Therefore, future interconnect designs should focus not only on the individual performance of an interconnect in terms of bandwidth and latency, but also the flexibility in terms of supporting topologies (e.g., mesh, star, and bus) and reconfigurability in terms of resource reservation. 3D interconnects enable vertical stacking of the on-chip components so as to support high density processing and memory nodes. However, the high density 3D SoC components may have relatively higher failure rates as compared to 2D planar designs, due to aging and asymmetric interconnects usage.

While physical (wired) interconnects exhibit aging properties, wireless and optical interconnects appears to be a promising solution against aging. Wireless interconnects reach across longer distances and are not limited by the end-to-end metallic and silicon wires between interconnected components. However, the downsides of wireless interconnects include the design, operation, and management of wireless transceivers that include decisions on wireless link parameters, such as carrier frequencies, line-of-sight operation, and spectrum bandwidth. Similarly, optical interconnects have promising features in terms of supporting high bandwidth and short latencies using Visible Light Communications (VLC) and guided optical paths Zhang *et al.* (2019c). The design of optical interconnects is challenging as it requires extreme precision in terms of transceiver design and placements which is integrated into SoC components such that there is a guided light path or line-of-sight operation.

In addition to data path enhancements of the interconnects, future interconnect designs should address the management of interconnect resources through dedicated control plane designs. To this end, Software-Defined Network-on-Chip (SD-NoC) Ruaro *et al.* (2019) designs include a dedicated controller. The dedicated controller could be employed in future research to reconfigure the NoC fabric in terms of packet (interconnect data) routing and link resource reservations so as to achieve multi-interconnect reconfiguration that spans across multiple segments, e.g., CPU and memory. While such reconfiguration is not supported today, SD-NoC provides a general framework to enable demand based interconnect resource allocation between processing (CPUs), memory (DRAM), and I/O devices (e.g., storage) components. A related future research direction is to develop Software Defined Wireless NoC (SD-WNoC), whereby the wireless link properties are configured based on decisions made by the SDN controller to meet application requirements and available wireless interconnect resources.

3.4.3 Memory

DRAM

Understanding the latency components of DRAM memory accesses facilitates the effective design of NF applications to exploit the locality of data within DRAM system memory with reduced latency. Chang et al. Chang *et al.* (2016a) have comprehensively investigated the DRAM access latency components, which are: *i*) activation, *ii*) precharge, and *iii*) restoration. The latency variations across these components are due to manufacturing irregularities, which result in memory cells with asymmetric latency within the same DRAM chip. A shortcoming of the traditional DRAM memory access approaches is to assume that all memory cells are accessible with uniform latency. Chang et al. Chang *et al.* (2016a) have performed a quantitative study of DRAM chips to characterize the access latencies across memory cells, and then to exploit their relative latency characteristics to meet the application needs. Interestingly, the memory cells that exhibit longer latencies also exhibit spatial locality. Thus, the long-latency memory cells are in some localized memory regions that can be isolated and demarcated. Based on this insight, Chang et al. Chang *et al.* (2016a) proposed a Flexible-LatencY DRAM (FLY-DRAM) mechanism that dynamically changes the access to memory regions based on the application's latency requirements. The evaluations in Chang *et al.* (2016a) have shown nearly 20% reduction of the average latencies.

Utilizing similar techniques to reduce DRAM access latency, Hassan et al. Hassan *et al.* (2016) have proposed a DRAM access strategy based on the memory controller timing changes so as to achieve latency reductions up to 9%. Conventionally, DRAM is accessed row by row. After an initial memory access in a row, other locations in the same memory row can be accessed faster due to the precharge (applied during the initial access) than locations in other rows. The ChargeCache mechanism proposed in Hassan *et al.* (2016) tracks the previously accessed memory addresses in a table. Then, any new address locations that map to the same row are accessed with tight timing constraints, resulting in reduced access latencies.

In terms of increasing the DRAM memory density and performance, 3D package technology allows memory cells to be stacked in the third dimension and interconnected by Through Silicon Vias (TSVs). Jeddeloh et al. Jeddeloh and Keeth (2012) have proposed such a 3D stacking technology to stack heterogeneous dies close to each other with numerous interconnects between stack layers, reducing the latencies due to the short distances that signals propagate.

Bulk transfers of data blocks are common in data processing applications. However, data transfers are generally implemented through the CPU, whereby, data is first moved from the DRAM source to the CPU and then moved back to a new DRAM destination. As a result, the applications suffer from degraded performance due to *i*) limited DDR link capacity (whereby the DDR link connects the DRAM to the CPU bus), and *ii*) CPU usage for moving the data. Existing connectivity wires within a DRAM array can provide a wide internal DRAM bandwidth for data transfers. However, these data transfers are not possible out of DRAM arrays. Overcoming this limitation, Chang et al. Chang *et al.* (2016b) have proposed a Low-cost Inter-Linked SubArrays (LISA) scheme to enable fast inter-subarray data transfers across large memory ranges. LISA utilizes the existing internal wires, such as bitlines, to support data transfers across multiple subarrays with a minuscule space overhead of 0.8% of DRAM area. Experiments showed that LISA improves the energy efficiency for memory accesses and reduces the latency of workloads that involve data movements.

The performance of NF applications depends directly on the DRAM throughput and latency. The DRAM latency and throughput are degraded by data-dependent failures, whereby the data stored in the DRAM memory cells are corrupted due to the interference, especially when the DRAM has long refresh intervals. The DRAM-internal scramble and remapping of the system level address space makes it challenging to characterize the data-dependent failures based on the existing data and system address space. To address this challenge, several techniques have been proposed based on the observed pre-existing data and failures Khan *et al.* (2017, 2016). In addition to the mapping of data-dependent failures, it is also critical to dynamically map the failures with respect to the memory regions with a short time scale (high time resolution) so that applications as well as the OS and hypervisors can adapt to the failure characteristics. Hence, to enhance the performance of NF applications, the memory access reliability should be improved by minimizing the data-dependent failures.

Non-Volatile Memory (NVM)

In contrast to DRAM, the Non-Volatile Memory (NVM) retains memory values without having to refresh the memory cells. NVM is traditionally based on NAND technology. Emerging technologies that offer superior performance of NVM in terms of read and write speeds, memory density, area, and cost have been discussed by Chen et al. Chen (2016). Some of the NVM technologies that are being considered as potential solution to the growing needs of applications, such as neuromorphic computing and hardware security, include Phase Change Memory (PCM), Spin-Transfer-Torque Random Access Memory (STTRAM) Wang *et al.* (2018b), Resistive Random Access Memory (RRAM), and Ferroelectric Field Effect Transistor (FeFET). The investigative study of Chen et al. Chen (2016) indicated that a scalable selector of the memory module is a critical component in the architecture design. The NVM challenges include high-yield manufacturing, material and device engineering, as well as the memory allocation optimization considering both the NVM technology constraints and application needs.

As compared to 2D planar NAND technology, 3D Vertical-NAND (V-NAND) technology supports higher density memory cells and provides faster read/write access speeds. However, the challenges of further scaling of V-NAND include poor Word Line (WL) resistance, poor cell characteristics, as well as poor WL-WL coupling, which degrades performance. Overcoming these challenges, Kang et al. Kang *et al.* (2016a) have proposed a 3rd generation V-NAND technology that supports 256 Gb with 3 b/cell flash memory with 48 stacked WLs. In particular, Kang et al. Kang *et al.* (2016a) have implemented the V-NAND with reduced number of external components; also, an external resistor component is replaced by an on-chip resistor to provide I/O strength uniformity. A temperature sensing circuit was designed to counter the resistor temperature variation such that resistance characteristics are maintained relatively constant. Compared to the previous V-NAND implementation generation, a performance gain of 40% was observed, with the write throughput reaching 53.3 MB/s and a read throughput of 178 MB/s.

Summary of Memory

The NF performance on GPC infrastructures is closely correlated with the memory sizes and access speeds (latency). Therefore, both research and enabling technology development (Sec. 3.3.3) efforts have been focused on increasing memory cell density in a given silicon area, and improving the access (read and write) speeds of memory cells. Towards this end, 3D NAND technology improves the memory cell density through 3D vertical stacking of memory cells as compared to 2D planar linear scaling. The relatively recent NV-NAND technology defines persistent memory blocks that—in contrast to DRAM—retain data without a clock refresh (i.e., without a power supply) Choi *et al.* (2020). Without a clock refresh, the NV-NAND memory cells can be packed more densely than DRAM, resulting in large (by many folds compared to regular DRAM) persistent memory blocks. However, the main downsides of NV-NAND memory components are the slower read and write access speeds as compared to DRAM.

In addition to the physical aspects of memory, other considerations for memory performance include address translation, caching, paging, virtualization, and I/O device to memory accesses. Close examinations of accessing data from DRAM memory cells have found asymmetric latencies, whereby the data belonging to the same row of the memory cells can be accessed faster than rows that have not been accessed in recent DRAM refresh cycles. These asymmetric latencies can result in varying (non-deterministic) read and write latencies for applications with memory-intensive operations, such as media processing.

The proximity of the DRAM to the CPU determines the overall computing latency of the applications, therefore, memory blocks should be integrated in close proximity of the CPU. For instance, memory blocks can be integrated within the socket, i.e., on-package, and possibly even on-die. The tight integration of the DRAM with the CPU impacts the application performance when there is a inter-die and inter-socket memory transactions due to Non-Uniform Memory Access (NUMA) Broquedis *et al.* (2009). Illustrating the benefits of integrated DRAM, Zhang et al. Zhang *et al.* (2020) have proposed a method to integrate memory cells into compute modules (i.e., CPUs and accelerators) based on phase change memory (PRAM) modules Durai *et al.* (2020). PRAM is a memory storage cell type that can be incorporated in the DRAM, but also directly inside the accelerators and CPUs. For DRAM-less designs, the PRAM memory cells are integrated inside the accelerators and CPUs, resulting in a DRAM-less acceleration framework that achieves an improvement of 47% as compared to acceleration involving DMAs to DRAM Zhang *et al.* (2020).

An important memory-related bottleneck that needs to be addressed in future research is to improve the effective utilization of system memory when shared by multiple platform components, such as CPUs (inter- and intra-socket) and I/O devices (e.g., hardware accelerators and storage). More specifically, the interactions between CPUs, I/O devices, and system memory (DRAM) are shared by a common memory controller and system bus (DDR). The DRAM allows a single path data read and write into memory cells, whereby the memory requests (from both CPUs and I/O devices) are buffered and serialized at the memory controller when data is written to and read from the DRAM. One possible future research direction is to design a parallel request handler, which enables concurrent reads and writes with the DRAM memory cells. The concurrent reads and writes enable multiple CPUs and I/O devices to simultaneously interact with the memory cells, improving the overall throughput of the memory access. A key challenge to overcome with this concurrent memory access approach is to ensure synchronization when the same memory location is concurrently accessed by multiple components (i.e., memory accesses collide) and to avoid data corruption. Colliding memory accesses need to be arbitrated by serializing the memory accesses in a synchronization module. On the other hand, non-colliding concurrent memory accesses by multiple components to different memory locations, i.e., concurrent reads and writes to different DRAM locations, can improve the memory utilization.

3.4.4 Accelerators

Data Processing Accelerators

Specialized hardware accelerators can significantly improve the performance and power efficiency of NFs. Ozdal et al. Ozdal *et al.* (2016) have designed and evaluated a hardware accelerator for graph analytics, which is needed, e.g., for network traffic routing, source tracing for security, distributed resource allocation and monitoring, peer-to-peer traffic monitoring, and grid computing. The proposed architecture processes multiple graph vertices (on the order of tens of vertices) and edges (on the order of hundreds of edges) in parallel, whereby partial computing states are maintained for vertices and edges that depend on time-consuming computations. The computations for the different vertices and edges are dynamically distributed to computation execution states depending on the computational demands for the different vertices and edges. Moreover, the parallel computations for the different vertices and edges are synchronized through a specialized synchronization unit for graph analytics. Evaluations indicate that the developed graph analytics accelerator achieves three times higher graph analytics performance than a 24 core CPU while requiring less than one tenth of the energy.

While the accelerator of Ozdal et al. Ozdal *et al.* (2016) is specifically designed for graph analytics, a generalized reconfigurable accelerator FPGA logic referred to as *Configurable Cloud* for arbitrary packet NFs as well as data center applications has been proposed by Caulfield et al. Caulfield *et al.* (2020); Caulfield *et al.* (2018). The Configurable Cloud structure inserts a layer of reconfigurable logic between the network switches and the servers. This reconfigurable logic layer can flexibly transform data flows at line rate. For instance, the FPGA layer can encrypt and decrypt 40 Gb/s data packet flows without loading the CPU. The FPGA layer can also execute packet operations for Software-Defined Networking (SDN). Aside from these packet networking related acceleration functions, the FPGA layer can accelerate some of the data center processing tasks that are ordinarily executed by the data center CPUs, such as higher-order network management functions.

Gray Gray (2016) has proposed a parallel processor and accelerator array framework called Phalanx. In Phalanx, groups of processors and accelerators form shared memory clusters. Phalanx is an efficient FPGA implementation of the RISC-V IS (an open source instruction set RISC processor design), achieving high throughput and I/O bandwidth. The clusters are interconnected with a very-high-speed Hoplite NoC Kapre and Gray (2015). The Hoplite NoC is a 2D switching fabric that is reconfigurable (for routing), torus (circular ring), and directional. Compared to a traditional FPGA routing fabric, Hoplite provides a better area $\times$ delay product. The Phalanx FPGA processor design was successfully implemented to boot with 400 cores, run a display monitor, perform billions of I/O operations, as well as run AES encryption. Platforms with large numbers of parallel processors and high interconnect bandwidth can perform both many independent tasks as well as handle large amounts of inter-thread communications. Such architectures are uniquely positioned to run NF applications that operate on a flow basis. Thereby, one or more cores can be dedicated to process a single packet flow, and scale up the resources based on dynamic flow requirements.

MapReduce performs two operations on a data set: *i*) map one form of data to another, and *ii*) reduce the data size (e.g., by hashing) and store the reduced data as key-value pairs (tuples) in a database. MapReduce typically operates on large data sets and employs a large number of distributed computing nodes. Networking applications use MapReduce for various data analysis functions, such as traffic (packet and flow statistics) analysis Lee *et al.* (2010b) and network data analytics (e.g., related to users, nodes, as well as cost and efficiency of end-to-end paths) Song *et al.* (2016), especially in the centralized decision making of SDN controllers. Therefore, hardware acceleration of MapReduce in a platform further enhances the performance of NF applications that perform network traffic and data analytics. Towards this goal, Neshatpour et al. Neshatpour *et al.* (2018) have proposed the implementation of big data analytics applications in a heterogeneous CPU+FPGA accelerator architecture. Neshatpour et al. have developed the full implementation of the HW+SW mappers on the Zynq FPGA platform. The performance characterization with respect to core processing requirements for small cores (e.g., Intel® Atom) and big cores (e.g., Intel® i7) interacting with hardware accelerators that implement MapReduce has been quantified. In case of small cores, both SW and HW accelerations are required to achieve high benchmarking scores; while in case of big cores, HW acceleration alone yields improved energy efficiency.

Deep-Learning Accelerator

Neural Networks (NNs) have been widely used in applications that need to learn inference from existing data, and predict an event of interest based on the learned inference. NF applications that use NNs for their evaluations include traffic analysis NFs, such as classification, forecasting, anomaly detection, and Quality-of-Service (QoS) estimation Lotfollahi *et al.* (2020). Generally, NN computations require large memory and very high computing power to obtain results in a short time-frame. To this end, Zhang et al. Zhang *et al.* (2016) have proposed a novel accelerator, Cambricon-X, which exploits the sparsity and irregularity of NN models to achieve increased compute efficiency. Cambricon-X implements a Buffer Controller (BC) module to manage the data in terms of indexing and assembling to feed into Processing Elements (PE) that compute the NN functions. With sparse connections, Cambricon-X achieves 544 Giga Operations Per second (GOP/s), which is 7.2$\times$ the throughput of the state-of-the-art DianNao implementation Chen *et al.* (2014), while Cambricon-X is 6.4$\times$ more energy efficient.

Deep learning and NN based applications require large numbers of parallel compute nodes to run their inference and prediction models. To meet this demand, massive numbers of high performance CPUs, custom accelerator GPUs and FPGAs, as well as dedicated accelerators, such as Cambricon-X Zhang *et al.* (2016) have been utilized by the software models. However, a critical component that limits the scaling of computing is memory in terms of both the number of I/O transactions and the capacity. The I/O bound transactions that originate collectively from the large number of threads running on numerous cores in CPUs, GPUs, and FPGAs use a Message Passing Interface (MPI) for inter-thread communications. In some cases, such as large-scale combinatorial optimization applications, each thread needs to communicate with every other thread, resulting in a mesh connection that overloads the MPI infrastructure. Mahdi et al. Bojnordi and Ipek (2016) have proposed a dedicated hardware accelerator to overcome the memory I/O and capacity bottlenecks that arise with the scaling of computing resources. In particular, a hardware accelerator is designed based on the Resistive Random Access Memory (RRAM) technology Akinaga and Shima (2010) to support the compute and memory requirements of large-scale combinatorial optimizations and deep learning applications based on Boltzmann machines. The RRAM based accelerator is capable of fine-grained parallel in-memory computations that can achieve 57-fold compute performance improvements and 25-fold energy savings as compared to traditional multi-core systems. In comparison to Processing In-Memory (PIM) systems (see Sec. **??**), the RRAM based accelerator shows 6.9-fold and 5.2-fold performance improvement and energy savings, respectively.

Traditional CISC based IS architecture CPUs are not optimized to run compute-intensive workloads with massive data parallelism, e.g., deep learning and data analytics. Therefore, to supplement the specialized and dedicated computing infrastructures in the parallel processing of large data, hardware offloading based on FPGA and GPU can employed. However, hardware offloading generally comes with the following challenges: *i*) application specific for a given configuration, *ii*) memory offloading, and *iii*) reconfiguration delay. The present reconfigurable hardware components, such as FPGA and GPU, require a standardized programming model, synthesis, and configuration of FPGA and GPU; this hardware reconfiguration does not support short (near run-time) time scales. As application requirements change much faster than the typical FPGA and GPU configuration cycles, a CPU based acceleration could offer faster adaption to changing needs. The Configurable Spatial Accelerator (CSA) (CSA) **?** architecture (see Fig. 3.33) was proposed to accelerate large parallel computations. The CSA consists of high density floating point and integer arithmetic logic units that are interconnected by a switching fabric. The main CSA advantages are: *i*) the support of standard compilers, such as C and C++, which are commonly used for CPUs, and *ii*) short reconfiguration times on the order of nanoseconds to a few microseconds Zhang (2019). The CSA can directly augment existing CPUs, whereby the CSA can use the CPU memory without having to maintain a separate local memory for computing as compared to external accelerators. In particular, the CSA can be adopted as an integrated accelerator in the form of a CSA die next to the CPU die in the same socket and package; CSA memory read/write requests will be forwarded through inter-die interconnects and (intra-CPU die) 2D mesh to the CPU memory controller. Figure 3.33 illustrates the architectural CSA components consisting of a switching network, Integer Processing Elements (Int PEs), and Fused Multiply-Add (FMA) PEs. A large number of Int PEs and FMA PEs are interconnected via switches to form a hardware component that can support compute-intensive workloads. The CSA adapts quickly to varying traffic demands at fine-grained time-scales so that NF applications can adapt to changing requirement through hardware acceleration reconfiguration.

In terms of stress on the interconnects, deep learning and inference software models implement large numbers of inter-communicating threads, resulting in significant interconnect usage. Typically, the thread communication is enabled by a Message Passing Interface (MPI) provided by the OS kernel. However, as the numbers of threads and compute nodes increase, the OS management of the MPI becomes challenging. Dos et al. Dosanjh *et al.* (2019) have proposed a hardware acceleration framework for the MPI to assist the CPU with the inter-thread communications. The MPI hardware acceleration includes a fuzzy matching of source and destination to enable the communication links with a probable partial truth rather than exact (deterministic) connections at all times. Fuzzy based hardware acceleration for link creation reduces the overhead on the interconnect with reduced usage of communication links for both control and actual data message exchanges between threads. Evaluations of the hardware-accelerated MPI have shown 1.13 GB in memory (DRAM) savings, and a matching time improvement of 96% as compared to a software-based MPI library.

GPU-RDMA Accelerator

Remote Direct Memory Access (RDMA) enables system memory access (i.e., DRAM) on a remote platform, usually either via the PCIe-based NTB (see Sec. 3.3.6) or Ethernet-based network connections. The Infiniband protocol embedded in the NIC defines the RDMA procedures for transferring data between the platforms. Typically, the CPU interacts with the NIC to establish end-to-end RDMA connections, whereby the data transfers are transparent to applications. That is, the external memory is exposed as a self-memory of the CPU such that if a GPU wants to access the remote system memory, the GPU requests the data transfer from the CPU. This process is inefficient as the CPU is involved in the data transfer for the GPU. In an effort to reduce the burden on the CPU, Daoud et al. Daoud *et al.* (2016) have proposed a GPU-side library, *GPUrdma*, that enables the GPU to directly interact with the NIC to perform RDMA across the network, as shown in Fig. 3.34(c). The *GPUrdma* implements a Global address-space Programming Interface (GPI). The *GPUrdma* has been evaluated for ping-pong and multi–matrix-vector product applications in Daoud *et al.* (2016). The evaluations showed 4.5-fold faster processing as compared to the CPU managing the remote data for the GPU.

Crypto Accelerator

Cryptography functions, such as encryption and decryption, are computationally intensive processes that require large amounts of ALU and branching operations on the CPU. Therefore, cryptography functions cause high CPU utilizations, especially in platforms with relatively low computing power. In mobile network infrastructures, such as in-vehicle networks, the computing power is relatively lower compared to traditional servers. In-vehicle networks require secure on-board data transactions between the sensors and computing notes, whereby this communications is critical due to vehicle safety concerns. While cryptography is commonly adopted in platforms with low computing resources, hardware cryptography acceleration is essential. In-vehicle networks also require near-real-time responses to sensor data, which further motivates hardware-based acceleration of cryptography functions to meet the throughput and latency need of the overall system. An in-vehicle network design proposed by Baldanzi et al. Baldanzi *et al.* (2019) includes a hardware acceleration for the AES-128/256 data encryption and decryption. The AES accelerator was implemented on an FPGA and on 45 nm CMOS technology. The latency of both implementations was around 15 clock cycles, whereby the throughput of the FPGA was 1.69 Gbps and the CMOS achieved 5.35 Gbps.

Similarly, Intrusion Detection Systems (IDSs) perform security operations by monitoring and matching the sensor data. In case of NF applications, this is applicable to network traffic monitoring. Denial-of-Service attacks target a system (network node) with numerous requests so that genuine requests are denied due to resource unavailability. A CPU-based software IDS implementation involves *i*) monitoring of traffic, and *ii*) matching the traffic signature for an anomaly, which is computationally expensive. Aldwairi et.al Aldwairi *et al.* (2005) have proposed a configurable network processor with string matching accelerators for IDS implementation. In particular, the hardware accelerator architecture includes multiple string-matching accelerators on the network processor to match different flows. Simulation results showed an overall performance up to 14 Gbps at run-time wire speed while supporting reconfiguration.

Although encryption and decryption hardware acceleration improve the overall CPU utilization, the performance of hardware offload is significant only for large data (packet) sizes. For small data sizes, the offload cost can outweigh the gains of hardware accelerations. To address this trade-off, Zhong et al. Zhong *et al.* (2019)

have proposed a Self-Adaptive Encryption and Decryption Architecture (SAED) to balance the asymmetric hardware offload cost by scheduling the crypto computing requests between CPU and Intel® Quick Assist Technology® (a hardware accelerator, see Sec. 3.3.5). SAED steers the traffic to processing either by the CPU or the hardware accelerator based on the packet size. SAED improves the overall system performance for security application in terms of both throughput and energy savings, achieving around 80% improvement compared to CPU processing alone, and around 10% improvement compared to hardware accelerator processing alone.

In-Memory Accelerator

An in-memory accelerator utilizes DRAM memory cells to implement logical and arithmetic functions, thus entirely avoiding data movements between the accelerator device and DRAM (i.e., system memory). The CPU can utilize the high bandwidth DDR to communicate with the acceleration function residing at DRAM memory regions. While it is challenging to design complex arithmetic functions inside the DRAM, simple logic functions, such as bitwise AND and OR operations, can be implemented with minimal changes to existing DRAM designs. Seshadri et al. Seshadri *et al.* (2017) have proposed a mechanism to perform bulk bitwise operations on a commodity DRAM using a sense amplifier circuit which is already present in DRAM chips. In addition, inverters present in the sense amplifiers can be extended to perform bitwise NOT operations. These modifications to DRAM require only minor changes (1% of chip area) to the existing designs. The simulation evaluations in Seshadri *et al.* (2017) showed that performance characteristics are stable, even with these process variations. In-memory acceleration for bulk bitwise operations showed 32-fold performance improvements and 35-fold energy consumption savings. High Bandwidth Memory (HBM) with 3D stacking of DRAM memory cells has shown nearly ten-fold improvements. Bulk bitwise operations are necessary for NF applications that rely heavily on database functions (search and lookup). Thus, in-memory acceleration provides a significant acceleration potential to meet the latency and energy savings demands of NFs relying on database functions.

Generally, the DRAM capacity is limited and therefore the in-memory acceleration capabilities in terms of supporting large datasets for data-intensive applications fall short in DRAM. Non-Volatile Memory (NVM) is seen as a potential extension of existing DRAM memory to support larger system memory. It is therefore worthwhile to investigate in-memory acceleration in NVM memory cells. Li et al. Li *et al.* (2019b) have presented an overview of NVM based in-memory based acceleration techniques. NVM can support wider function acceleration, such as logic, arithmetic, associative, vector. and matrix-vector multiplications, as compared DRAM due to the increased NVM memory and space availability.

For instance, Li et al. Li *et al.* (2016b) have proposed the Pinatubo processing-in-memory architecture for bulk bitwise operations in NVM technologies. Pinatubo reuses the existing circuitry to implement computations in memory as opposed to new embedded logic circuits. In addition to bitwise logic operations between one or two memory rows, Pinatubo also supports one-step multi-row operations which can be used for vector processing. As a result, Pinatubo achieves 1.12× overall latency reductions and 1.11× energy savings compared to a conventional CPU for data intensive graph processing and database applications.

Summary of Accelerators

Custom accelerators, such as FPGA and GPU, provide high degrees of flexibility in terms of programming the function of the accelerators. In contrast, dedicated accelerators implement specific sets of functions on the hardware which limits the flexibility. NFs have diverse sets of function acceleration requirements, ranging for instance from security algorithm implementation to packet header lookup, which results in heterogeneous characteristics in terms of supporting parallel executions and compute-intensive tasks. Regardless of all the options available for hardware acceleration, the overall accelerator offloading efficiency depends on memory transactions and tasks scheduling. One possible future research direction towards increasing accelerator utilization is to compile the application with “accelerator awareness” such that a given task can be readily decomposed into subtasks that are specific to FPGAs, GPUs, and dedicated accelerators. Accelerator-specific subtasks can be independently scheduled to run on the hardware accelerators to coordinate with the main task (application) running on the CPU. Future research should develop common task scheduling strategies between hardware accelerators and CPU, which could improve the utilization of hardware accelerators but also enable applications to reap the individual benefits of each accelerator component.

Other open research challenges in the design of accelerators include supporting software definable interconnects and logic blocks Zhang *et al.* (2019b) with run-time reconfiguration and dynamic resource allocation. In contrast to an FPGA, a GPU can switch between threads at run-time and thus can be instantaneously reconfigured to run different tasks. However, the GPU requires a memory context change for every thread scheduling change. To overcome this GPU memory context change requirement, High Bandwidth Memory (HBM) modules integrated with a GPU can eliminate the memory transactions overhead during the GPU processing by coping the entire data required for computing to the GPU’s local memory. HBM also enables the GPUs to be used as a remote accelerator over the network Stunkel *et al.* (2020); Hamidouche and LeBeane (2020); Sharkawi and Chochia (2020). However, one limitation of remote accelerator computing is that results are locally evaluated (e.g., analytics) on a remote node in a non-encrypted format. The non-encrypted format could raise security and privacy concerns as most GPU applications involve database and analytics applications that share the data with remote execution nodes.

Dedicated accelerators provide an efficient way of accelerating NFs in terms of energy consumption and hardware accelerator processing latency. However, the downsides of dedicated accelerators include: *i*) the common task offloading overheads from CPU, i.e., copying data to accelerator internal memory and waiting for results through polling or interrupts, and *ii*) the management of the accelerators, i.e., sharing across multiple CPUs, threads, and processes. To eliminate these overheads, in-memory accelerators have been proposed to include (embed) the logic operations within the DRAM memory internals such that a write action to specific memory cells will result in compute operations on the input data and results are available to be read instantaneously. While this approach seems to be efficient for fulfilling the acceleration requirements of an application, the design of in-memory accelerators that are capable of arithmetic (integer and floating point) operations is highly challenging Reis *et al.* (2020); Sebastian *et al.* (2020). Arithmetic Logic Units (ALUs) would require large silicon areas within the DRAM and to include ALUs at microscopic scale of memory cells is spatially challenging. Another important future direction is to extend the in-memory acceleration to 3D stacked memory cells Zhu *et al.* (2013) supporting HBM-in-memory acceleration.

3.4.5 Infrastructure

SmartNIC

SmartNICs enable programmability of the NICs to assist NFs by enabling hardware acceleration in the packet processing pipeline. Ni et al. Ni *et al.* (2019) have outlined performance benefits of SmartNIC acceleration for NF applications. However, the accessing and sharing of hardware functions on a SmartNIC by multiple applications is still challenging due to software overheads (e.g., resource slicing and virtualization). Yan et al. Yan *et al.* (2019) have proposed a UniSec method that allows software applications to uniformly access the hardware-accelerated functions on SmartNICs. More specifically, UniSec provides an unified Application Programming Interface (API) designed to access the high-speed security functions implemented on the SmartNIC hardware. The security functions required by NF applications include for instance Packet Filtering Firewall (PFW) and Intrusion Detection System (IDS). The implementation of UniSec is classified into control (e.g., rule and log management) and data (i.e., packet processing) modules. Data modules parse packets and match the header to filter packets. UniSec considers stateless, stateful, and payload based security detection on the packet flows on a hardware Security Function (hSF). A virtual Security Function (vSF) is generated through Security Function (SF) libraries, as illustrated in Fig. 3.35. UniSec reduces the overall code for hardware re-use by 65%, and improves the code execution (CPU utilization) by 76% as compared to a software-only implementation.

Traditionally hardware acceleration of software components is enabled through custom accelerators, such as GPUs and FPGAs. However, in large-scale accelerator deployments, the CPU and NIC become the bottlenecks due to increased I/O bound transactions. To reduce the load on the CPU, SmartNICs can be utilized to process the network requests to perform acceleration on the hardware (e.g., GPU). Tor et al. Tork *et al.* (2020) have proposed a SmartNIC architecture, Lynx, that processes the service requests (instead of the CPU), and delivers the service requests to accelerators (e.g., GPU), thereby reducing the I/O load on the CPU. Figure 3.36 illustrates the Lynx architecture in comparison to the traditional approach in which the CPU processes the accelerator service requests. Lynx evaluations conducted by Tor et al. Tork *et al.* (2020) where GPUs communicate with an external (remote) database through SmartNICs show 25% system throughput increases for a neural network workload as compared to the traditional CPU service requests to the GPU accelerator.

Summary of Infrastructures

Infrastructures consist of NICs, physical links, and network components to enable platforms to communicate with external compute nodes, such as peer platforms, the cloud, and edge servers. SmartNICs enhance existing NICs with hardware accelerators (e.g., FPGAs and cryptography accelerators) and general purpose processing components (e.g., RISC processors) so that functional tasks related to packet processing that typically run on CPUs can be offloaded to SmartNICs. For instance, Li et al. Li *et al.* (2020) have implemented a programmable Intrusion Detection System (IDS) and packet firewall based on an FPGA embedded in the SmartNIC. Belocchi et al. Belocchi *et al.* (2020) have discussed the protocol accelerations on programmable SmartNICs.

Although state-of-the-art SmartNIC solutions focus on improving application acceleration capabilities, the processing on the SmartNICs is still coordinated by the CPU. Therefore, a interesting future research directions is to improve the independent executions on the SmartNICs with minimized interactions with the CPU. Independent executions would allow the SmartNICs to perform execution and respond to requests from remote nodes without CPU involvement.

3.4.6 Summary and Discussion of Research Studies

Research studies on infrastructures and platforms provide perspectives on how system software and NF applications should adapt to the changing hardware capabilities. Towards this end, it is important to critically understand both the advantages and disadvantages of the recent advances of hardware capabilities so as to avoid the pitfalls which may negatively impact the overall NF application performance.

In terms of computing, there is a clear distinction between CISC and RISC architectures: CISC processors are more suitable for large-scale computing and capable of supporting high-performing platforms, such as servers. In contrast, RISC processors are mainly seen as an auxiliary computing option to CISC, such as for accelerator components. Therefore, NF applications should decouple their computing requirements into CISC-based and RISC-based computing requirements such that the respective strengths of CISC- and RISC-based computing can be harnessed in heterogeneous platforms.

As the number of cores increases, the management of threads that run on different cores becomes more complex. If not optimally managed, the overheads of operating multiple cores may subdue the overall benefit achieved from multiple cores. In addition, extensive inter-thread communication stresses the core-to-core interconnects, resulting in communication delay, which in turn decreases the application performance. Therefore, applications that run on multiple cores should consider thread management and inter-thread communication to achieve the best performance.

The power control of platform components is essential to save power. However, severe power control strategies that operate on long time-scales can degrade the performance and induce uncorrectable errors inside the system. Therefore, power and frequency control strategies should carefully consider their time-scale of operation so as not to impact the response times for the NF applications. A long-time-scale power control would take numerous clock cycles to recover from low performance states to high performance states. Conversely, a short-time-scale power control is highly reactive to the changing requirements of NF applications (e.g., changing traffic arrivals); however, short time-scales result in more overheads to evaluate requirements and control states.

While several existing strategies can increase both on-chip and chip-to-chip interconnect capabilities, future designs should reduce the cost and implementation complexity. The Network-on-Chip (NoC) provides a common platform, but an NoC increases the latency as the number of components increases. In contrast, 2D mesh interconnects provide more disjoint links for the communications between the components. Millimeter wireless and optical interconnects provide high-bandwidth, long-range, and low-latency interconnects, but the design of embedded wireless and optical transceivers on-chip increases the chip size and area. A 3D NoC provides more space due to the vertical stacking of compute components, but power delivery and heat dissipation become challenging, which can reduce the chip lifespan.

3.5 Open Challenges and Future Research Directions

Building on the survey of the existing hardware-accelerated platforms and infrastructures for processing softwarized NFs, this section summarizes the main remaining open challenges and outlines future research directions to address these challenges. Optimizing hardware-accelerated platforms and infrastructures, while meeting and exceeding the requirements of flexibility, scalability, security, cost, power consumption, and performance, of NF applications poses enormous challenges. We believe that the following future directions should be pursued with priority to address the most immediate challenges of hardware-accelerated platforms and infrastructures for NF applications. The future designs and methods for hardware-accelerated platforms and infrastructures can ultimately improve the performance and efficiency of softwarized NF applications.

We first outline overarching grand challenges for the field of hardware-accelerated platforms and infrastructures, followed by specific open technical challenges and future directions for the main survey categories of CPUs, memory, interconnects, accelerators, and infrastructure. We close this section with an outlook to accelerations outside of the immediate realm of platforms and infrastructures; specifically, we briefly note the related fields of accelerations for OSs and hypervisors as well as orchestration and protocols.

3.5.1 Overarching Grand Challenges

Complexity

As the demands for computing increase, two approaches can be applied to increase the computing resources: *i*) horizontal scaling and *ii*) vertical scaling. In horizontal scaling, the amount of computing resources are increased, such as increasing the number of cores (in case of multi processors) and number of platforms. The main challenges associated with horizontal scaling are the application management that runs on multiple cores to maintain data coherency (i.e., cache and memory locality), synchronization issues, as well as the scheduling of distributed systems in large scale infrastructures. In vertical scaling, the platform capacities are improved, e.g., with higher core computing capabilities, larger memory, and acceleration components. The main challenges of vertical scaling are the power management of the higher computing capabilities, the management of large memories while preserving locality (see Sec. 3.4.3), and accelerator scheduling. In summary, when improving the platform and infrastructure capabilities, the complexity of the overall system should still be reasonable.

Cost

The cost of platforms and infrastructures should be significantly lowered in order to facilitate the NF softwarization. For instance, the 3D stacking of memory within compute processors incurs significant fabrication costs, as well as reduced chip reliability due to mechanical and electrical issues due to the compact packaging of hardware components Guo *et al.* (2018). Hardware upgrades generally replace existing hardware partially or completely with new hardware, incurring significant cost. Large compute infrastructures also demand high heat sinking capacities with exhausts and air circulation, increasing the operational cost. Therefore, future research and design needs to carefully examine and balance the higher performance-higher cost trade-offs.

Flexibility

Hardware flexibility is essential to support the diverse requirements of NF applications. That is, an accelerator should support a wide range of requirements and support a function that is common to multiple NF applications such that a single hardware accelerator can be reused for different applications, leading to increased utilization and reduced overall infrastructure cost. A hybrid interconnect technology that can flexibly support different technologies, such as optical and quantum communications, could allow application designers to abstract and exploit the faster inter-core communications for meeting the NF application deadlines. For instance, a common protocol and interface definition for interconnect resource allocation in a reconfigurable hardware would help application designers to use Application-Specific Interfaces (APIs) to interact with the interconnect resource manager for allocations, modification, and deallocations.

Power and Performance

Zhang et al. Zhang *et al.* (2019a) have extensively evaluated the performance of software implementations of switches. Their evaluations show that performance is highly variable with applications (e.g., firewall, DoS), packet processing libraries (e.g., DPDK), and OS strategies (e.g., kernel bypass). As a result, a reasonable latency attribution to the actions of the switching function in the software cannot be reasonably made for a collection of multiple scenarios (but is individually possible). While there exist several function parameter tuning approaches, such as increasing the descriptor ring (circular queue) sizes, disabling flow control, and eliminating MAC learning in the software switches, hardware acceleration provides better confidence in terms of performance limitations of any given function.

Software implementations also consume more power as compared to dedicated hardware implementations due to the execution on CPUs. Therefore, software implementations of NF applications are in general more power expensive than hardware implementations. Nevertheless, it is challenging to maintain the uniformity in switching and forwarding latency of a software switch (an example of NF application). Hence a pitfall to avoid is to assume uniform switching and forwarding latencies of software switches when serving NF applications with strict deadline requirements.

On the other hand, hardware implementations generally do not perform well if offloading is mismanaged, e.g., through inefficient memory allocation. Also, it is generally inefficient to offload relatively small tasks whose offloading incurs more overhead than can be gained from the offloading.

3.5.2 CPU and Computing Architecture

Hardware based Polling

As the number of accelerator devices increases on a platform, individually managing hardware resources becomes cumbersome to the OS as well as the application. In particular, the software overheads in the OS kernel and hypervisor (except for pass-through) increase with the task offloading to increasing numbers of accelerator devices;

moreover, increasing amounts of CPU resources are needed for hardware resource and power management. One of the software overheads is attributed to polling based task offloading. With polling based task offloading, the CPU continuously monitors the accelerator status for task completion, which wastes many CPU cycles for idle polling (i.e., polling that fetches a no task completion result). Also, as the number of applications that interact with the accelerator devices increases, there is an enormous burden on the CPU. A solution to this problem would be to embed a hardware-based polling logic in the CPU such that the ALU and opcode pipelines are not used by the hardware polling logic. Although this hardware polling logic solution would achieve short latencies due to the presence of the hardware logic inside CPU, a significant amount of interconnect fabric would still be used up for the polling.

CPU based Hardware Acceleration Manager

The current state-of-the-art management techniques for accelerating an NF application through utilization of a hardware resource (component) involve the following steps: the OS has to be aware of the hardware component, a driver has to initialize and manage the hardware component, and the application has to interact with user-space libraries to schedule tasks on the hardware component. A major downside to this management approach is that there are multiple levels of abstraction and hardware management. An optimized way is to enable applications to directly call an instruction set (IS) to forward requests to the hardware accelerator component. Although, this optimization exists today (e.g., `MOVDIR` and `ENQCMD` ISs from Intel Intel Corporation (2020d)), the hardware management is still managed by the OS, whereby only the task submission is performed directly through the CPU IS. A future enhancement to the task submission would be to allow the CPU to completely manage the hardware resources. That is, an acceleration manager component in the CPU could keep track of the hardware resources, their allocations to NF applications, and the task management on behalf of the OS and hypervisors. Such a CPU based management approach would also help the CPU to switch between execution on the hardware accelerator or on the CPU according to a comprehensive evaluation to optimize NF application performance.

Thermal Balancing

In the present computing architectures, the spatial characteristics of the chip and package (e.g., the socket) are not considered in the heterogeneous scheduling (see Sec. 3.4.1) of processes and tasks. As a result, on a platform with an on-chip integrated accelerator (i.e., accelerator connected to CPU switching fabric, e.g., 2D mesh), a blind scheduling of tasks to accelerators can lead to a thermal imbalance on the chip. For instance, if the core always selects an accelerator in its closest proximity, then the core and accelerator will be susceptible to the same thermal characteristics. A potential future solution is to consider the spatial characteristics of the usage of CPUs and accelerators in the heterogeneous task scheduling. A pitfall is to avoid the selection of accelerators and CPUs that create lot of cross-fabric traffic. Therefore, the spatial balancing of the on-chip thermal characteristics has to be traded off with the fabric utilization while performing CPU and accelerator task scheduling.

API based Resource Control

Although there exists frequency control technologies and strategies (see Sec. 3.3.1), the resource allocation is typically determined by the OS. For instance, the DVFS technique to control the voltage and CPU clock is in response to chip characteristics (e.g., temperature) and application load. However, there are no common software Application Programming Interfaces (APIs) for user space applications to request resources based on changing requirements. A future API design could create a framework for NF applications to meet strict deadlines. A pitfall to avoid in API based resource control is to ensure isolation between applications. This application isolation can be achieved through fixed maximum limits on allocated resources and categorizing applications with respect to different (priority) classes of services along with a best effort (lowest priority) service class.

3.5.3 Interconnects

Cross-Chip Interconnect Reconfiguration

In accelerator offload designs, the path between CPU and accelerator device for task offloading and data transfer may cross several interconnects (see Sec. 3.3.2). The multiple segments of interconnects may involve both on-chip switching fabrics and chip-to-chip interconnects of variable capacities. For instance, an accelerator I/O interacting with a CPU can encompass the following interconnects: *i*) accelerator on-chip switching fabric, *ii*) core-to-core interconnect (e.g., 2D mesh), *iii*) CPU to memory interconnect (i.e., DDR). In addition to interconnects, the processing nodes, such as CPU and memory controllers, are also shared resources that are shared by other system components and applications. A critical open challenge is to ensure a dedicated interconnect service to NF applications across various interconnects and processing elements. One of the potential future solutions is to centrally control the resources in software-defined manner. Such a software-defined central interconnect control requires monitoring and allocation of interconnect resources and assumes that interconnect technologies support reconfigurability. However, a pitfall to avoid would be a large control overhead for managing the interconnect resources.

3.5.4 Memory

Heterogeneous Non-Uniform Memory Access (NUMA)

Sieber et al. **?** have presented several strategies applied to cache, memory access, core utilization, and I/O devices to overcome the hardware level limitations of the NFV performance. The main challenge that has been stressed is to ensure the performance guarantees of a softwarized NF. Specific to NUMA node strategies, there can be I/O devices in addition to memory components that can be connected to CPU nodes. The cross node traffic accessed by I/O devices can significantly impact the overall performance. That is, a NIC connected to CPU1 (socket 1), trying to interact with the cores of CPU2 (socket 2) would have lower effective throughput as compared to a NIC that is connected to CPU1 and communicates with the CPU1 cores. Therefore, not only the I/O device interrupts need to be balanced among the available cores to distribute the processing load across available cores, but balancing has to be specific to the CPU that the NIC has been connected into. An important future research direction is to design hardware enhancements that can reduce the impact of NUMA, whereby a common fabric extends to connect with CPUs, memory, and I/O devices.

In-Memory Networking

Processing In-Memory (PIM) has enabled applications to compute simple operations, such as bit-wise computations (e.g., AND, OR, XOR), inside the memory component, without moving data between CPU and memory. However, current PIM technologies are limited by their computing capabilities as there is no support for ALUs and floating point processors in-memory. While there are hardware limitations in terms of size (area) and latency of memory access if a memory module is implemented with complex logic circuits, many applications (see Sec. 3.4.4) are currently considering to offload bit-wise operations, which are a small portion of complex functions, such as data analytics. On the other hand, most NF packet processing applications, e.g., header lookup, table match to find port id, and hash lookup, are bit-wise dominant operations; nevertheless, packet processing application are not generally considered as in-memory applications as they involve I/O dominant operations. That is, in a typical packet processing application scenario, packets are in transit from one port to another port in a physical network switch, which inhibits in-memory acceleration since the data is in transit. Potential applications for packet based in-memory computing could be virtual switches and routers. In virtual switches and routers, the packets are moved from one memory location to another memory location which is an in-memory operation. Hence, exploring in-memory acceleration for virtual switches and routers is an interesting future research direction.

3.5.5 Accelerators

Common Accelerator Context

As the demands for computing and acceleration grow, platforms are expected to include more accelerators. For example, a CPU socket (see Sec. 3.3.2) can have four integrated acceleration devices (of the same type), balancing the design such that an accelerator can be embedded on each socket quadrant, interfacing directly with CPU interconnect fabric. On a typical four-socket system, there are then a total of 16 acceleration devices of the same type (e.g., QAT®). In terms of the PCIe devices, a physical device function can be further split into many virtual functions of similar types. All of these developments attribute to a large number of accelerator devices of the same type on a given platform. A future accelerator resource allocation management approach with a low impact on existing implementation methods could share the accelerator context among all other devices once an application has registered with one of these accelerator functions (whereby an accelerator function corresponds to either a physical or virtual accelerator device). A shared context would allow the application to submit an offload request to any accelerator function. A pitfall to avoid is to consider the security concerns of the application and accelerator due to the shared context either through hardware enhancements, such as the Trusted Execution Environment (TEE) or Software Guard eXtensions (SGX) Jain *et al.* (2016b).

As hardware accelerators are more widely adopted for accelerating softwarized NFs, the platforms will likely contain many heterogeneous accelerator devices, e.g., GPU, FPGA, and QAT® (see Secs. 3.3.4 and 3.3.5). In large deployments of platforms and infrastructures, such as data centers, the workload across multiple platforms often fluctuates. Provided there is sufficient bandwidth and low latency connectivity between platforms with high and low accelerator resource utilization, there can be inter-platform accelerator sharing through a network link. This provides a framework for multi-platform accelerator sharing, whereby the accelerators are seen as a pool of resources with latency cost associated with each accelerator in the pool. A software defined acceleration resource allocation and management can facilitate the balancing of loads between higher and lower utilization platforms while still meeting application demands.

Context based Resource Allocation

In terms of the software execution flow, an NF application which intends to communicate with an accelerator device for task offloading is required to register with the accelerator, upon which a context is given to the application. A "context" is a data-structure that consists of an acknowledgment to the acceleration request with accelerator information, such as accelerator specific parameters, policies, and supported capabilities. An open challenge to overcome in future accelerator design and usage is to allocate the system resources based on context. Device virtualization techniques, such as Scalable I/O virtualization (SIOV) **?**, outline the principles for I/O device resource allocation, but do not extend such capabilities to system-wide resource allocation. When an application registers with the accelerator device, the accelerator device can make further requests to system components, such as the CPU (for cache allocation) and memory controller (for memory I/O allocation), on behalf of application. To note, applications are typically not provided with information about the accelerators and system-wide utilization for security concerns, and therefore cannot directly make reservation requests based on utilization factors. Therefore, the accelerator device (i.e., driver) has to anchor the reservation requests made by the application, to coordinate with the accelerator device, CPU, and other components (such as interconnects and memory) to confirm back to the application with the accepted class of service levels. The NF application makes request to the accelerator during registration and the accepted class of service will be provided in the “context” message returned to the NF application.

3.5.6 Infrastructure

SmartNIC Offline Processing Without CPU

Traditional systems process packets in two modes (see Sec. 3.2.6): *i*) polling mode, such as DPDK Poll Mode Driver (PMD), and *ii*) interrupt mode. Most of the widely adopted strategies for network performance enhancement focus on improving the network throughput by: *i*) batching of packets in the NIC during each batch-period before notifying the poller, and *ii*) deferring the interrupt generation for a batch-period by the NIC (e.g., New API [NAPI] of Linux).

The basic trade-offs between state-of-art based interrupts and polling methods are: *i*) Polling wastes CPU cycle resources when there are no packets arriving at the NIC; however, when a packet arrives, the CPU is ready to process the packet almost instantaneously. The polling method achieves low latency and high throughput. However, the polling by the application/network-driver is agnostic to the traffic class, as the driver has no context of what type of traffic and whose traffic is arriving over the link (in the upstream direction) to the NIC. *ii*) Interrupts create overhead at the CPU through context switches, thereby reducing the overall system efficiency, especially for high-throughout scenarios. Although, there exist packet steering and flow steering strategies, such as Receive Side Scaling (RSS) at the NIC, interrupt generation results in significant overheads for heavy network traffic. To note, either through polling-alone or interrupts-alone, or through hybrid approaches: The common approach of the NICs keeping the CPUs alive for delay tolerant traffic imposes an enormous burden on the overall power consumption for servers and clients Gobriel (2012). Thus, future SmartNICs should recognize the packets of delay-tolerant traffic, and decide not to disturb the CPUs for those specific packet arrivals while allowing the CPU to reside in sleep states, if the CPU is already in sleep states. The packets can directly be written to memory for offline processing. Extending this concept, future SmartNICs should be empowered with more responsibilities of higher network protocol layers (transport and above), such that the CPUs intervention is minimal in the packet processing. A pitfall to consider in the design is to ensure the security of offline packet processing by the SmartNIC such that the CPU is not distracted (or disrupted) by the SmartNIC and memory I/O operations, as most security features on the platform are coordinated by the CPU to enable isolation between the processes and threads.

3.5.7 NF Acceleration Beyond Platforms and Infrastructures

Operating Systems and Hypervisors

The Operating System (OS) manages the hardware resources for multiple applications with the goal to share the platform and infrastructure hardware resources and to improve their utilization. The OS components, e.g., kernel, process scheduler, memory manager, and I/O device drivers, themselves consume computing resources while managing the platform and infrastructure hardware resources for the NF applications. For instance, moving packet data from a NIC I/O device to application memory requires the OS to handle the transactions (e.g., kernel copies) on behalf of the applications. While the OS management of the packet transaction provides isolation from operations of other applications, this OS management results in an overhead when application throughput and hardware utilization is considered Yasukata *et al.* (2016). Therefore, several software optimizations, such as zero copy and kernel bypass, as well as hardware acceleration strategies, such as in-line processing Eran *et al.* (2019), have been developed to reduce the OS overhead.

Similarly for hypervisors, the overhead of virtualization severely impacts the performance. Virtualization technologies, such as single root and scalable I/O Virtualization (IOV) **?**Pitaev *et al.* (2018), mitigate the virtualization latency and processing overhead by directly allocating fixed hardware device resources to applications and VMs. That is, applications and VMs are able to directly interact with the I/O device—without OS or hypervisor intervention—for data transactions between the I/O device (e.g., NIC) and system memory of the virtualized entity (e.g., VM). In addition to the data, the interrupt and error management in terms of delivering external I/O device interrupts and errors to VMs through the hypervisors (VMM) should be optimized to achieve short application response times (interrupt processing and error recovery latencies) for an event from the external I/O devices. For instance, external interrupts that are delivered by the I/O devices are typically processed by the hypervisor, and then delivered to VMs as software based message interrupts. This technique generates several transitions from the VM to the hypervisor (known as VM exits) to process the interrupts. Therefore, the mechanism to process the interrupts to the VM significantly impacts the performance of applications running on a VM.

A comprehensive up-to-date survey of both the software strategies and hardware technologies to accelerate the functions of the OS and hypervisors supporting NF applications would be a worthwhile future addition to the NF performance literature.

Orchestration and Protocols

Typically, applications running on top of the OS are scheduled in a best-effort manner on the platform and infrastructure resources, with static or dynamic priority classes. However, NF applications are susceptible to interference (e.g., cache and memory I/O interference) from other applications running on the same OS and platform hardware. Applications can interfere even when software optimizations and hardware acceleration are employed, as these optimization and acceleration resources are shared among applications. Therefore, platform resource management technologies, such as the Resource Director Technology (RDT) Intel Corp. (2019d), enable the OS and hypervisors to assign fixed platform resources, such as cache and memory I/O, for applications to prevent interference. Moreover, the availability of heterogeneous compute nodes, such as FPGAs, GPUs, and accelerators, in addition to CPUs results in complex orchestration of resources to NF applications. OneAPI Intel Corp. (2020a) is an enabling technology in which applications can use a common library and APIs to utilize the hardware resources based on the application needs. Another technology enabling efficient orchestration is Enhanced Platform Awareness (EPA) Ge *et al.* (2014);

Nehama *et al.* (2014). EPA exposes the platform features, such as supported hardware accelerations along with memory, storage, computing, and networking capabilities. The orchestrator can then choose to run a specific workload on a platform that meets the requirements.

In general, an orchestrator can be viewed as a logically centralized entity for decision making, and orchestration is the process of delivering control information to the platforms. As in the case of the logically centralized control decisions in Software Defined Networking (SDN) Zilberman *et al.* (2015), protocol operations (e.g., NF application protocols, such as HTTP and REST, as well as higher layer protocol operations, such as firewalls Fiessler *et al.* (2017), IPSec, and TCP) can be optimized through dynamic reconfigurations. The orchestration functions can be accelerated in hardware through *i*) compute offloading of workloads, and *ii*) reconfiguration processes that monitor and apply the actions on other nodes. Contrary to the centralized decision making in orchestration, decentralized operations of protocols, such as TCP (between source and destination), OSPF, and BGP, coordinate the optimization processes which requires additional computations on the platforms to improve the data forwarding. Thus, hardware acceleration can benefit the protocol function acceleration in multiple ways, including computation offloading and parameter optimizations (e.g., buffer sizes) for improved performance.

In addition to orchestration, there are plenty of protocol-specific software optimizations, such as Quick UDP Internet Connections (QUIC) Langley *et al.* (2017), and hardware accelerations Moon *et al.* (2020); Ruiz *et al.* (2019) that should be covered in a future survey focused specifically on the acceleration of orchestration and protocols.

3.6 Conclusions

This article has provided a comprehensive up-to-date survey of hardware-accelerated platforms and infrastructures for enhancing the execution performance of softwarized network functions (NFs). This survey has covered both enabling technologies that have been developed in the form of commercial products (mainly by commercial organizations) as well as research studies that have mainly been conducted by academically oriented institutions to gain fundamental understanding. We have categorized the survey of the enabling technologies and research studies according to the main categories CPU (or computing architecture), interconnects, memory, hardware accelerators, and infrastructure.

Overall, our survey has found that the field of hardware-accelerated platforms and infrastructures has been dominated by the commercial development of enabling technology products, while academic research on hardware-accelerated platforms and infrastructures has been conducted by relatively few research groups. This overall commercially-dominated landscape of the hardware-accelerated platforms and infrastructures field may be due to the relatively high threshold of entry. Research on platforms and infrastructures often requires an expensive laboratory or research environment with extensive engineering staff support. We believe that closer interactions between technology development by commercial organizations and research by academic institutions would benefit the future advances in this field. We believe that one potential avenue for fostering such collaborations and for lowering the threshold of entry into this field could be open-source hardware designs. For instance, programmable switching hardware, e.g., in the form of SmartNICs and custom FPGAs, could allow for open-source hardware designs for NF acceleration. Such open-source based hardware designs could form the foundation for a marketplace of open-source designs and public repositories that promote the distribution of NF acceleration designs among researchers as well as users and service providers to reduce the costs of conducting original research as well as technology development and deployment. Recent projects, such as RISC-V, already provide open-source advanced hardware designs for processors and I/O devices. Such open-source hardware designs could be developed into an open-source research and technology development framework that enables academic research labs with limited budgets to conducted meaningful original research on hardware-accelerated platforms and infrastructures for NFs. Broadening the research and development base can aid in accelerating the progress towards hardware designs that improve the flexibility in terms of supporting integrated dedicated acceleration computation (on-chip), while achieving high efficiency in terms of performance and cost.

Despite the extensive existing enabling technologies and research studies in the area of hardware-accelerated platforms and infrastructures, there is a wide range of open challenges that should be addressed in future developments of refined enabling technologies as well as future research studies. The open challenges range from hardware based polling in the CPUs and CPU based hardware acceleration management to open challenges in reconfigurable cross-chip interconnects as well as improved heterogeneous memory access. Moreover, future technology development and research efforts should improve the accelerator operation through creating a common context for accelerator devices and allocating accelerator resources based on the context. We hope that the thorough survey of current hardware-accelerated platforms and infrastructures that we have provided in this article will be helpful in informing future technology development and research efforts. Based on our survey, we believe that near-to-mid term future development and research should address the key open challenges that we have outlined in Section 3.5.

More broadly, we hope that our survey article will inform future designs of OS and hypervisor mechanisms as well as future designs of orchestration and protocol mechanisms. As outlined in Section 3.5.7, these mechanisms should optimally exploit the capabilities of the hardware-accelerated platforms and infrastructures, which can only be achieved based on a thorough understanding of the state-of-the-art hardware-accelerated platforms and infrastructures for NFs.

Moreover, we believe that it is important to understand the state-of-the-art hardware-accelerated platforms and infrastructures for NFs as a basis for designing NF applications with awareness of the platform and infrastructure capabilities. Such an awareness can help to efficiently utilize the platform and infrastructure capabilities so as to enhance the NF application performance on a given platform and infrastructure. For instance, CPU instructions, such as `MOVDIR` and `ENQCMD` Intel Corporation (2020d), enable applications to submit acceleration tasks directly to hardware accelerators, eliminating the software management (abstraction) overhead and latency.

3.7 Introduction

3.7.1 Motivation

In traditional implementation of Network Functions (NFs) where the dedicated (non-programmable) compute infrastructures process the packets, the packet processing latency and overall system throughput could be deterministically characterizable. Hence, the Quality-of-Service (QoS) of an end-to-end link could be strictly estimated. However, due to the nature of vendor specific design of dedicated (non-programmable) compute infrastructures, the network processing nodes (e.g., switches, routes, and gateways) are typically in the form of closed black-boxes and proprietary. In addition, the NF implementation on the dedicated infrastructures were also non-reconfigurable and non-programmable for redefining the NFs limiting the scalability and flexibility. To this end, the recent trends in the NFs design and implementation have focused on utilizing the General Purpose Compute (GPC), e.g., x84 (Intel), x64, AMD, and ARM processors. The packet processing is implemented on the GPC infrastructures that run the Operating Systems (OS) and HyperVisors (HV) which hosts applications defining the NFs. Hence, the application development provides a great degrees of freedom in terms of flexibility and scalability in designing a NF while achieving programmability and reconfigurability.

The need for configurable and programmable networks has grown in significant demand in recent network infrastructure development. To this end, the paradigm of Software Defined Networking (SDN) has been widely adopted to enable high degrees of configurability and programmability. Whereas, the technology of Network Function Virtualization (NFV) has been adopted to for enabling scalability and flexibility. Therefore in conjunction to softwarization of NFs implemented as applications on a OS/HV running on a GPC infrastructures, the SDN and NFV have presented unprecedented opportunities for the future network designs, while bringing numerous challenges to the GPC infrastructures and OS/HV to offer efficient packet processing strategies.

A close examination into the characteristics of packet processing by the OS/HV on the GPC infrastructure provides insights into the optimization scope for platform hardware and OS/HV design principles (e.g., memory management). The challenges of packet processing in the GPC infrastructures arises from the traditional methods of hardware resource management principles of OS/HV, such as memory abstraction (user and kernel spaces). The primary requirements of OS/HV were to share the hardware resources among multiple tenants (i.e., applications) and provide isolation.

Characteristics of Packet Processing on General Purpose Compute (GPC)

When a packet processing NF is implemented as an application on an OS/HV using traditional approaches, i.e., without any OS/HV optimizations and hardware accelerations, following challenges may arise depending on the packet arrival rate:

i) *I/O intensive*: Packet arrival to the Network Interface Card (NIC) at the line rate results is an overwhelming I/O activity in the OS/HV system. The I/O activity would impose burden on the interconnects for data hauling between processing elements in the GPC platform, both on-chip (e.g., Advanced eXtensible Interface [AXI]) and chip-to-chip (e.g., Peripheral Component Interconnect extended [PCIe]).

ii) *Compute intensive*: The wide range of packet processing type i.e., from a simple header-lookup to relatively larger compute such as packet encryption (e.g., IPSec), and deep packet inspection involving both arithmetic and logical operations would impose large burden on CPU. In addition, the packet size, packet rate [bit/sec], and inter-packet arrival duration determine the overall load on the CPU.

iii) *Repeated and routine operations*: The compute on the packets by the cores, are routine in nature and repeated for every packet arrival. For instance, the L2 forwarding (switching) NF application would inspect the header, and identify the forwarding port for every incoming packet.

iv) *Latency sensitive*: As processes and threads are scheduled on the core for processing based on their relative priorities, there is room for variation in the processing latencies which would inducing jitters in the packet flow. Therefore, the softwarize NF system should consider minimizing delay variations, in addition to the traditional buffering and queuing delays.

v) *Dedicated capacity*: Softwarized NF that are designed to operate at line rate, e.g., 10 Gbps for each port, should reserve hardware resources such as number of cores, and memory to support dedicated capacity.

iv) *Large memory*: Softwarized NFs that implement gateway functions need to reserve large system memory to buffer the data from multiple ports. For instance, an aggregation gateway node that multiplexes traffic from *N* links to one outgoing port in `N:1` fashion.

Characteristics of OS/HV on General Purpose Compute (GPC)

Softwarized NF for packet processing on GPC platforms imposes restrictions on the OS/HV strategies to achieve latency and throughput requirements. The goal since the inception of OS/HV were to maximize the utilization of platform hardware resources (e.g., CPU, memory, and I/O devices) by running multiple applications concurrently. Hence the characteristics of OS/HV in terms of running multiple application on a common hardware platform can be summarized as:

i) *Compute*: Hyper-threading, C/P states (turbo), processes and thread scheduling, priorities, context switching, caches, and thermal characteristics

ii) *Memory*: NUMA, abstractions, and traditional OS memory management, kernel space, user-space

iii) *I/O devices*: NIC, Accelerators and Storage: interrupts, polling, memory movement, power management

iv) *Virtualization*: Hypervisors, CPU, memory, I/O device virtualization.

This paper surveys the enabling technologies and research studies on OS and hypervisor aspects that directly impact the performance of NFs. The NF softwarization relies on OS services, hardware control, and resource management for the implementation and deployment of NFs as applications hosted on the OS. A hypervisor, on the other hand, provides an abstraction layer of the hardware to aggregate and slice resources for an isolated abstracted environment. In practice, a hypervisor can be considered as a fully functional OS which is built to host applications as well as other OSs. As discussed in Sec. 3.2, NF applications can be in developed in the form of applications hosted directly on an OS interacting with hardware, on a container, or a VM. Therefore, NF performance directly relates to the OS and hypervisor management as well as the allocation of hardware resources to the applications.

3.8 Enabling Technologies

3.8.1 Abstraction Layer

The abstraction layer manages the hardware resources and provides hardware access services to applications running on the OS (or guest OS in case of hypervisors). For NF applications, reducing the overhead of hardware access and reducing the memory access latency by the CPU are primary concerns for improving the overall performance.

OS The traditional OS kernel involves several software overheads resulting from the isolation of user and kernel space in the system memory, paging process, and security protection methods that incur memory copies as well as process and transactional latencies for applications. For instance, a packet that arrives to the NIC from the physical link is processed as follows: *i*) the packet is DMAed into kernel memory space managed by the NIC driver, *ii*) the CPU moves the packet from the kernel space to user-space NF application memory, *iii*) the CPU moves the packet into the kernel-space of the accelerator driver, *iv*) packet data is moved from the kernel space of the accelerator driver to the hardware accelerator memory, *v*) the accelerator writes back the packet to the kernel-space after processing, *vi*) the CPU moves the packet from the kernel-space to the user-space memory of the application, *vii*) the CPU writes the packet again into the NIC driver kernel-space memory region, and *viii*) the packet is DMAed into the NIC memory, and the packet is sent out onto the link, as illustrated in Fig. 3.39. In addition to memory transaction overheads, the synchronization of threads and processes between the common queue increases wait times resulting from the locking and unlocking of resources. Thus, traditional OS packet processing methods are inefficient to achieve high throughput and low latencies for NF applications.

Several OSs optimized for NF applications have been recently proposed, such as OS^v OS^v (2020) to address the overheads resulting from synchronization and locking through spin-lock free implementation and NFVTime Telco Systems, A Bath Company (2020) to address overheads arising from virtualization and memory access. A OS can host multiple NF applications which many need traffic communication paths between NFs, in traditional designs, a packet from a source NF would traverse to an external switch and be received back before reaching the destination NF on the same host. To reduce such traffic loop-back through an external switch, the kernel can implement a virtual switch function as a kernel component. The Open vSwitch (OvS) is an open source implementation of a virtual switch that forwards traffic between multiple applications within the OS kernel. OvS is supported by the extended Berkeley Packet Filter (eBPF) Fleming (2017) developed by the project IO Visor Linux Foundation (2020) of the Linux Foundation. The eBPF implements efficient packet in-kernel processing methods through a graph based approach to reduce the number of computations required for packet matching and filtering. In the case of SDN based virtual switches, eBPF is employed to implement packet analytics, monitoring and inspection, and debugging of packets by filtering and forwarding the packets to an SDN controller.

If communication between NFs spreads across multiple hosts, a Logical Distributed Switch (LDS) can be implemented as a part of a kernel-module as extension to virtual switches. An LDS spans across multiple hosts where traffic is tunneled between LDS switch components for dedicated logical L2 connectivity using protocols, such as VxLAN and Generic Routing Encapsulation (GRE) protocols for within host and host-to-host packet transactions within L2 and L3 domains.

The Address Family eXpress Data Path (AF-XDP) Hohlfeld *et al.* (2019) is a method for high speed packet processing aiming to reduce memory copying overhead through a common memory region for both application and device driver, effectively resulting in zero-copy processing. An application reserves a part of contiguous virtual memory, referred to as UMEM, as an array and shares the address of the UMEM with the driver. The NIC can then directly write into system memory into the UMEM for packet receptions and read from the UMEM for packet transmissions.

Hypervisors The virtualization of hardware resources provides a framework to multiplex multiple OS workloads (Virtual Machines [VMs]) onto a single set of hardware resources, improving the overall utilization of the physical resources. The multiplexing of OS workloads is achieved by the hypervisors, whereby each VM is assigned independent abstracted (virtualized) resources while maintaining isolation between the VMs. The main downside of virtualization is the overhead of memory, CPU, and I/O device abstraction, which requires multi-level page walks, virtual to physical mapping of CPU, I/O device DMA, and interrupt remapping. As the number of CPUs and I/O devices, size of memory, and number of VMs increase, a hypervisor incurs a large overhead to map between physical and abstracted hardware resources. Generally, NF applications are memory and I/O intensive workloads with a large number of recurring events generated by packet reception, processing, and transmissions which results in large overheads, reducing the overall system throughput.

Therefore, hardware assisted virtualization provides enhanced features for VMs to interact directly with the hardware. For instance, for CPU virtualization, VMX instruction described in Sec. 3.3.1 allow VMs to directly execute CPU instructions on the physical CPUs. In the traditional methods, VM to Hypervisor (i.e., Virtual Machine Monitor [VMM]) control transitions were managed through TLB flushing Shah and Patil (2013) and reestablishment of page translation caching, which increased memory access latencies. To avoid the TLB flush process during VM and VMM transitions, a virtual-MMU (vMMU) Yang (2008) supported by hardware has been designed to allocate an independent memory management context to VMs. The vMMU uses a Virtual Processor IDentifier (VPID) to identify the TLB cache lines utilized by both VMs and VMM, thus avoiding the TLB cache flush during VM-VMM transitions with an Extended Page-Table (EPT) Merrifield and Taheri (2016) mechanism whereby hardware based nested address translation is employed.

Containers Containers are an abstraction at the application layer that packages code and dependencies together. A Container Engine, such as Docker and CRI-O Rodriguez and Buyya (2019), manages the containers at run-time for instantiation, resource allocation, isolation, saving contexts and tear-down. In general, hypervisor and VM implementations target the isolation property in the infrastructure and application deployment, while container implementations target the flexibility and scalability.

The traditional implementation of containers and VMs involves multiple levels of abstraction for effective management of the hardware resources and application demands. Multi-level abstractions with nested VMs and containers are particularly necessary for network and resource slicing in MEC, where high degrees of isolation and flexibility are needed. For instance, VMs running on an MEC can provide service provider level isolation, whereas containers running inside VMs can provide application level isolation for the CNFs. Moreover, the dynamic nature of NFs requires flexibility and agility in terms of network service instantiation, service migration, and scalability which makes containers an ideal choice for the NF softwarization.

Clear Linux Hardware acceleration of containers provides enhanced features to further optimize the application deployment in terms of scalability and isolation. Intel® Clear Linux Intel Corporation (2020a) containers optimize kernel and application execution on Intel® CPUs with AVX instructions and optimized libraries for software APIs along with kernel configurations, compilers, and auto-select of hardware enhanced features for applications, such as Intel® Virtualization Technologies (VT) Van Doorn (2006). Clear Linux also utilizes hardware features to separate the file systems for user data, operating systems, and system configuration for easy management Bahena and de Alba (2014).

Kata Container In an effort to get the best of both VMs and containers, Kata containers present the combination of Clear Linux and Hyper RunV projects to achieve the flexibility and speed of containers while still preserving the VM level isolation and security. More specifically, Kata containers can achieve the performance of Clear Linux, e.g., < 100 ms boot time while supporting heterogeneous CPU architectures and hypervisors. This is achieved by exploiting hardware enhanced features, such as CPU AXI and VT technologies. As opposed to multiple containers on a VM, the Kata container model involves only a single container per VM. Each VM instantiates a Clear Linux container on a virtualized hardware by the Linux kernel (acting as a hypervisor) as illustrated in Fig. 3.41.

Summary of Abstraction Layers Abstraction is necessary to provide security, flexibility, and scalability to NF applications. However, the more abstraction layers, the larger is the overhead as the control and data messages have to traverse through each abstraction layer, resulting in multiple copies of messages as they are moved through the abstraction layers. Each abstraction layer is also associated with resource management, which incurs further overhead. Therefore, the NF performance is significantly reduced compared to native bare metal as the number of abstraction layers increases. To overcome the hypervisor overhead, lightweight containers have been developed to alleviate the problems of full abstraction to shared abstraction (whereby shared abstraction reuses the resources or arbitrates the access to the abstracted resources). Clear Linux and Kata containers utilize hardware enhanced features to achieve the best of both container and VM. While there exit multiple deployment options, it is important to carefully consider the pitfalls of abstraction which could impede the overall performance, such as, resource allocation for slicing and management, application level isolation and security, and access control as well as policy management in heterogeneous deployments involving containers, VMs, hypervisors, and hardware.

3.8.2 Memory Access

Non-Uniform Memory Access (NUMA) In the initial GPC generations, the system memory (i.e., DRAM) was connected to a common system bus which gave all CPUs equal memory accessibility in terms of throughput and latency. Thus, the CPU access characteristics to memory were uniform regardless of the relative locations of core and CPU in terms of socket, and the memory access method was referred to as Uniform Memory Access (UMA). However, in the recent designs with the increasing numbers of cores per CPU, the system memory is partitioned into memory nodes, whereby each memory node corresponds to one CPU (or socket). The CPU memory access characteristics to their respective memory nodes are faster relative to the memory nodes that are not directly connected to their socket. Fig. 3.42 shows the local and remote memory access methods by the CPU, resulting in asymmetric memory access characteristics for different memory nodes, referred to as Non-Uniform Memory Access (NUMA). With the support from the OS, NF application designs should consider the memory access based on the NUMA configuration such that there is no performance impact due to non-NUMA node memory access. That is, non-NUMA memory access should be avoided since a non-uniform memory access would degrade performance. Therefore, the data should be maintained with locality in place such that memory access is uniform.

Similar to NUMA nodes which define socket-specific memory regions, the NIC and other devices, such as accelerators, which are based on PCIe, could also be connected to CPUs through socket-specific nodes. The access characteristics to these devices by the CPU are faster if they are connected directly to the CPU socket. Hence, NF applications require NUMA knowledge from the OS to achieve high performance levels from NICs and accelerators.

Continuous Memory Allocation Each memory node in NUMA is further divided into continuous memory chunks (also referred to as pages) in the system memory. The OS accesses the system memory in terms of pages as opposed to bytes of data. The typical page size during normal operation is 4 Kbytes. Therefore, for each CPU memory access, the data which belongs to a block of 4 Kbytes is entirely copied from the secondary storage to system memory for the application processing. Each page is accessed through a page index to which each address is mapped to by translating virtual addresses used by the OS to physical addresses in system memory. The CPU evaluates the page index during the run-time of application every time there is a request for memory access and performs a look-up for the page index in Translation Lookahead Buffers (TLB). The TLB caches all the page index mappings from virtual to physical memory addresses. If there is a TLB miss, then the CPU searches through an elaborated page table for the virtual to physical address mapping of the page index which incurs a significant performance overhead.

For NF applications, the TLB misses would result in non-deterministic performance characteristics in terms overall throughput and latencies of packet processing. To avoid TLB misses, large continuous memory regions can be allocated to NF applications by allocating a large page ($\gg$ 4 Kbyte) from the OS. With a large page, the page index mapping is available with high probability as cached entity in the TLB. Typically, NF application, such as DPDK, allocate huge pages of 1 Gbyte, compared to 4 Kbyte allocations for traditional OS applications. However, the downside of huge pages are memory wastage, as a complete page may not be used by the application, reducing the overall OS memory utilization. Therefore, NF applications should consider the memory requirements when allocating the page size to avoid system memory wastage.

IO Memory Management Unit (IOMMU) The MMU is responsible for the translation of the virtual address of the OS to the physical address of the system memory when requested by the CPU. In the case of I/O devices, system memory access can be enabled in two ways: *i*) direct physical address memory assignment, or *ii*) address translation via the MMU which takes additional steps for the CPU to coordinate the address translation for the devices. Direct physical address access can result in security vulnerabilities, e.g., a device can access memory regions that are not related to the device. Avoiding such vulnerabilities, the IO Memory Management Unit (IOMMU) Advanced Micro Devices Inc. (2016) has been designed to enable the virtual to physical address translations. A traditional IOMMU maintains its own page tables through IO Virtual Addressing (IOVA) which is different from CPU page tables, whereby the I/O device and CPU maintain their own TLB and page fault resolution mechanisms.

To further enhance the memory access performance, Shared Virtual Memory (SVM) has been introduced to obtain a common memory view across both I/O device and CPU, such that a pointer to memory allocated by an application in user space can be passed to the device for memory access. In SVM, both system MMU and IOMMU are coherent at all times and are in coordination to ensure consistent translations between virtual and physical addresses. The OS provides applications with unique virtual addresses, which are common to other applications, but map to independent physical addresses in memory. That is, the same virtual addresses could be used by multiple processes, but are still independent in physical memory. OS and MMU would identify the processes based on address space identifiers. Therefore, for SVM, the Process Address Space IDentifiers (PASID) context must be passed down to the device along with the virtual addresses of the applications such that the IOMMU can perform the address resolution specific to each application based on the PASID. In case of hypervisors, the PASID is used for the guest OS address space resolution by the IO devices.

Summary of Memory Access Memory intensive NF applications depend on strategies and technologies that overcome the traditional bottleneck in the memory access. Edge applications, such as a Content Distribution Networks (CDNs), frequently receive and forward data that needs to temporarily stored at edge nodes. At the system level, the data arriving at the NIC (I/O device) is copied to several memory regions before the application can process the data. As the number of processor sockets increases on a platform, an access can fall into a region that is part of the current socket or a region that is part of a different socket. The memory access latency differs between these two cases, which can result in asymmetrical performance. Additionally, the memory fragmentation by the OS for paging process is susceptible to TLB thrashing and page faults causing additional overheads.

Memory access strategies try to overcome these limitations through memory pinning of applications for NUMA, as well as contiguous memory allocations and large pages for memory fragmentation and TLB thrashing. Also, IOMMU and Shared Virtual Memory (SVM) have been developed to assist I/O devices in reading and writing data from system memory. Cache coherency mechanisms further mitigate the memory access latency experienced by the CPU by keeping the updated data in caches. Thus, memory access plays an important role in the overall NF application development and deployments.

3.8.3 Accelerator Offload Designs

Accelerators are typically enumerated in the OS as PCIe devices. Physically, the PCIe accelerator devices can be on-chip within the CPU die, or connected externally through a PCIe bus and on-board connectors. In either case, the PCIe protocol defines the procedures to send instructions to the accelerator to read data from the DRAM, perform computations on data, and write back data to the DRAM. The accelerator driver component of the OS, in conjunction with the PCIe driver, takes care of the protocol operations and transactions. An application that needs to utilize the accelerator services has to interact with the accelerator service drivers of the

OS. A typical control flow for task offloading to the accelerator includes registering of the application with the accelerator device, task submission to the accelerator worker-queue, monitoring for a response from the accelerator device, and access the results. This section presents an overview of the different methods applied by the OS and applications to interact with a hardware accelerator.

For the NF acceleration which involves packet processing, the function task offloading can be performed in two ways: *i*) In-line processing, whereby tasks are offloaded to an acceleration module inside the NIC itself as packets arrive to the NIC I/O device, or *ii*) Look-aside processing, where tasks are offloaded to an additional acceleration device that is external to the NIC.

In-Line Processing (ILP) Task acceleration that involves the processing of packets inside the NIC as packets arrive at the NIC is referred to as In-Line Processing (ILP) Eran *et al.* (2019). ILP is an effective method of acceleration in terms of latency and memory transactions to-and-from DRAM and an accelerator for NF application as the benefits of function processing are achieved at the packet origination and termination points (for packet transmission and reception over a link). However, the downsides and pitfalls of implementing the acceleration component at the NIC for ILP are the cost of integrating specific complex function accelerations, such as HTTP load balancing on the NIC, and the reduced flexibility for using accelerators for applications that involve packet processing. The acceleration of complex functions, such as HTTP load balancing, requires the NIC to maintain application states in order to be able to analyze the incoming packets beyond the traditional IP layer processing. Therefore, the ILP should be applied only to simple packet processing tasks, such as tunneling, MACSec, and IPSec protocol processing of L2 and L3 processing tasks, which do not impose restrictions, nor increase the complexity of the overall NIC functionality.

Look-Aside Processing (LAP) In contrast to ILP, Look-Aside Processing (LAP) is a traditional method of accelerator task offloading to an external (to CPU) PCIe component. In LAP, upon receiving the packets from the NIC, the NF application is required to send dedicated instructions to the accelerator to read the packet data into the accelerator device, processes the data on the accelerator, and write the data back to the desired memory location that can be read by the application. The application can then further process the accelerator data or interpret the result. As opposed to ILP, LAP does not require any specialized hardware design nor software changes. The main downside of LAP is the large overhead to move packet data between the kernel and user space regions of the DRAM memory, as well as to-and-from the PCIe devices (i.e., NIC and accelerator device). For small packets, the memory movement overhead, i.e., the offloading cost is higher than the acceleration benefit, resulting in overall reduced application performance.

Interrupt Mode When a task is offloaded to an accelerator, the CPU can either choose to poll the accelerator status for task completion or ask for an interrupt to be sent to the CPU (in response to the interrupt, the CPU can then further process the results from the accelerator). The interrupt generation and the CPU response to the interrupt can be of various forms. Physical interrupts via dedicated interrupt lines, such as INTA, INTB, INTC, and INTD (i.e., INTx), are used to get the attention of the CPU for highest priority interrupts. Physical interrupt lines are limited and need to multiplexed for sharing interrupts between multiple I/O devices. To overcome this limitation, the PCIe specification introduced Messaged Signaled Interrupts (MSI) Tu *et al.* (2015), which replace the physical interrupts with a standard memory write process, whereby the memory address is mapped to the interrupt type and the data value written to memory is the message conveying the interrupt details. While the physical INTx interrupts are limited to 4, the MSI extend the interrupt capability to 32 interrupt vectors per PCIe function. To further enhance the MSI capabilities, MSI-eXtensions (MSI-X) have been designed to extend the interrupt vector size to 2048 MSI-X interrupts per PCIe functions. MSI-X consists of a unique memory address for each interrupt and data value to indicate the interrupt details.

The interrupts generated by a device in the form of MSI and MSI-X are sent to the PCIe Root Port (RP). The RP forwards the MSI-X messages to the I/O Advance Interrupt Controller (I/O APIC) which is mapped to the indicated MSIx address. Each CPU core has a local APIC as indicated in Fig. 3.44. The I/O APIC identifies to which local APIC the MSI-X interrupt needs to be delivered. The APIC is also responsible for handling the interrupts generated by timers, performance monitors, and thermal sensors. The APIC also considers interrupt forwarding between cores through Inter-Processor Interrupts (IPI) for load balancing, handling software generated self-interrupts, and preemptive scheduling.

NF applications can use the APIC controller and IPI strategies for efficient packet processing methods, such as Receive Side Scaling (RSS), Receive Packet Steering (RPS), Receive Flow Steering (RFS), and Receive Side Coalescing (RSC). RSS is primarily intended to distribute interrupts across multiple cores for packet processing. Without RSS, all packets for a given socket connection arrive to a single processing core which could increase the overall processing time. RSS uses hardware queues to distribute the packets among multiple cores. On the other hand, RPS uses IPI mechanisms and software queues to distribute the packets to different cores based on protocol needs. RFS targets a specific core based on the application running on the CPU to increase the cache hit rate such that the processing latency is decreased. In contrast, RSC coalesces the TCP/IP packets which belong to the same socket connection into a single packet, effectively reducing the interrupts between NIC and

CPU as well as the per-packet processing overhead.

Figure 3.45 illustrates the interrupt delivery mechanisms to the Guest OS running on a VMM (hypervisor). Traditional methods of interrupt delivery by the VMM to the Guest OS involve a software management of the external interrupts on the VMM. The VMM management of the external interrupts by the software component results in VM exits (transfer of CPU "privileged mode" control) between the Guest OS and VMM. The trapping of the VM exits by the VMM and delivering the interrupts to the Guest OS incurs significant delay because of VM exits. The CPU hardware virtualization of the APICv allows the APICv context to be allocated to the Guest OS, and external interrupts can be delivered directly to the hardware APICv by the VMM, without a VM exit. Although this mechanism reduces the number of VMM exits between the VM and VMM, the software delivery of interrupts that are not registered by the APICv would still result in VM exits. An improved version of the hardware based APICv is to enable posted mode interrupts, whereby an external interrupt is delivered as a message written into a dedicated memory region. Posted interrupts eliminate the requirement of VMM, as the VMM is transparent to the external interrupts except for the hardware based APICv registered interrupts. The posted interrupt benefits include fewer VM exits (for hardware APICv), easier migration of interrupt contexts (whereby the context resides in a VM as opposed to a VMM), and support for I/O virtualization technologies (see Sec. 3.8.3) for delivering I/O device based interrupts.

Polling Mode In contrast to the CPU registering an interrupt from an accelerator, the CPU can choose to poll the status of completion of the accelerator. Polling typically involves a repeated (periodic or aperiodic) memory read operation of a specific address (e.g., designated completion status address) in the memory region where an accelerator updates the status of task completion. More generally, the CPU offloads the task with the submission of a descriptor to the worker queues of the accelerator. The descriptor includes the details of the memory regions where the completion status and result should be written back once the accelerator completes the task. After task completion, the accelerator DMAs the results to the system memory (i.e., DRAM) and updates the completion status to the address provided by the CPU. The software polling thread recognizes the completion status value during the next polling read operation and notifies the relevant process thread in the OS for the result interpretation.

For NF application development, the Data Plane Development Kit (DPDK) includes a Poll Mode Driver (PMD) Cerrato *et al.* (2014). The PMD continuously monitors (i.e., polls) the NIC for a packet arrival such that the CPU can process the packets almost instantaneously. In contrast, waiting for an interrupt and then invoking interrupt service routines generally incurs additional packet processing latencies since the packet processing threads will be put to wait states after timeouts when there are no packets (and waking up a thread from a wait state incurs delays). Also, the raising of the interrupt by the NIC to the CPU incurs latencies, and the CPU has to preempt currently ongoing executions to service the interrupt. PMD avoids these additional latencies as the PMD thread is always ready for packet processing. The PMD maximizes the throughput from an overall system perspective, however, each PMD thread is pinned to a core to monitor and service packets from the NIC. The continuous polling of the PMD thread fully (100%) utilizes the core. Hence, one core is completely occupied by the PMD for an NIC port. Thus, the PMD improves the system throughput and latency at the expense of high power consumption (as compared to interrupt mode) since the CPU is continuously active, even when there are no packets.

Summary of Accelerator Offload Designs Hardware acceleration is associated with two main offload costs: *i*) moving input and output data through the accelerator and system memory, and *ii*) notification of task completion by the accelerator. The data movement among application user space, accelerator kernel space, and accelerator device memory requires in-memory copying, increasing the overall processing duration of the accelerator; this traditional method of offloading is referred to as "Look-Aside Processing". In contrast, for NF applications, packets can be processed by accelerators at the NIC as they arrive, without requiring the packets to be moved between system and accelerator memories; this accelerator processing approach is referred to as "In-Line Processing".

Notification of accelerator task completion to the CPU can be achieved through a) interrupt, or b) polling. In the interrupt mode, the accelerator I/O device generates an interrupt to the CPU to check the accelerator output at a predetermined memory location. The interrupt method incurs overhead, especially when the CPU is in a sleep state and requires a recovery time for wake-up to start processing the accelerator output. In contrast, the polling mode keeps the CPU awake and constantly checks the accelerator for completion, which minimizes the delay for processing the accelerator output by the CPU. However, polling is energy-inefficient, as the CPU is kept awake for the entire duration of the accelerator offload processing.

Hardware Assisted I/O Virtualization

Beyond the traditional IT server infrastructure management, virtualization is particularly important in MEC environments as MEC infrastructure resources are typically shared among multiple tenants. MECs are usually owned by private third party businesses, such as coffee shops, shopping malls, schools, and universities. Third party infrastructures are then leased by multiple service providers, whereby the MEC infrastructures are resource sliced to provide isolated environments through virtualization. Thus, virtualization plays a vital role in MEC deployments.

If an NF requires limited OS support, for instance to establish a packet filtering mechanism, a lightweight container could host the function capabilities for the NF. In contrast, complex NFs may require access to the full host OS capabilities. For example, enforcing QoS through time synchronization, load balancing at the application layer, and traffic shaping require full host OS access in isolation for complete hardware control by the NF applications.

Para-virtualization Orthogonal to device virtualization techniques, the concept of para-virtualization is to specifically design the virtualized entities to run on a specific abstraction layer (physical device or software environment), rather than being agnostic to the hardware and software layers that implement the virtualized entity. That is, in para-virtualization, the software is specifically adapted to interact either with a physical device or with a virtual (software) environment. This technique takes advantage of prior knowledge of the physical device and virtual environment such that the appropriate software can be run on the OS to optimally interact with the physical device or virtual environment. In traditional virtualization techniques, the software is agnostic to the virtualization, i.e., the same software would interact with both physical devices and virtual environments.

I/O devices NFs primarily depend on two types of I/O devices for accomplishing their tasks: *i*) NICs to communicate with external nodes through physical network interfaces, and *ii*) accelerators for NF application task offloading. Therefore, I/O device virtualization is a key aspect of designing NFVs in a virtualized platform. I/O device virtualization can be achieved in different forms, namely through *i*) software emulation, *ii*) hardware based virtualization, or *iii*) hardware assisted software virtualization.

i) Software emulation: An accelerator function can be implemented in software by emulating accelerator functions that are typically implemented in hardware. Software emulation helps NFs in two ways: *a*) saving of accelerator context during service migration of NFs over virtualized infrastructures, and *b*) sharing of a hardware accelerator function among multiple VMs through abstraction. Software emulation provides continuity of NF application migration when a VM is migrated to a platform, but no hardware acceleration is found. However, software emulation results in an additional overhead for emulation and abstraction.

ii) Hardware based virtualization: With hardware based virtualization, a hardware accelerator function is programmed to operate as multiple virtual functions with static resource slicing. For instance, a single physical I/O device exposes itself to the OS as multiple virtual functions that are pre-defined in the hardware. These virtual functions are then assigned to VMs and application to operate in isolation. Single Root I/O Virtualization (SR-IOV) Dong *et al.* (2008) technology defines the specification for hardware based virtualization.

iii) Hardware assisted software virtualization: In hardware assisted software virtualization, a single I/O device is both emulated in software and hardware. The emulated device is sliced into a flexible number of virtual functions to provide a higher resolution of resource slicing as opposed to a fixed number of slices in hardware based virtualization. Thus, hardware assisted software virtualization has more degrees of freedom in terms of resource slicing of accelerator hardware and allocation of virtual functions to VMs and applications. Scalable-I/O Virtualization (S-IOV) Jani *et al.* (2019) technology defines the specification for hardware based virtualization.

Single-Root I/O Virtualization (SR-IOV) The PCIe specification defines a hardware based virtualization technique that extends a single Physical Function (PF) of a device into multiple Virtual Functions (VFs) that can be assigned to VMs and applications Kutch (2011). The extension of a PF to multiple VFs can only be made to a single Root Port (RP); in other words, VFs from a single device cannot communicate with multiple RPs. All VFs respond to requests and report errors to a single RP and this virtualization concept is therefore referred to as Single-Root I/O Virtualization (SR-IOV). SR-IOV manages the application requests from multiple virtualized entities and applications into a single work queue by multiplexing and managing the result distribution to requesters. An SR-IOV device maintains a different work queue and command processing pipeline individually for each VF within an IO device. The number of VFs for an I/O device is fixed in SR-IOV for an I/O device as the resources per VF are fixed and statically defined. In case of hypervisors, an SR-IOV VF function can directly DMA the I/O data to the VM's physical memory without having the hypervisor/VMM manage the data transfers. This bypassing of the VMM control of the I/O enhances the efficiency as compared to the VMM managing the I/O device. However, a downside is the context loss when a VM is migrated to another node. Due to the context loss, a VMM lacks information about the state of the I/O device to ensure the continuity of application interactions with the I/O device.

Scalable I/O Virtualization (S-IOV) The Scalable-IOV (S-IOV) Intel Corp. (2018) virtualization technology targets two primary downsides of SR-IOV: *i*) fixed allocation of resources for each VF, and *ii*) preserving context of an I/O device to facilitate VM migration. In S-IOV, the resource slicing and allocation to VMs and applications are performed through Assignable Device Interfaces (ADIs). Each ADI is an instance of a I/O device, fully capable of receiving worker queue tasks and commands for accelerator offloading. The S-IOV implements an interrupt posting hardware unit and an Interrupt Message Store (IMS) for providing interrupt support to ADIs. The IMS interrupts, which are posted as messages, can be directed towards a physical, virtual (load balanced among multiple cores), or logical (remapped from physical) core.

S-IOV technology also supports Process Address Space IDentifier (PASID) (See Sec. 3.8.2) which allows ADIs to directly read and write data to application and VM memories without transitioning through OS and hypervisors. Another differentiation of S-IOV from SR-IOV is the implementation of the Virtual Device Composition Module (VDCM) which is an emulated instance of an IO device within the OS or hypervisor. The VDCM decouples the hardware interaction for infrequent I/O device accesses, thus avoiding congestion in the hardware resources which are freed up for the usage from applications that demand more frequent usage. Thus, the VDCM separates the "slow-path" from the "fast-path" for efficient hardware resource management. In addition, the VDCM facilitates the live migration of VMs by preserving the device contexts within the emulated hardware instance. The potential downsides of S-IOV from SR-IOV include software complexity and memory requirements by the VDCM in the S-IOV.

Summary of Hardware Assisted I/O Virtualization In summary, an OS or hypervisor (VMM) would support both SR-IOV and S-IOV technologies for I/O device (i.e., accelerator) virtualization. However, application usage requirements may determine the level of flexibility that is required for accelerator offloading. For example, IO emulation may be best suited for supporting missing hardware features that applications are expecting. Fixed IO assignment—as in the case of SR-IOV—may provide the best performance when hosting I/O-intensive workloads, such as packet inspection and forwarding. Although, the S-IOV adds the software complexity of the VDCM, the "fast-path" access allows applications to directly interact with the device. Also, the IMS enables the device to send interrupts to the cores that the application or VM is running on. As a result, the S-IOV allows I/O device sharing with isolation in multiple scenarios from 1-to-1, 1-to-N, over-provisioning, and emulating for compatibility mapping of hardware ADIs to VDCM based Virtual DEVices (VDEV), as illustrated in Fig 3.47.

Alternative to the direct assignment of I/O devices to the VMs and applications (which is also referred to as "pass-through"), implementing an NF as a virtualized entity, and using a virtualized I/O device by the NF should be carefully considered to avoid pitfalls. Common virtualization pitfalls are:

1) Overhead: The main concern about the virtualization is the software overhead. During the virtualization process, the hypervisor itself consumes resources to manage the virtualization processes to host virtualized entities on the hypervisor. The overhead can originate due to multiple levels of virtualization implementation, i.e., first level hypervisor, second level, and so on. Levels equivalently correspond to the software abstraction of the infrastructure, whereby each level abstracts the lower level implementation to higher layers. This could impact the overall latency in terms of translating the instructions at each layer, originating from the network applications and terminating at the physical hardware.

2) Isolation: Virtualization should provide a high degree of resource isolation, i.e., the activities of one VNF should not impact the activities of another VNF on the same physical infrastructure. For instance, the cache activities of one VNF should not compromise the cache activities of another VNF. Similarly, the data stored on the file system of one VNF should not be modified by another VNF,

unless granted permission to do so. Poor isolation properties can lead to security vulnerabilities.

3) Migration: One of the key benefits of virtualization is the dynamic migration of the virtualized entities, such as the VNFs. VM migration has been widely studied in the cloud native environment Strunk (2012); however, the VM migration still needs to be explored in the MEC context. The VM migration in the MEC context is much more complex than in the cloud native environment because of resource constraints, both in terms of the target compute resource availability and the connectivity bit rate (i.e., throughput) restrictions. Another challenge of VNF migration is to transfer the hardware context from one physical hardware to another. Pass-through technology for virtualization bypasses the hypervisor for hardware assignment to VNFs. Hence, transferring the pass-through hardware context from one physical infrastructure to another while maintaining same hardware configuration is challenging.

Summary of Enabling Technologies

Hardware abstraction strategies determine the effective utilization of hardware by NF applications. The OS, VM, and container, each view the hardware differently, and hence the hardware interactions of the applications running on these abstraction layers differ, whereby these differences have performance implications. Application on an OS provide the most native performance of hardware due to direct access; however, the OS cannot provide isolation to interference from other applications. Applications on a VM provide the highest degree of isolation due to resource slicing; however, the VM suffers from increased overhead due to memory transactions and resource scheduling. Memory access strategies provide methods to reduce overhead arising from isolation due to memory copies. Zero copy, kernel bypass, and PASID avoid data copies as the data is moved between hardware components and system memory. Additionally, Data Direct I/O (DDIO) (see Sec. **??**) can write the data on the I/O device directly to the CPU cache, effectively reducing the memory hops.

Accelerator task offloading can be achieved in two forms: look aside and in-line. With look aside offloading, the accelerator is considered as an additional I/O device and data is hauled to the accelerator and back to the system memory. This additional step of hauling data results in overhead in the look-aside model. In the in-line model, the data is sent to the acceleration component by the NIC for the acceleration as the data arrives at the NIC for transmission or after reception. The acceleration component is located on the NIC hardware designs; both acceleration device and NIC use a common memory for computations. The downside of the in-line model is that not all data traverses through the NIC, e.g., disk-only data or memory-only data does not traverse the NIC; however, applications that depended on the NIC for data reception and transmission, i.e., packet processing for NFs, can be accelerated through the in-line model.

The application can be notified about the output of the accelerator task in two ways: polling and interrupts. Polling requires the CPU to check the status of the accelerator task completion in every polling period. This periodic checking wastes CPU cycles but allows the CPU to process the accelerator output almost immediately. Polling by many threads, processes, and applications results in a large number polling events, and if there is a common queue that needs to be polled, threads and processes need to be synchronized. The synchronization can be achieved through locking of the queue before reads and writes which can severely impact the overall queue management. In case of interrupts, the CPU may go to sleep states during extended idle periods which incur then core wakeup times. These pitfalls should be carefully considered for the design and implementation of NF applications with accelerator offloading.

Virtualization strategies impact the way I/O devices are utilized by the applications running on various abstraction layers. Hardware assisted virtualization technologies are designed to provide the best performance in case of the SR-IOV and the highest flexibility in case of the S-IOV. However, the complexity of implementation and management of virtualization strategies shadow the flexibility of S-IOV. The NF application design should determine which strategy brings the most benefits for a considered task. Thus, enabling technologies of OSs and hypervisors related to hardware-acceleration directly impact the overall NF designs for flexibility, scalability, and performance.

In this section, we review and report on several recent state of the art proposals and techniques emphasized in the research community related to the OS and hypervisor technologies. The general outline and surveyed literature is illustrated in Fig. 3.48. As highlighted in the enabling technologies section (section **??**), traditional OS design cannot keep up with NF application demands for near line rate processing and minimal overhead. New designs and architectural concepts needs to be envisioned for NFV accelerators incorporating several hardware and software changes that can augment and enhance NF applications. For example, OmniX Silberstein (2017), an accelerator centric OS for omni-programmable systems, presents a programmable OS that extends standard OS abstractions across all system processors (multitudes of CPUs, GPUs, and FPGAs), provides the ability for near-X accelerator units (NXUs) to communicate more effectively, and reduces the programming development complexity. These systems run applications (user code) near-X (X being storage, network, memory). The main objective of OmniX OS is to efficiently execute user applications on omni-programmable systems, i.e., systems with several independent hardware NXUs, at low development cost. This type of extensible OS can scale well with increasing NF application performance demands due to the fact that inter-NXU communication is off-loaded away from the CPUs in centralized OS architectures. Furthermore, energy consumption under virtualized OSes of portable devices is a challenge. For example, virtualized clusters of devices with FreeRTOS network stacks without any hardware optimized acceleration units quickly consume energy since more CPU cycles and performance degradation due to larger overhead causes more energy consumption leading to lower device life span. Therefore, hardware assistance and acceleration for such systems is imperative to drive down CPU instructions per service Batmaz and Doğan (2019).

Different designs and technological innovations in resource management and access (memory, I/O, network, storage, compute), and virtualization (hypervisors, virtual machines, containers), and the unison of both approaches are surveyed in the following sections that highlight the importance of future OS and hypervisor designs and implementations that conform to NF application requirements.

Virtualization Approach

Hypervisors One of the cornerstones of server virtualization is the hypervisor (also known as the virtual machine monitor, VMM). The hypervisor is used to abstract the hardware resources from tenants or VMs within a computing platform (servers, clusters, data centers). Several surveys have addressed the notion of hypervisors and accompanying classification/taxonomy Desai *et al.* (2013), SDN Blenk *et al.* (2015a), and monitoring, analytics, and security Bauman *et al.* (2015). In this section, we provide a brief literature survey on recent advancements on acceleration techniques used in association with hypervisor based computing platforms with emphasis on NF application.

NFV and NFI deployment typically struggles in achieving line rate processing and throughput due to large overhead on the NIC and general network I/O performance. Nakajima et al. Nakajima *et al.* (2015) presents a virtualized NIC (vNIC) framework that aims to accelerate I/O performance and enable low overhead for hypervisor based virtualization with DPDK enabled vSwitch. vNIC leverages and extends the virtio-net framework Russell (2008) that uses a shared memory space for communication between DPDK based NFV application and a DPDK based vSwitch. Nakajima et al. evaluates their implementation using the DPDK benchmark application and achieves 122.9 Gbps throughput and over 14.2 massively parallel processors (MPPS) I/O processing resulting in high performance network I/O.

Efficiently allocating and distributing network resources accurately using SDN and NFV for large scale virtualized environments is a difficult challenge. Each tenant in the NFI can receive its own virtual SDN network managed through the network virtualization layer (also known as the hypervisor) allowing tenants to bring their own controllers suited for their application needs. Similar to the controller placement problem in regular SDNs, for virtual SDNs, the placement of the hypervisor that hosts the SDN for tenants is investigated in Blenk *et al.* (2015b). Four latency performance optimization metrics in the hypervisor placement problem is provided and two metrics are used to quantify the performance per virtual SDN network. Simulations are used to evaluate the proposal and results show that the hypervisor placement can have a large impact on the latency values when optimizing on different latency metrics while numbers and locations of virtual SDNs can affect the results in terms of hypervisor placement and resulting latency values. The main idea and proposal in Blenk *et al.* (2015b) is extended in Blenk *et al.* (2016) to include multi-layer controllers and different hypervisor architectures for the evaluating the control plane latencies in NFV environments.

In terms of 5G cellular environments, Cheng et al. Chang *et al.* (2018) presents and evaluates Open5GCore which is a SDN-enabled Evolved Packet Core (EPC) using both NFV approaches to virtualized instances, *i*) hypervisor (e.g., KVM), and *ii*) container (e.g., Docker). Three applications are executed using Open5GCore (VoIP, Video, and FTP) and metrics under network performance (throughput), memory usage, and CPU usage are used to compare and contrast the virtualization approaches deployed. Their results show that both hypervisor and container approaches increase network delay and can cause disruptions under high volume data traffic, containers use less compute resources than the hypervisor approach, and similarly, the boot time, recovery time, and reboot time for the container approach are faster than the hypervisor approach. The main idea is that the container approach is generally more appealing to processes and applications that do not require modification to kernel space in the computing platform. Similarly, the authors in Ricart-Sanchez *et al.* (2020) presents a hardware based virtualization framework based on data plane programmability for 5G networks. More specifically, resource allocation and QoS differentiated services is implemented using hardware based traffic classification, priority classification, and traffic scheduling. The framework is implemented on FPGAs and the proposed approach is tested. The results indicates improved QoS-aware (packet loss, jitter, latency) network slicing at the data plane.

While metrics such as performance and security is imperative to NFV applications and technologies, availability including fault tolerance in the case of severe disruptions for mission critical applications is another aspect of OS and hypervisor research that was thoroughly investigated and presented in Bressoud and Schneider (1995). More specifically, Bressoud and Schneider (1995) presents the practicality of using a hypervisor approach to replicate primary VMs with backup VMs using a replica-coordination protocol. Such protocols are designed with two important performance goals in mind, namely how epoch lengths and interrupt delays affect system performance. Typically, virtualization and software handling of various system tasks incurs a performance cost and therefore needs to be optimized and/or accelerated using off-chip hardware that can augment the process by off-loading some tasks in NFV applications.

In addition to performance degradation due to the virtualization environment impact, security and privacy concerns are raised when resources are shared under a converged hardware computing platform. One of the early research into hypervisor security architectures is in Sailer *et al.* (2005). The authors present sHype that uses a security reference monitor interface to enforce information flow constraints between VMs and leverages the isolation of VMs to contain security breaches. However, security breaches where VMs can be compromised without detection by the hypervisor is an appealing form of attack since hypervisors do not necessarily include large security technologies (e.g., virus scanners, firewalls, etc.) due to their large performance overhead in high performance carrier-grade QoS and QoE services. Therefore, Trusted Computing (TC) can be used to provide integrity verification (or remote attestation) based on trusted and verifiable measurements and calculations for NFV computing platforms. A hypervisor based TC scheme is implemented in Lauer and Kuntze (2016) that efficiently applies TC to virtual environments. Trusted Platform Module (TPM), an external chip attached on the motherboard of the computing platform is presented and used to collect software (OS, device drivers, etc.) measurements by communicating with the hypervisor before the instantiation of the VM as a process.

In contrast to TPM based integrity checks which targets VM protection, reliably assuring that the hypervisor is continuously protected is a challenging problem that is investigated in Wang and Jiang (2010). The authors present HyperSafe, a bare-metal hypervisor-based low cost approach to provide lifetime supported control-flow code integrity even under the assumption that the hypervisor is untrustworthy. The authors proposes two key techniques that protects the hypervisor from hypervisor attacks (e.g., VM escape and hypervisor rootkits). The first is non-bypassable memory lockdown where a memory page is locked down (i.e., enforced read-only attribute) and can only be modified after unlocking the page under forced supervision using the *WP* bit (write protect) used in *x*86 CPUs. While this can prevent malicious code, it can also prevent benign code updates. The second technique is restricted pointer indexing where the software's runtime execution path is monitored and enforced in target tables (tables where control data is replaced with a restricted index) preventing attackers from arbitrarily controlling the code execution. A proof of concept for HyperSafe is evaluated by measuring runtime overhead with standard benchmarking software (e.g., LMbench, UnixBench, etc.) with and without the HyperSafe protection. Overall, HyperSafe introduced about 5% added overhead but can lead to better cache utilization improvement in situations where multiple target tables and the use of unnecessary memory reads when performing control flow transfers. Similarly, the work in Szefer and Lee (2012) presents HyperWall, a hypervisor-secure virtualization architecture that enables security to guest VMs from an untrusted or even compromised hypervisor. HyperWall uses the hardware resources, that once allocated are referenced and managed by hardware in the Confidentiality and Integrity Protection (CIP) table which protects VMs from unwanted or direct memory access. Another proposed hypervisor that provides lifetime kernel code integrity is presented in Seshadri *et al.* (2007). They propose SecVisor, a low memory footprint hypervisor that ensures integrity verification for OS kernels. SecVisor enforce integrity checks by only allowing user approved code to execute in kernel space.

With increasing security concerns in cloud based virtualized environments, and potentially large attack surface, a complete characterization of the hypervisor vulnerabilities is given in Perez-Botero *et al.* (2013) to provide analysis, recommendation, and pitfalls to designers and architects of NFI. Three main classification metrics are used to designate a vulnerability, the affected hypervisor functionality, the trigger source, and the affected target by the security breach. These classification metrics are then used to show potential attack paths, understand existing attacks, assist in focusing defenses where needed, and discover potential new attacks.

Energy monitoring and power management in large scale virtualized environments is a challenging problem due to both the number of users sharing the physical system and having multiple virtual OSes which are agnostic to the underlying hardware. While methods that inspect internal data structures of the lowest-level belonging to the physical computing platform (the privileged hypervisor and host device driver) which have complete control over the hardware and associated energy consumption provide energy management, they provide course grained per-VM information. Additionally, the host OS level has no knowledge of application level energy consumption. To this end, a hypervisor based power management framework is proposed in Stoess *et al.* (2007) enabling enforcement of power and thermal limits for energy aware/unaware guest OSes and applications running on top of multi-layered distributed virtualized environment. In particular, the framework uses two main energy management subsystems, *i*) host level, and *ii*) guest level management subsystems. The host level provides course grained information on allotted and consumed energy but provides both global or direct and "hidden" energy consumption of VMs and enforces the power allotted to each VM. The guest level provides fine grained energy information to applications by regulating the power allotted to virtual devices ensuring they do not use more than the given energy budget. The proposed method is evaluated against an external data acquisition tool (DAQ) to demonstrate the effectiveness and precision of regulating power to individual physical and virtual devices for energy aware or unaware guest OSes.

Containers While hypervisor based NFV architectures offer a wide variety of services with full fledged OS stacks, less complex applications that utilize such an approach usually have large operating cost and an unnecessary overhead associated with VM deployment times. Containers are introduced to provide low memory footprint for containerized applications using supported virtual memory for isolation but share some of the underlying OS features. Highlighting and demonstrating the main performance differences between container based NFV and hypervisor based NFV, Cziva et al. Cziva *et al.* (2015) presents a open container based NFV framework that utilize SDN based OpenFlows enabled switches to deploy NF applications and services. The framework is designed with a northbound API where NF applications can be instantiated with different services ranging from traffic steering and workload balancing functions. The framework's NFI servers uses an agent (daemon), GLANF Agent, to instantiate and manage the NFs running on the commodity servers. The use of Docker container based NFV architecture improves the NF instantiation by up to 68% and achieves sub-milliseconds latency times in data center networks. Moreover, further evaluations of the framework in data center networks show a throughput improvement as NF service chaining scale grows.

Similarly, Cziva et al. extends Cziva *et al.* (2015) in Cziva and Pezaros (2017) to present Glasgow Network Functions (GNF) architecture which provides generic lightweight Linux containers to host NF applications and services in network edge infrastructure closer to user equipment. GNFs offers support for roaming and mobility that provides consistent and location transparent services. Additionally, transparent traffic steering through provider edge and core networks without disruptions to current NFs is implemented. The GNF architecture is composed of the service, orchestration, NFI management, and infrastructure planes. At the service plane, users can setup and manage policies through a UI or GNF manager API. According to the policies, different virtual NFs are forwarded towards the GNF manager at the orchestration plane. The manager (ETSI's MANO architecture) provides a set of REST APIs to issue instructions to the virtual NFs according to geographical locations of users. Hence, the GNF manager has network wide locations of all virtual NFs and load statistics from edge devices to optimize deployment/placement and enhance utilization. At the virtual infrastructure management level, the edge devices, NFI servers, and the SDN controllers are deployed. GNF Agents or daemons are deployed within the servers at this level to insert OpenFlow rules and migrate containers if necessary. The NFI management layer is similar to the standard SDN control plane. The last layer is the infrastructure plane where all the physical network devices are deployed. This layer acts as the data plane in the SDN paradigm. This proof of concept is evaluated using case studies using three applications (IoT DDoS mitigation, Distributed, on demand monitoring and diagnosing, and roaming network functions).

Traditional NFV platforms are usually formed using either VM or container based approaches that provides a trade-off between security and performance respectively. Kata Container architecture is introduced to utilize the security features of VM based hypervisor approach and the performance benefit of containers. Moreover, the Kata Container is Open Container Initiative (OCI) complainant which indicates that it can be integrated into industry standards easily. A qualitative comparison between Docker, the de facto model for containers, *runc* layer and the Kata Container architecture's *Kata-runtime* layer is investigated in Randazzo and Tinnirello (2019). The qualitative comparison uses metrics such as live migration, security model, resource constraint management, boot time, isolation, integration among others to illustrate the differences in operation between *runc* and *Kata-runtime* components. Kata containers still lack in live migration operation while being significantly better in isolation and security features.

A large trend of NFV research is into replacing VM based hypervisor approaches with cloud native containers that support agile development and uses the "single concern" principle that each container has one responsibility and must execute that responsibility well. However, containers usually have drawbacks related to security concerns of sharing the same OS. Therefore, containers are usually isolated, i.e., not allowed to execute any code in the kernel space. This container isolation and OS platform sharing limits applications that require kernel customization and prevents developers from tuning and optimizing the OS for their applications. X-containers Shen *et al.* (2019) is a library OS (*LibOS*) based container architecture, that mitigates the container isolation issue without requiring hardware assisted virtualization support, and offers a new security paradigm for isolating the X-containers.

The usage of the single concern principle implies a large number of containers in a single commodity server running numerous tasks. Therefore, as the container number grows, so does the memory resources needed for each new container image. CNTR (Center) Thalheim *et al.* (2018) is a container architecture introduced to remove unnecessary tools and software that are not required for common use-cases providing performance benefits of lightweight containers and the functionality of heavyweight containers when necessary. More specifically, CNTR combines the light and heavyweight container images using a new nested namespace that is agnostic to the application allowing high degrees of flexibility and ease of integration. The proposed approach has been implemented and tested demonstrating reasonable image compression by an average of 66% with minor performance overhead.

Paravirtualization System calls by VMs and applications running on top can be expensive and difficult to execute in virtual compared to native environments. Paravirtualization is a virtualization approach that inserts a software layer between the VM's guest OS and hypervisor enabling the VM to become aware of the hypervisor and communicate directly with it. This in turn improves performance compared to software assisted full virtualization at the cost of modification to guest OSes. In essence, paravirtualization selectively emulates the hardware devices that a virtualized OS needs which in turn can minimize overhead similar to the Container virtualization approach's goal Vaughan-Nichols (2006).

High Performance Computing (HPC) environments where the virtualization overhead is highly critical to applications involved in HPC systems is an attractive environment where paravirtualization can be utilized to minimize overhead and optimize performance. An empirical evaluation on the efficacy of using paravirtualization of Xen hypervisors in HPC environments is investigated in Youseff *et al.* (2006). Specifically, a series of comparison runs are executed and results are collected across several benchmark tests that comprehensively analyzes the performance implications of using HPC with and without paravirtualization. The tests are classified as micro-benchmarks, macro-benchmarks, and real HPC applications. In particular, for micro-benchmarks, evaluations of specific machine components and protocols such as Message Passing Interface (MPI) based network bandwidth and latency, CPU, memory, and disk I/O stress tests on and between HPC cluster nodes. For macro-benchmarks, full system performance tests are used that evaluate the efficiency of HPC in handling critical operations (e.g., computational fluid dynamics). Finally, for HPC applications, an oceanographic and climatology general circulation model to simulate a year's worth of oceanographic and climatology data at one second resolution. The comparative results found indicate that Xen paravirtualization does not impose significant overhead compared to non-virtualized Linux HPC systems.

Typical Paravirtualization systems directly loads the hypervisor on the underlying hardware (bare-metal hypervisors) that suffers from a single point of failure if the hypervisor fails. Significant research is being directed towards adding and automating the failover protection in the event a hypervisor or underlying hardware fails. Landis et al. Landis *et al.* (2011) proposes a paravirtualization model that addresses this limitation by logically or virtually partitioning the host system with special infrastructure partitions that controls resource management and I/O device drivers. The tracking application "ultravisor" within the special infrastructure partition is used to monitor and isolate resources used by guest partitions. Moreover, different infrastructure partitions in different systems can communicate and share hardware resources creating a virtualized datacenter. Each individual system can have multiple failover infrastructure partitions to protect the virtualization process in case of failure. Improving performance using SR-IOV on a paravirtualized system is not easily accomplished since no standard way exists to leverage the benefits provided by the virtualization I/O functions. Solomon et al. Solomon and Hoglund (2012) proposes a specialized driver that is coupled with the paravirtualization driver to implement I/O virtualization data transmission for guest OSes.

Using paravirtualization (or full virtualization) on real-time hypervisors on multi-core systems is a challenging problem due to the added overhead. Hardware assistance for virtualization is used to streamline the deployment process. However, several platforms do not feature hardware assisted virtualization and therefore cannot be used. To address this limitation, Gilles et al. Gilles *et al.* (2013) presents Proteus real time bare-metal hypervisor that targets multi-core platforms for both paravirtualization and full virtualization without the use of hardware assistance as shown in Fig. 3.49. Proteus ensure spatial and temporal separation of guess VMs. Hypercalls are used when paravirtualized guest OSes are used. Proteus is evaluated in terms of memory footprint, interrupt latencies, emulation routines, hypercall execution, and hypervisor context switches.

I/O Virtualization

HW I/O Virtualization While full virtualization allows large computing platform clusters to share and better utilize resources, it can severely degrade the processing performance in softwarized I/O device drivers such as virtual NICs and other virtualized PCIe devices, that limit scalability of the overall system. Similarly, using paravirtualized I/O devices (e.g., direct device assignment or "passthrough I/O"), while better than full virtualization in terms of performance for I/O intensive workloads, introduce scalability limitations and that limits their viability for applications in carrier grade environments.

Off loading and optimization schemes are heavily researched to adapt techniques, e.g., SR-IOV and self-assisted devices Raj and Schwan (2007), in an attempt to mitigate software virtualization overhead by incorporating dedicated hardware assistance. However, I/O interposition in full virtualization stacks is needed for critical applications such as live migration, VM replication, and dynamic load balancing among others that hardware assisted virtualization (paravirtualized I/O) approach prevents Har'El *et al.* (2013). In such cases, hardware assisted virtualization has significantly limited use. Therefore, solutions involving purely hardware assistance essentially evolve to becoming hardware dependent limiting their applications (e.g., live migration, etc) in high end virtualization environments. Some commercial closed source virtualization approaches introduce *composability*, where simple frequent I/O operations are run on hardware while more complex operations involving hypervisor I/O interposition on guest VMs can be emulated in software (e.g., Intel Scalable IOV). This allows flexible operation of hardware assisted virtualization which address the limitations involved in direct device assignment (i.e., VMs communicate directly with I/O devices) and scales well for modern cloud computing environments. Similar to Intel's scalable IOV, Nakajima et al. Nakajima *et al.* (2011) presents a hybrid virtualization approach of taking advantage of both hardware assisted virtualization and paravirtualized I/O devices. The proposed idea is to run a paravirtualized guest on a hardware assisted VM maximizing resource utilization and hardware capabilities as shown in Fig. 3.50. Their experiments demonstrated that the hybrid approach improves paravirtualized systems by approximately 30% in memory intensive tests and 50% in micro-benchmarks while showing about 16% improvement in I/O intensive workloads than pure hardware assisted virtualization solutions.

Focusing on the virtualization overhead incurred by network I/O devices, dong et al. Dong *et al.* (2011) presents a software based I/O optimization approach that leverages both multi-core processors using virtual Receive Side Scaling (RSS) and aggregating expensive virtual network I/O interrupts using an adaptive multi-layer interrupt coalescing optimization scheme. For the coalescing or aggregating interrupt approach, two implementations are presented. *i*) The frontend virtual interrupt coalescing (FVIC), and *ii*) backend virtual interrupt coalescing (BVIC). The FVIC controls the frequency virtual interrupts within the guest OSes by generating a periodic timer that polls arriving packets in shared rings (buffers). In contrast,the BVIC operates in domain 0 (privileged domain in Xen hypervisors) and delays interrupts by controlling their delivery from domain 0. FVIC is determined to have better performance since BVIC incurs more context switching than FVIC and requires an extra timer in domain 0. The multi-layer (physical and virtual interface layers) interrupt coalescing approach is an optimization scheme used to coordinate interrupt coalescing policies between the physical to virtual (i.e., domain 0 to the guest domain) with the goal of minimizing latency and maximizing performance. It is adaptive since it modifies the design parameters of the scheme to adjust the cost per interrupt through a coalescing manager running in domain 0. Finally, the last improvement is efficiently using multi-core processors using RSS to effectively load balance incoming packets to different CPUs so that one CPU does not get saturated while others are idled. These enhancements are tested on Xen based paravirtualized hypervisors to demonstrate the performance overhead in terms of CPU utilization and bandwidth.

As the network performance increase (e.g., using 10-Gigabit Ethernet ports), virtualized I/O intensive workloads related to NF applications scale proportionally. The main inhibiting factors in network I/O virtualization are inter-domain communication (I/O interrupts) and extra scheduling overhead (context switching) in the hypervisor. Investigating the impact of hypervisor scheduling and domain 0 on multi-core compute platforms, Liao et al. Liao *et al.* (2008) proposes two optimizations for the default hypervisor scheduler, *i*) a cache-aware scheduler and *ii*) a credit-stealing policy, that aims to improve the capabilities of NFVs to process high speed network traffic. The cache-aware scheduler approach provides the guest OS with improved cache locality (space and/or time) when accessing packets reducing inter-domain communication. The credit-stealing policy is an approach that scheduled virtual CPUs that process I/O requests (referred to as I/O VCPUs) in favor of domain 0 which results in reduces the latency in servicing I/O interrupt requests. The virtualized I/O optimizations are implemented and evaluated on Linux showing an improvement of 96% servicing I/O interrupts over the traditional Linux hypervisor and packet processing improvement with savings of 36% in core utilization per gigabit.

SR-IOV SR-IOV is used for multiplexing a single physical PCIe device into multiple virtual PCIe devices at the hardware level (i.e., without virtualization intermediary) allowing for scalability and increased utilization without affecting CPU utilization. In particular, for virtualized environments, NFV and NFI platforms introduces significant performance requirements for I/O PCIe devices (e.g., NIC) that require virtualization optimization in order to maximize I/O utilization and reduce resource contention. In contrast to traditional network devices where software implementations are highly optimized for the custom hardware platform, NFV architectural implementations in OS and hypervisors systems adds significant overhead for NF applications (e.g., Deep Packet Inspection, DPI) in terms of carrier-grade I/O packet processing. Several new and innovative designs using SR-IOV enabled devices with DPDK demonstrated in Kourtis *et al.* (2015) shows that the approach of combining SR-IOV with DPDK achieves significantly higher throughput than traditional Linux based network stacks targeting VNF deployments. An open source library, nDPI, is used to implement the proposed approach, and the approach is tested on an experimental setup comparing both DPDK and LibPCAP versions of DPI in physical and virtual environments. The results indicate a promising improvement of the virtualized DPI solution and a substantial improvement with DPDK.

While throughput is increasingly important for high performance I/O operations in large capacity networks, controlling latency and packet jitter with real time network traffic is needed to ensure optimal or satisfiable performance. Challa et al. Challa (2012) presents a study on various I/O virtualization technologies including SR-IOV that delivers near native I/O performance. In particular, the main advantages outlined for SR-IOV are, 1) provides near native performance through hardware assisted virtualization, 2) provides robust isolation improving security between various NFs, 3) can extend to virtualized adapters and existing I/O protocols with ease allowing easy integration into modern data centers, and 4) provides higher degree of flexibility and management for administrators. Main disadvantage, similar to the issues with hardware assisted virtualization, is that it cannot be used in VM migration applications and similar NF applications.

Memory Access and Management

IOMMU IOMMU connects a DMA capable I/O bus to the system's main memory allowing device drivers connected to PCIe components to map devices' virtual addresses to physical addresses enabling security, isolation, and memory protection in the event of faulty or malicious devices. While this technology provides benefits to virtualized systems using DMA Remapper (DMAR) and Interrupt Remapper (IR), the remapping process adds address resolution and validation overhead to all I/O requests. For network I/O applications, mapping translations are more frequent and can bottleneck system performance. The interaction between an IOMMU, CPU, and the hypervisor is shown in Fig. 3.52. Unlike a processor's virtualization and translation activity which can be intercepted in flight and restarted (e.g., via page faults), I/O virtualization activity generally is difficult to intercept in flight and restarted.

The IOMMU is used to control and manage guest tenants interfacing with device drivers for hosts that use typically direct device assignment (gust VM directly interacts with the I/O device without host intervention). However, several issues are raised with these MMUs including *i*) requires pinning all guest pages, and *ii*) it exposes the guest memory to failures in the device drivers. An efficient IOMMU, virtualized IOMMU (vIOMMU) Amit *et al.* (2011), is designed, implemented, and tested to solve these issues thereby allowing guest tenants and applications to make full use of I/O device drivers (e.g., NICs for NFV applications). Similarly, Amit et al. Amit *et al.* (2010) illustrates the potential impact of the IOTLB (I/O TLB) used in IOMMUs and presents enhancements that mitigates cache misses and can accelerate address resolution rates. In particular, each IOTLB cache miss results in high latency since it requires physical address resolution performed by a page walk using a DMAR. The proposed approach uses a software based virtualization layer (using vIOMMU) that reduces IOTLB miss rates.

NUMA NUMA architecture allows processors in multi-core systems to access shared memory simultaneously by coupling each NUMA node (socket with several cores) with a local memory space (including local memory controllers) avoiding the performance hit of having processors waiting for access to the shared memory space. However, coupling NUMA architecture with NFV environments introduces performance penalties for packet processing workloads since cache locality is not guaranteed. Investigating NUMA architecture performance and the cache exhaustion impact in NFV systems, Sieber et al. **?** presents an evaluation criteria, Network Efficiency Index (NEI), that is used to indicate the efficiency in running a VNF in multi-core NUMA based systems with emphasis on the packet handling application and in conjunction with DPDK. Several performance tests are taken with core utilization, cache hit ratio, and socket memory throughput. The results from experimentation shows that copying packets between NUMA sockets increases the CPU utilization resulting in penalties that should be avoided. The NEI, the ratio between the packets accessed by the VNF from the Last level Cache (LLC) and the total number of packets received by the VNF, is used to optimize the cost of placement of VNFs in terms of NUMA performance and cache overhead. The goal is to optimally distribute packets per VNF with respect to parameters such as NIC queue affinity, core affinity, etc. as shown in Fig. 3.53. With increasing network capacities, the event of packets being copied across NUMA nodes (using socket interconnects) increases as the scale of the virtualized environment increases resulting in more cache exhaustion and more cache misses.

Similarly, mitigating the impact of cross NUMA data exchange (or remote memory access), Wang et al. Wang (2017) investigates NUMA based system performance with DPDK based pipelined NFs. More specifically, a thread placement strategy on physical cores is implemented and tested to reduce the latency of processing packets due to remote socket memory access. The thread placement strategy, Locality First Mapping algorithm, is implemented on a application-level coordinator that assists NFs in maximizing the usage of the underlying hardware and increase data locality. The evaluation on different NFs (flow metering, packet checks, and firewall) shows reduced latency with the proposed algorithm compared to a balanced distribution of threads across the cores. Along the same lines, Lachaize et el. Lachaize *et al.* (2012) presents MemProf, a profiler that allows programmers to choose and implement efficient application level optimization for NUMA based systems. Memprof structures threads and their memory pages building temporal flows of interaction between them. This assists programmers in identifying why and which memory objects are accessed using expensive remote memory accessed providing them with the means of optimizing thread placement and core utilization improving system performance. Moreover, Majo et al. Majo and Gross (2011) presents an analysis on NUMA based multi-core Intel Nehalem Singhal (2008) platform where memory management and process scheduling are coupled and an optimization algorithm is given to maximize data locality and minimizing cache contention. A scheduling algorithm, NUMA-Multicore-Aware Scheduling Scheme (N-MASS), is proposed which optimizes data locality and reduces cache contention. N-MASS extends maximum-local scheduling algorithm that maps processes (threads) to cores improving performance up to 37% and 7% on average. Similarly, Vikranth et al. Vikranth *et al.* (2013) proposes a task stealing scheduling algorithm that dynamically analyzes the hardware interconnection between CPU and memory sockets grouping cores and connections as a logical topology tree. These trees are used to map multiple worker pools, i.e., stealing domains, that restricts task stealing to such domains improving performance. Focusing on NF applications in NUMA based multi-core platforms, Blagodurov et al. Blagodurov *et al.* (2010)

proposes a contention-aware scheduling algorithms for NUMA based systems. The application optimizer schedules threads on cores with the goal of isolating threads belonging to specific classes of NFs (or virtualized functions in general) minimizing contention of resources (memory and core utilization). The proposed approach is compared to similar schedulers in popular application involving Network Attached Storage (NAS) benchmarks such as *Cilk* and *OpenMP* showing on average 1.24 times improvement.

While none virtualized multi-core systems based on NUMA architecture presents added challenges such as resource contention and performance degradation for NIC I/O and memory controllers among others, virtualized systems aggravates the problem even further introducing virtualization (abstraction) overhead and even more resource contention. To this end, Rao et al. Rao *et al.* (2013) proposes a virtualized NUMA-aware vCPU scheduling that dynamically migrates vCPUs to minimize remote memory accesses overhead. In particular, a Bias Random vCPU Migration (BRM) is proposed to schedule vCPUs based on hardware metric latency (aggregated measurement of several hardware memory access events) improving CPU utilization and reducing latency. The proposed algorithm is compared to Xen credit scheduler demonstrating a 31.7% improvement.

Summary on OS and Hypervisor Research Studies

Different design approaches to OSes and hypervisors have a direct impact on NF applications in terms performance, flexibility, and scalability. Three main approaches to virtualized platforms hosting NFs are hypervisors (hosted or bare metal), containers, and paravirtualization with or without hardware assistance. While full virtualized hypervisors provide a very high degree of flexibility, scalability and performance are affected. In contrast, containers provide increased scalability but reduced flexibility and performance. Paravirtualization is the approach for high performance since some native OS functionality is added to the virtualized process, but suffers when it comes to scalability and flexibility. Generally, each virtualization approach targets a specific application or deployment environment that serves an application's Key Performance Index (KPI) metrics.

NFs are I/O and memory intensive applications that require near line rate processing to prevent bottlenecks in the virtualization stack especially in carrier grade environments such as MEC nodes and eMBB enhancements in 5G environments. Overhead associated with virtualization scales with network performance. More specifically, network I/O virtualization becomes a serious bottleneck that off loading and optimization techniques alone are not enough to circumvent these problems.

Efficient memory access and management in HPC platforms hosting carrier-grade NFs are difficult since many aspects of hardware resource management, e.g., I/O devices, caches, local/remote memory, thread/CPU mapping and scheduling, strongly affects the performance. Moreover, utilizing NUMA architecture for modular socket design grouping compute, local storage, and network resources compounds the issues faced when optimizing NF performance.

3.9 Overall Summary and Discussions on Operating Systems and Hypervisor

Operating Systems

One of the critical aspects of OS that impact the NF applications is the scheduling strategies of the applications in a multi-core environment. Application designers utilize multi-threaded approach to exploit the multi-core resources, whereas the inter-thread communications result in thread synchronization problems such as lock-unlock during common queue read/write. Message passing and queue management challenges are in addition to the core-to-core interconnect saturation in the hardware. Therefore, NF application designers must consider the pitfall of multi-threaded approach and underlying scheduling policies of the OS.

OS management of platform resources also includes the power and performance measurements and control strategies, which includes methods to transitions of core performance (P-states) and power states (C-states). Specifically for NF applications pitfall here is that, the cores that are operating at opportunistic higher frequency can result in varying performance behavior for packet processing which might induce jitter in the packet flow. Power saving strategies of OS could have implications on higher latency for the core wake-up from deep sleep states. Hence, for high availability NF applications as well as to operate under strict latency conditions, OS must maintain constant characteristics of frequency and power.

OS Application Optimizations Reconfigurable hardwares such as accelerators and computing hardware Guerrieri *et al.* (2019) provides methods to adapt the hardware characteristics to the applications requirements for improving the overall performance. Pitfall to avoid in such reconfigurable hardware for the management of these resources is to ensure the complex algorithms that are needed to identify the reconfiguration parameters that works best for the application does not consume excessive hardware resources. That is, the optimization algorithm (user-space application) that needs to evaluate reconfiguration parameters would itself use the compute, memory and storage resources of the host hardware.

OS Kernel Optimizations OS kernel effectively manages the hardware resources of the platform with process schedulers (for CPU), memory manager (for DRAM and cache), and device drivers (for disk and I/O). While OS provide the abstraction of hardware resources and facilitate NF applications to share the common hardware resources, mutual interferences of these applications while contending for resource which negatively affects the overall performance is a pitfall that needs to carefully considered. During OS scheduling of NF application processes over physical cores should consider the recently cores with hot cache to increase the probability of cache hits. Other considerations that directly impact the latency of processing are memory handling between user-space and kernel space, and reducing the kernel overhead in terms of managing hardware resources.

Hypervisors

Virtualization Strategies Virtualization provides abstraction procedures for the hardware components with the goal to ensure isolation between software entities that run on the hardware resources. Virtualization inherently generates overhead in the process of resource abstraction and allocation to virtualized entities such as VMs. Overhead could negatively impact overall performance, when resources are oversubscribed and multi-level abstractions are performed. When there are strict application requirements and limited hardware resources, it could be best to avoid virtualization to efficiently manage the hardware resources if a compromise on isolation between VMs are permitted.

Service Migration Service migration involves the movement of softwarized NFs from one hardware platform to another, typically on a virtualized infrastructure (i.e., hypervisor). One of the challenges in the service migration is to preserve the application context in a hypervisor and hardware domain such that the target platform in the migration process can be prepared for minimal disruptions. Service migration is performed to accommodate the NF application deployment in terms of scaling of applications, hardware and software upgrades, user movement etc. Pitfall to avoid in this process to ensure that the Hardware compatibility in terms of meeting latency and throughput requirements, transfer of virtualized entities between source and destination, ensure security and integrity, and finally to ensure safe instantiation of virtualized entities (i.e., VMs) on target virtualized infrastructures (i.e., hypervisors).

Resource Slicing Hardware resource utilization and management is an important aspect of OS and hypervisors. OS manages the hardware resources in best effort manner by sharing the resources, whereas hypervisors tries to enforce strict division of resources through static allocation of hardware and software resources. Therefore, hypervisors perform the logical slicing of resources that can be dedicatedly allocated to VMs. In terms of NF application deployment, virtualization provides an effective method for network slicing and coexistence of multiple VNFs applications on a common platform infrastructure. While virtualization provides flexibility, scalability, and migration capabilities in deployment of VNFs, the shortcomings of virtualization should be considered to avoid the pitfalls by ensuring the integrity of VNFs during scaling and migration. It is important to note that, a poor virtualization could lead to lower resource utilization, high power consumption, and security vulnerabilities.

3.9.1 OPEN CHALLENGES AND FUTURE RESEARCH DIRECTIONS

Application-to-Application QoS SDN based application-to-application stream reservation Virtualization provides an effective method for resource slicing and co-existence of multiple VNFs applications on a common GPC platform with isolation. Although virtualization provides flexibility, scalability, and migration capabilities in deploying the VNFs, the shortcomings of virtualization should be considered carefully to ensure the integrity of VNFs. A poor virtualization could lead to lower resource utilization, high power consumption, and security vulnerabilities

Hardware Based Virtualization When virtualization is needed and when we don't. For specialized execution, and that does not require changes, we don't need virtualization. Like high speed switching, which required dedicated system, fully efficient, and EPC where the implementation changes very often, as they release new specs, etc. desires virtualization. So identifying which application required what is a key part of differentiation. Hybrid networks will still co-exist in the years to come

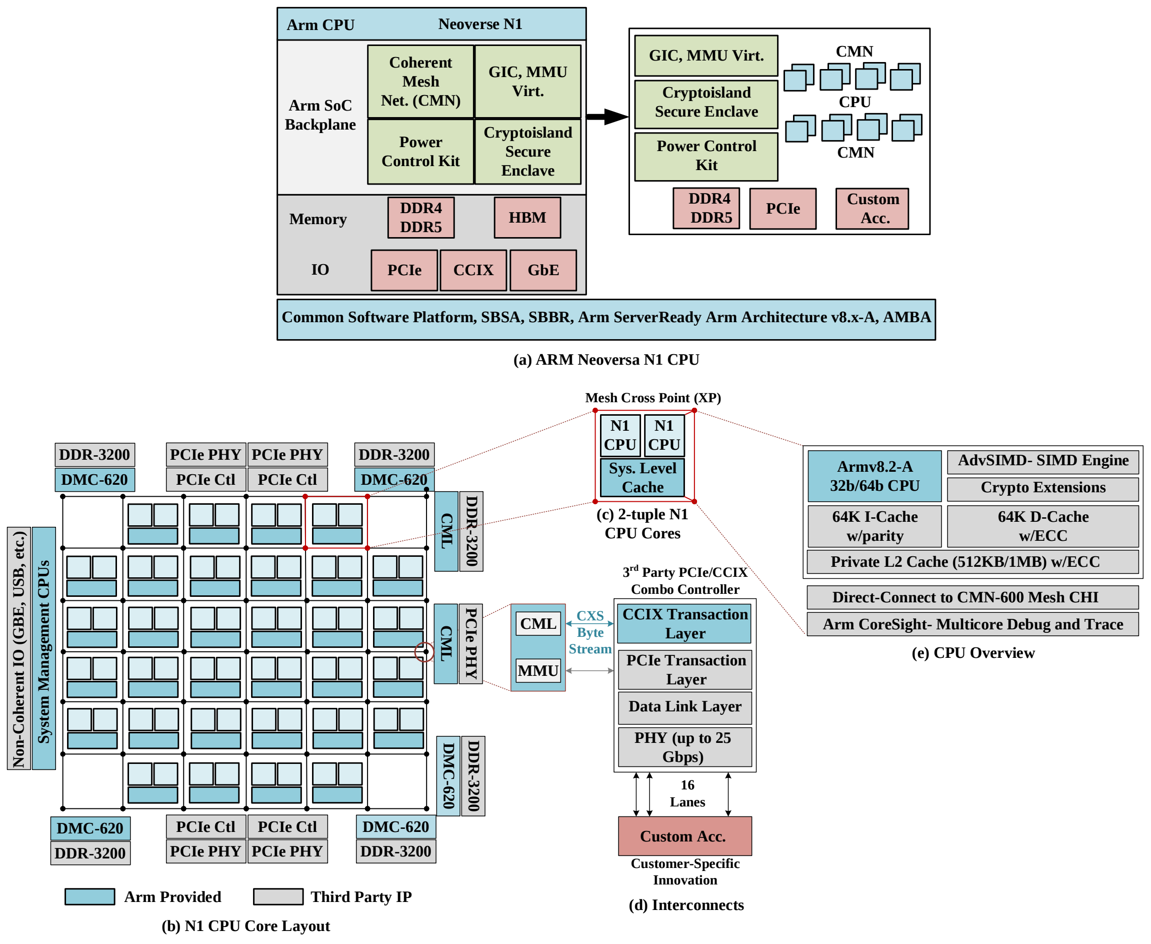


Figure 3.5: Overview of Arm® Nervosa N1 Architecture Pellegrini *et al.* (2020): (A) Illustration Of Arm Cpu Functional Blocks along with Cpu Interconnect, Memory Management Unit (Mmu), Power Management, and Security Components In Relation to Third-party Memory and I/O Components. Nervosa N1 Can Be Extended to Server-scale Deployments with Specifications Of Server Base System Architecture (Sbsa), Server Base Boot Requirements (Sbbr), and Advanced Microcontroller Bus Architecture (Amba) ARM Holdings (2019). Arm Neoverse N1 Cpu Sits on the Arm Soc Backplane (Uncore) along with Coherent Mesh Network (Cmn) And Power Control Kit. Memory and I/O Are Third-party Modules That Interface with Arm Designs Through Interfaces (Green and Blue Blocks Are from Arm, While Brown and Gray Color Blocks Are Third-party Blocks).

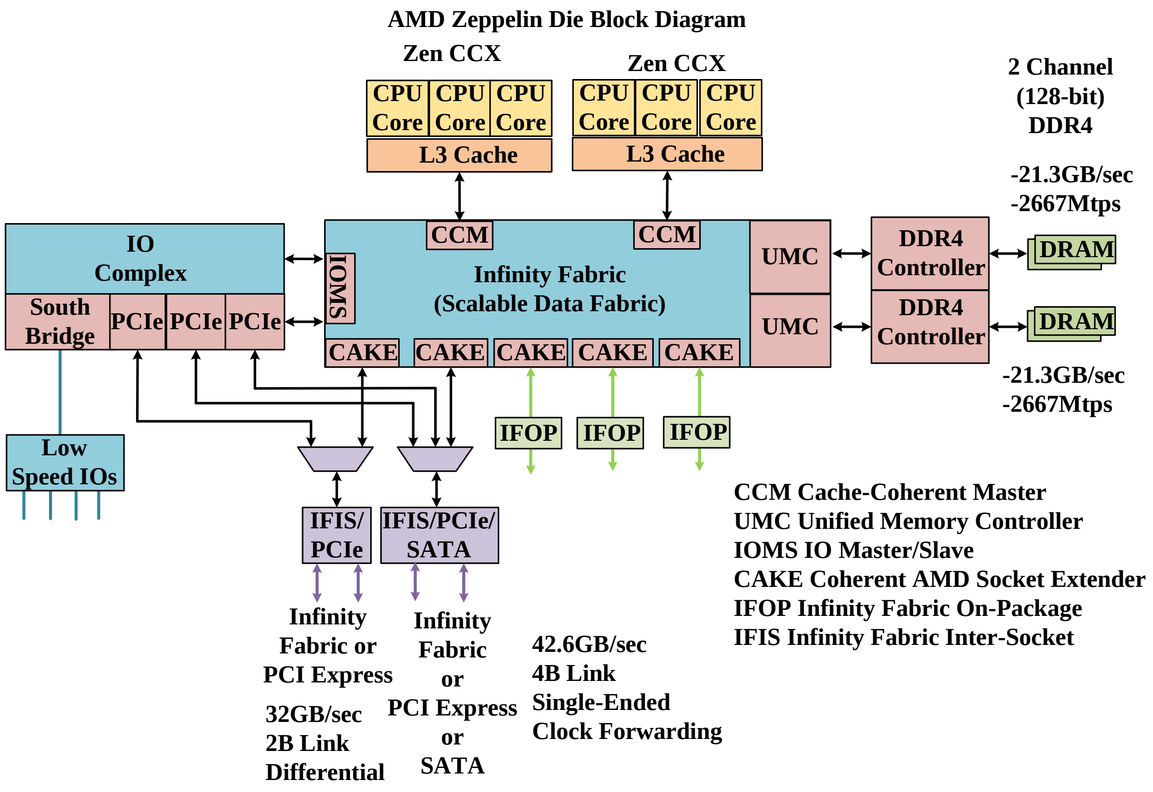


Figure 3.6: Overview of Amd® Zen Core and Infinity Core-to-core Fabric AMD (2019). The Infinity Fabric Defines a Scalable Data Fabric (Sdf) as On-die Core-to-core Interconnect. The Sdf Extends the Connectivity from On-die (On-chip) to Chip-to-chip (I.E., Socket-to-socket) Connectivity Though the Coherent Amd Socket Extender (Cake), Resulting in An Infinity Fabric Inter-socket (Ifis). An Sdf Extension To Connect with Multiple I/O Devices Is Enabled Through an I/O Master Slave Component. Similarly, Cache-coherent Master (Ccm) On The Sdf Directly Connects the Cores (On-die) That Are Associated With the L3 Caches Coherently, While the Unified Memory Controller (Umc) Extends the Connectivity to the Dram.

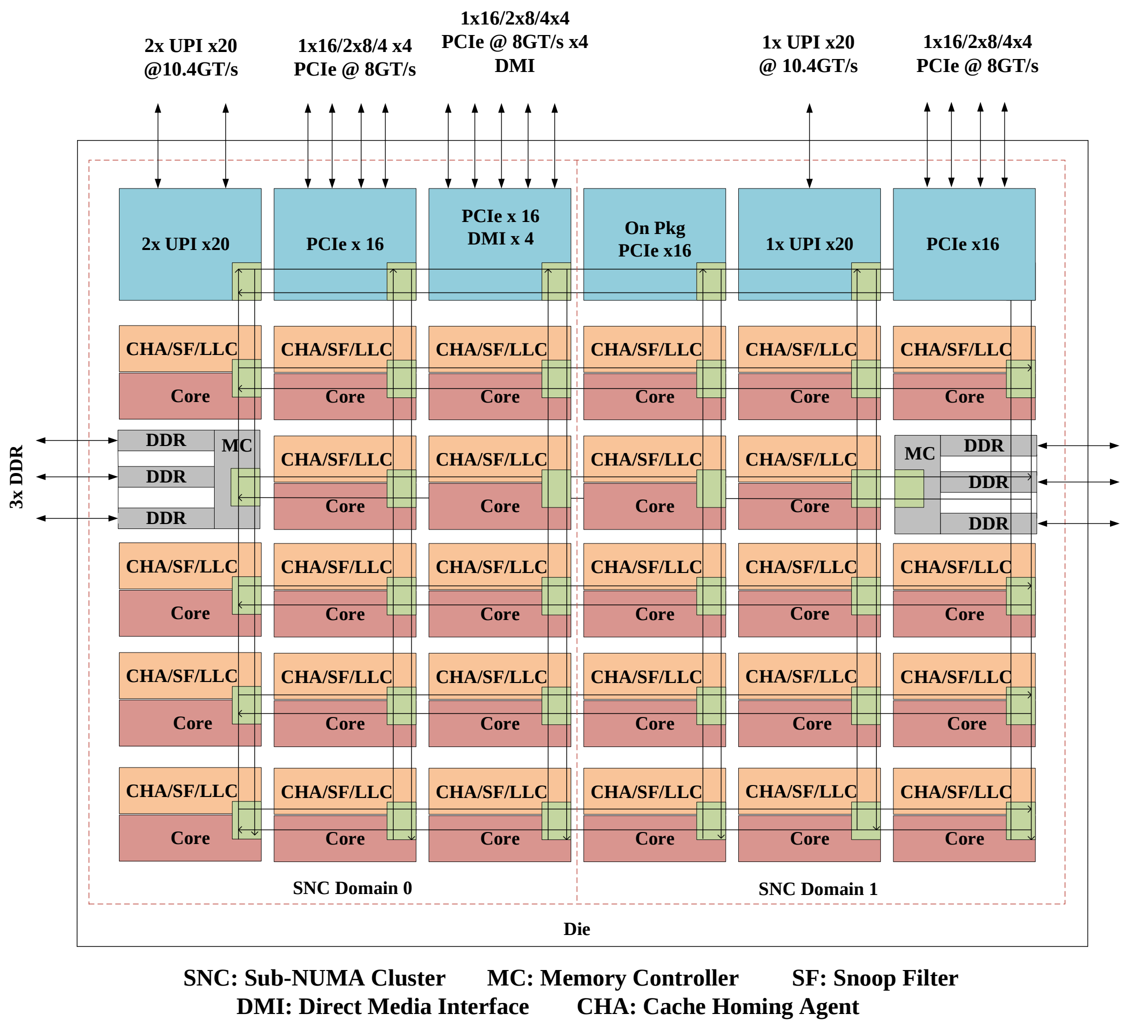


Figure 3.7: Intel® Xeon® Cpu Overview Intel (2019a): The Intel® Xeon Cpu in a Single-socket Package Consisting of Single Die with 22 Cores and 2 Memory Controllers (Mcs) on Either Side of the Die Extending to Ddr Interfaces. The Cores Are Arranged in a Rectangular Grid Supported by a 2d Mesh Interconnect That Connects All Cores Within a Single Socket. Each Core Component Is Interconnected With Uncore Components, Such as Cache and Homing Agent (Cha) to Apply Cache Policies, Snooping Filter (Sf) to Detect Cached Addresses At Multiple Caches to Maintain Coherency, and Last Level Cache (Llc) To Store Data Values.

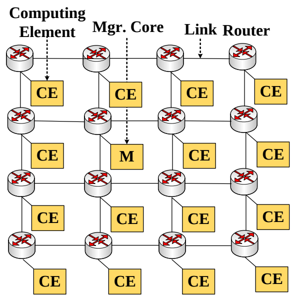


Figure 3.8: Overview of Network on Chip (Noc) Kumar *et al.* (2002) Where Each Compute Element (Ce) Connects to a Router: The Noc Comprises a Fabric of Interconnects That Provides On-chip Communication to Compute and Memory Elements Which Are Connected To Routers. The Noc Provides Homogeneous Connection Services as Opposed To Heterogeneous Interconnects Based on Different Technologies, Such As Ddr and Pcie for On-chip Components. The Noc Fabric Is Extensible and Can Be Easily Scaled as the Number of Compute Elements Increases.

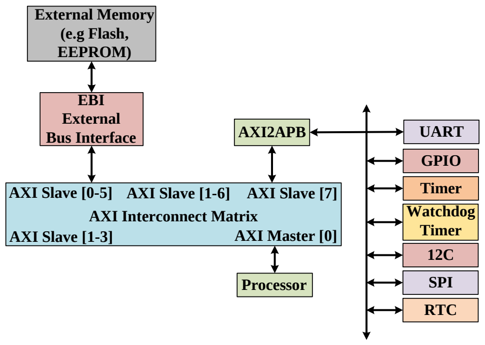


Figure 3.9: Overview of Advanced Extensible Interface (Axi) ARM Holdings (2019): The Axi Provides an On-chip Fabric For Communication Between Components. The Axi Operates in a Master And Slave Model, the Slave Nodes Read and Write Data Between Components As Directed by Master Nodes. The Axi Also Provide Cache Coherency With the Axi-coherency Extension (Ace) Specification ARM Holdings (2019) to Keep the Device Cache Coherent With Cpu Cores.

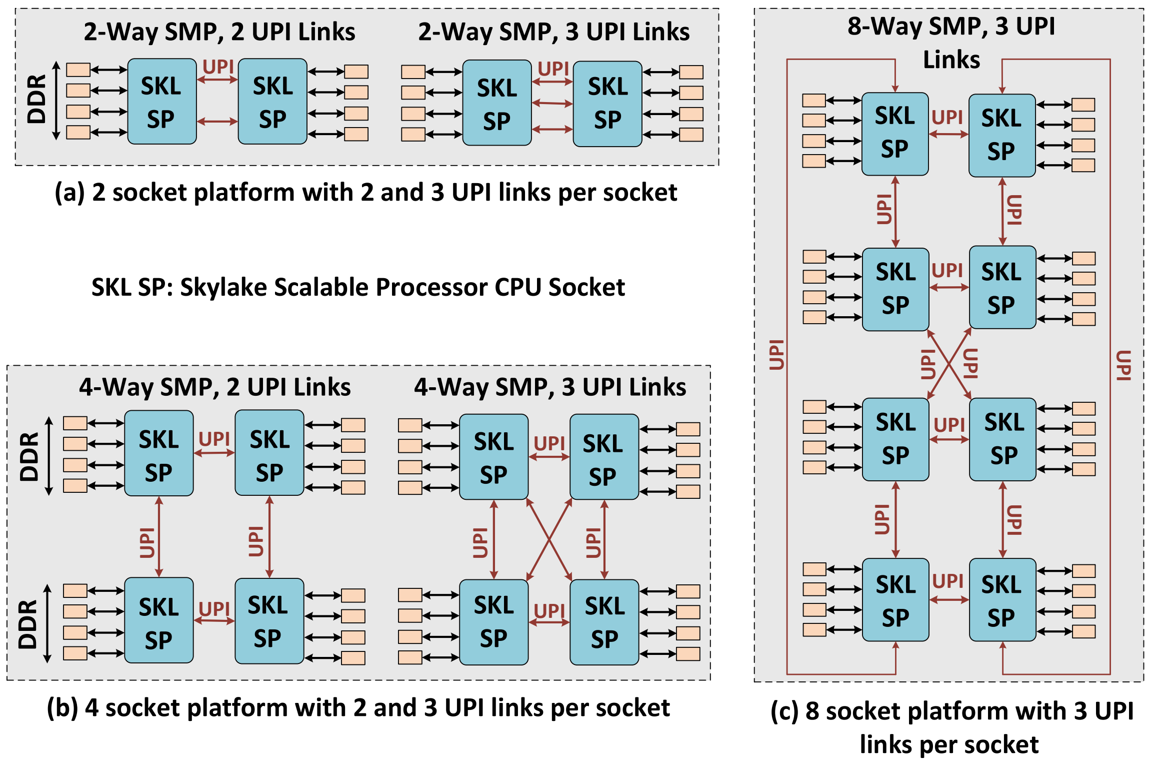

2-Way SMP, 2 UPI Links
2-Way SMP, 3 UPI Links
DDR
UPI
SKL SP
(a) 2 socket platform with 2 and 3 UPI links per socket
SKL SP: Skylake Scalable Processor CPU Socket
4-Way SMP, 2 UPI Links
4-Way SMP, 3 UPI Links
(b) 4 socket platform with 2 and 3 UPI links per socket
8-Way SMP, 3 UPI Links
(c) 8 socket platform with 3 UPI links per socket

Figure 3.10: Overview of Skylake Scalable Performance (Sp) Meng *et al.* (2018); Tam *et al.* (2018) With Intel® Ultra Path Interconnect (Upi): The Upi Is a Point-to-point Processor Interconnect That Enables Socket-to-socket (I.E., Package-to-package) Communication. Thus, with the Upi, a Single Platform Can Employ Multiple Cpu Sockets: (A) 2 Socket Platform Inter-connected by 2 Or 3 Upi Links per Cpu Socket, (B) 4 Socket Platform Interconnected by 2 or 3 Upi Links per Cpu Socket, and (C) 8 Socket Platform Interconnected by 3 Upi Links per Cpu Socket.

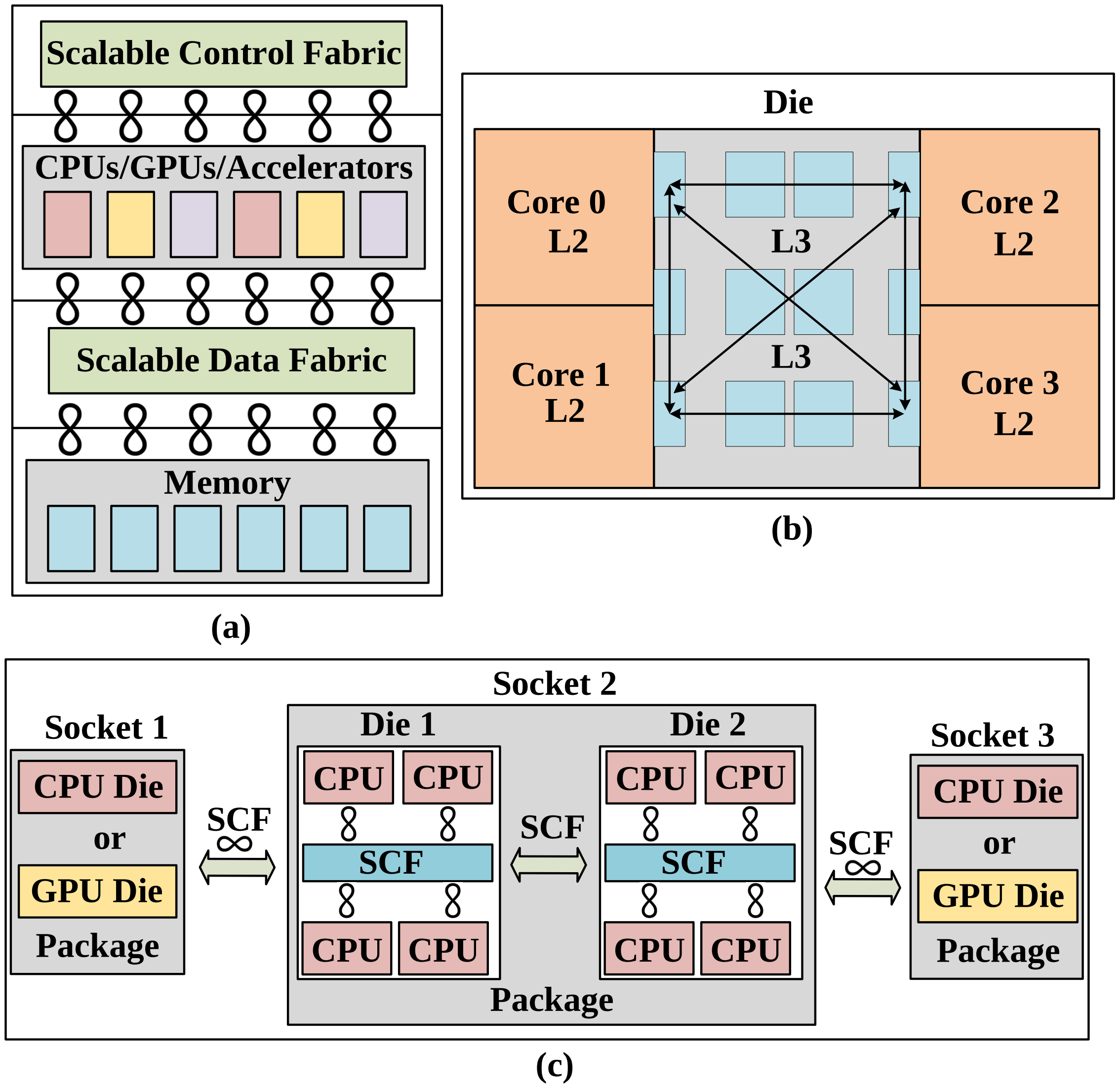


Figure 3.11: Overview of Amd® Infinity Fabric Beck *et al.* (2018); Lepak *et al.* (2017); AMDl (2020); Teich (2017)For On-chip and Chip-to-chip Interconnects An Accelerator Chip: (A) Shows the Overview of Interconnects Between Cpu and Gpu Through the Scalable Control Fabric (Scf), (B) Shows The Interconnects from Core-to-core Within a Die for Relative Comparison, and (C) Shows the Overall Fabric Extensions at The Socket, Package, and Die Levels.

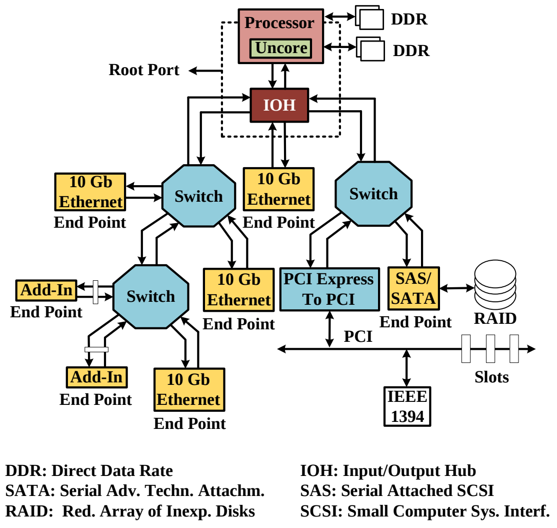


Figure 3.12: Overview of Peripheral Component Interconnect Express (Pcie) McGinnis (2017) Interface Which Is an Extension To Pci Technology: Pci Operated as a Parallel Bus with Limited Throughput Due to Signal Synchronization among the Parallel Buses. The Pcie Implements a Serial Communication per Bus Without Any Synchronization among Parallel Buses, Resulting in Higher Throughput. The Pcie Is a Universal Standard for Core-to-i/O Device Communications. The Pcie Protocol Defines a Point-to-point Link with Transactions to System Memory Reads and Writes by The I/O Devices, Which Are Referred to as “end Points” And Controlled by the Root Port (Rp). The Rp Resides at the Processor As an Uncore Component (See Fig. 3.3). The Pcie Switches Extend a Primary Pcie Bus to Multiple Buses for Connecting Multiple Devices and Route Messages Between Source And Destination. A Pcie Bridge Extends the Bus from Pcie to Pci so As To Accommodate Legacy Pci I/O Devices.

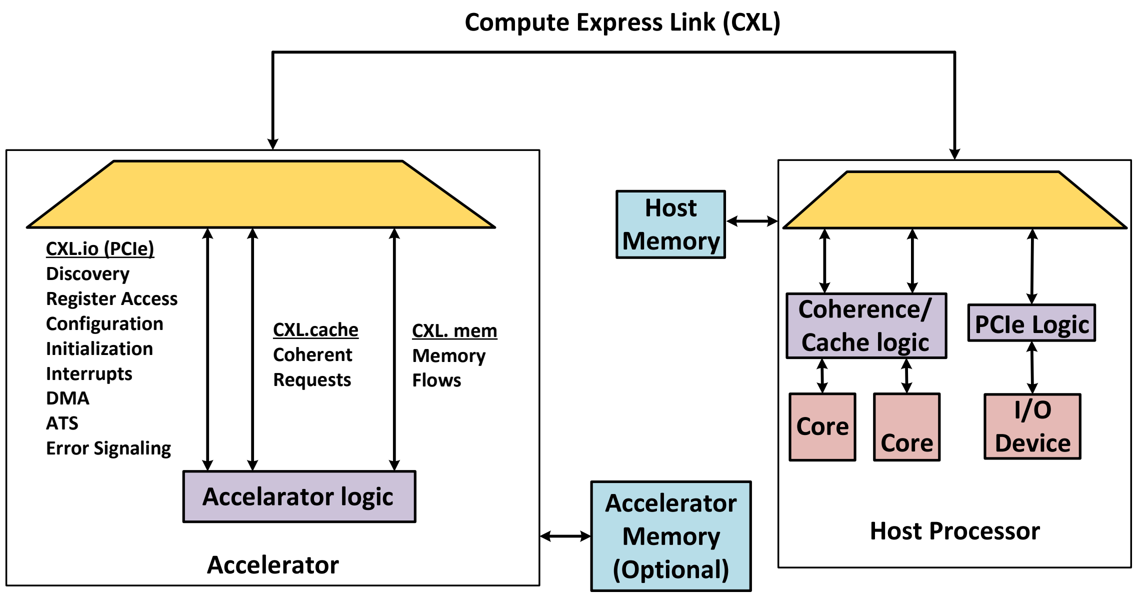


Figure 3.13: Overview of Compute Express Link (Cxl) CXL Consortium (2019) Interconnect (Which Uses the Pcie as Its Interface): The Cxl Provides a Protocol Specification over the Pcie Physical Layer To Support Memory Extensions, Caching, and Data Transactions from I/O Devices, While Concurrently Supporting the Pcie Protocol. I/O Devices Can Use Either the Pcie Protocol or the Cxl. The Cxl Transactions Include `Cxl.Io` Which Provides The Instructions for Traditional Pcie I/O Transactions, I.E., Memory Mapped I/O (Mmio), `Cxl.Cache` Which Provides The Instructions for Cache Coherency and Management, And `Cxl.Mem` Provides the Instructions for Memory Read and Write Between I/O Device Memory and System Memory.

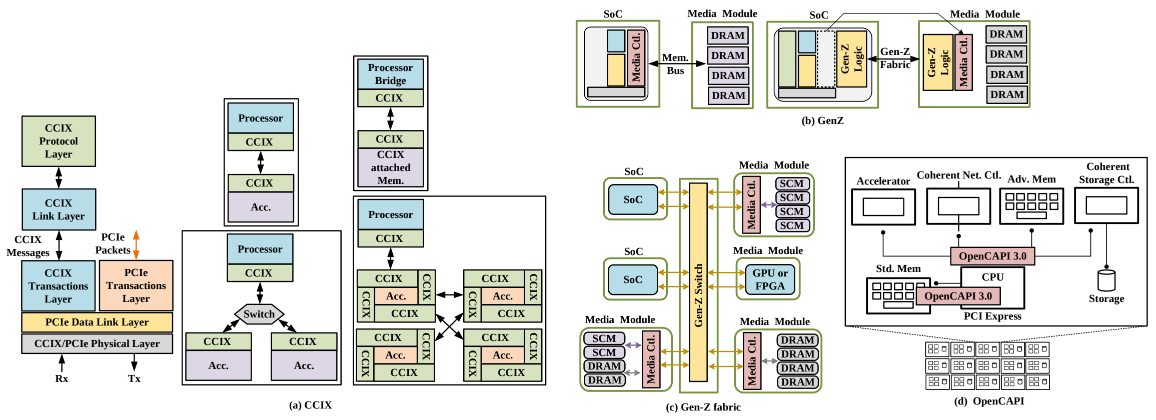


Figure 3.14: Overview of Coherent Interconnects for Hardware Accelerators Supporting Cache Coherency Across Common Switching Fabric: (A) Cache Coherent Interconnect for Accelerators (Ccix)® CCIX® Consortium Incorp. (2020) Defines a Protocol To Automatically Synchronizes Caches Between Cpu and I/O Devices. (B) And (C) Gen-z Gen (2020) Defines a Common Interface and Protocol Supporting Coherency for Various Topologies Ranging From On-chip And Chip-to-chip to Long-haul Platform-to-platforms. The Media/Memory Controller Is Moved From the Cpu Complex to the Media Module Such That Gen-z Can Independently Support Memory Transfers Across Gen-z Switches and the Gen-z Fabric. (D) Open Coherent Accelerator Processor Interface (Opencapi) Stuecheli *et al.* (2018) Homogeneously Connects Devices to a Host Platform With a Common Protocol To Support Coherency With Memory, Host Interrupts, and Exchange Messages Across Devices.

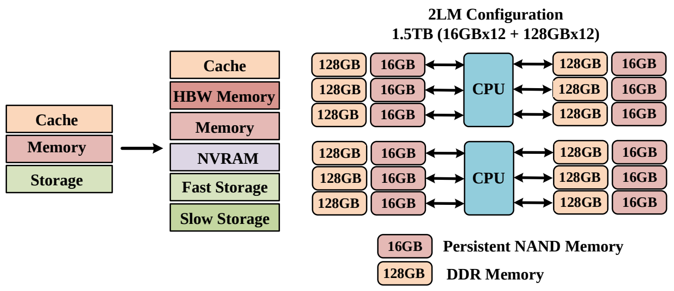


Figure 3.15: Overview of Intel® Optane Dc Persistent Memory Configured as 2 Level Memory (2lm) Where the Dram Is Used As Cache to Store Only the Most Frequently Accessed Data and Nvdimm Is Used as an Alternative to the Dram with the Byte-addressable Persistent Memory (B-apm) Technique.

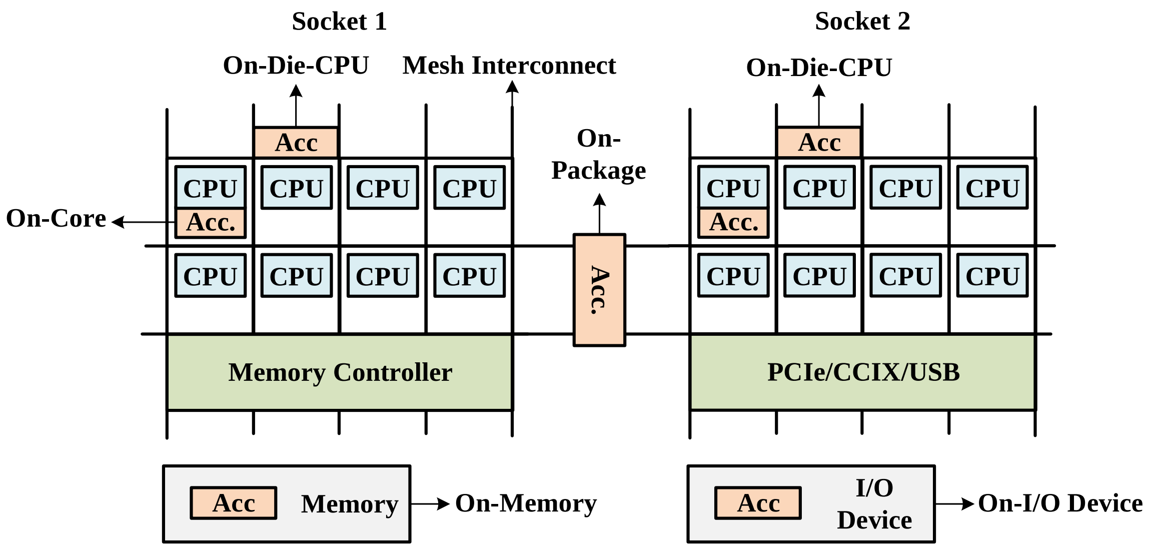


Figure 3.16: Hardware Accelerator Devices Can Be Realized on Silicon In Different Placements: *i*) On-core, Whereby the Accelerator Device Is Placed Right next to a Cpu Core; *ii*) On-cpu-die, Whereby The Accelerator Device Is Placed Around the Cpu Mesh Interconnects; *iii*) On-package, Whereby the Accelerator Device Is Placed Right On-package and External to Cpu-die; *iv*) On-memory, Whereby The Accelerator Function Is Placed on the Memory Module; *v*) On-i/O Device, Whereby the Accelerator Device Is Placed on an External (To Cpu) I/O Device via a Physical Interconnect.

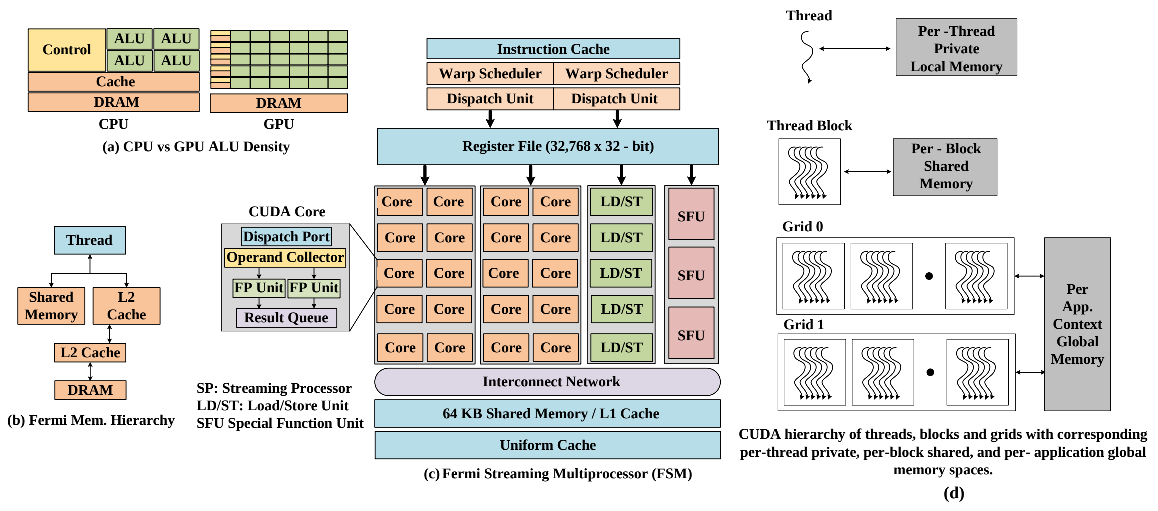

Control
ALU
ALU
ALU
ALU
Cache
DRAM
CPU
DRAM
GPU
(a) CPU vs GPU ALU Density
Thread
Shared Memory
L2 Cache
L2 Cache
DRAM
(b) Fermi Mem. Hierarchy
CUDA Core
Dispatch Port
Operand Collector
FP Unit
FP Unit
Result Queue
SP: Streaming Processor
LD/ST: Load/Store Unit
SFU Special Function Unit
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32 - bit)
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
(c) Fermi Streaming Multiprocessor (FSM)
Thread
Per -Thread Private Local Memory
Thread Block
Per - Block Shared Memory
Grid 0
Grid 1
Per App. Context Global Memory
CUDA hierarchy of threads, blocks and grids with corresponding per-thread private, per-block shared, and per- application global memory spaces.
(d)

Figure 3.17: Overview of Typical Graphics Processing Unit (Gpu) Architecture: (A) Illustration of Arithmetic Logic Units (Alus) Specific to Each Core in a Cpu as Compared to a Gpu; A Gpu Has A High Density of Cores with Alus with Relatively Simple Capabilities as Opposed to the More Capable Alus in the Relatively Few Cpu Cores, and (B) Overview of Memory Subsystem of Fermi Architecture with a Single Unified Memory Request Path for Loads And Stores, One L1 Cache per Sm Multiprocessor, and a Unified L2 Cache. (C) Overview of Fermi Streaming Microprocessor (Fsm) Which Implements the Ieee 754–2008 Floating-point Standard, With a Fused Multiply-add (Fma) Instruction for Single and Double Precision Arithmetic. (D) Overview of Cuda Architecture That Enables Nvidia Gpus to Execute C, C++, and Other Programs. Threads Are Organized in Thread Blocks, Which in Turn Are Organized into Grids NVidia Fermi (2009).

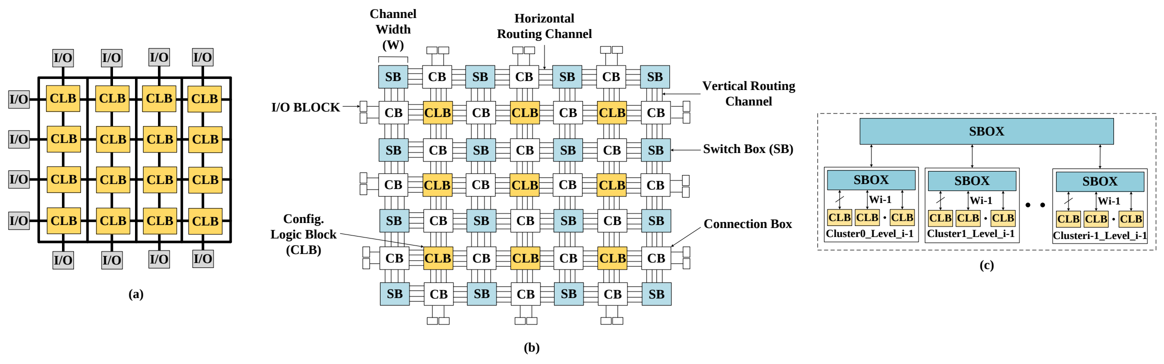


Figure 3.18: (A) Overview of Fpga Architecture: Configurable Logic Blocks (Clbs) Are Interconnected in a Two-dimensional Programmable Routing Grid, with I/O Blocks at the Grid Periphery. (B) Illustration of a Traditional Island-style (Mesh Based) Fpga Architecture with Clbs; The Clbs Are "islands in a Sea of Routing Interconnects". The Horizontal and Vertical Routing Tracks Are Interconnected Through Switch Boxes (Sb) and Connection Boxes (Cb) Connect Logic Blocks in the Programmable Routing Network, Which Connects to I/O Blocks. (C) Illustration of Hierarchical Fpga (Hfpga) with Recursively Grouped Clusters of Logic Blocks, Whereby Sboxes Ensure Routability Depending on The Topologies Farooq *et al.* (2012).

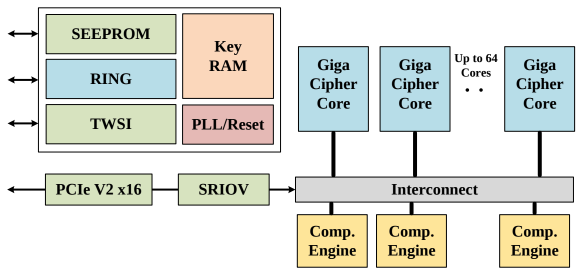


Figure 3.19: Block Diagram of Nitrox Cryptography and Compression Accelerator Marvell (2020): 64 giga Cipher Cores Offer High Throughput Due to Parallel Processing, Coupled with Dedicated Compression Engines. The Nitrox Hardware Accelerator Is External To The Cpu and Interfaces with the Cpu via a Pcie Interconnect.

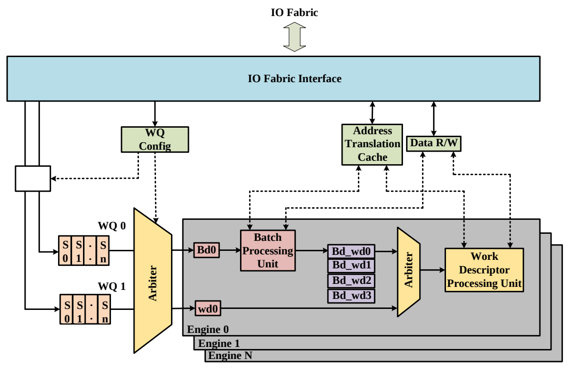


Figure 3.20: Illustration of High-level Blocks Within The Intel® Dsa Device at a Conceptual Level. In Dsa, The Receiving of Downstream Work Requests from Clients and Upstream Work Requests, Such as Read, Write and Address Translation Operations, Are Accessed with the Help of I/O Fabric Interfaces. The Inclusion Of Configuration Registers and Work Queues (Wq) Helps in Holding Of Descriptors by Software, While Arbiters Implement Qos and Fairness Policies. Batch Descriptors Are Processed Through the Batch Processing Unit by Reading the Array of Descriptors from the Memory And the Work Descriptor Is Composed of Multiple Stages to Read Memory, Perform Data Operations, and Write Data OutputIntel Corp. (2019a).

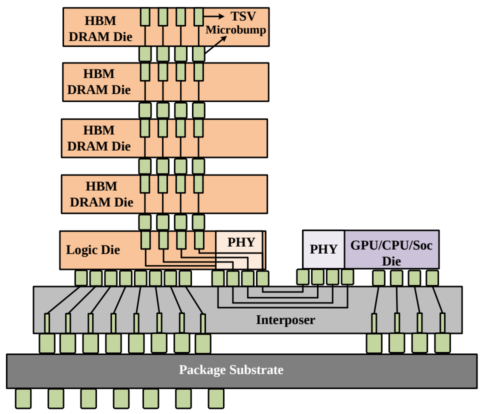


Figure 3.21: Illustration of High-bandwidth Memory (Hbm) with Low Power Consumption and Ultra-wide Bus Width. Several Hbm Dram Dies Are Vertically Stacked (to Shorten the Propagation Distance) And Interconnected by "through-silicon Vias (Tsv)", While "microbumps" Connect Multiple Dram Chips Macri (2015) The Vertically Stacked Hbms Are Plugged into an Interposer, I.E., An Ultra-fast Interconnect, Which Connects to a Cpu Or Gpu Macri (2015).

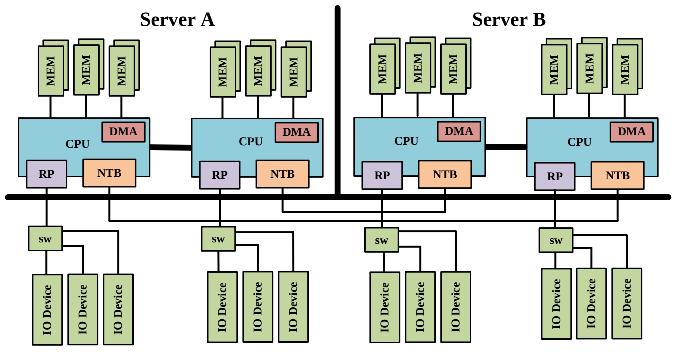


Figure 3.22: Overview of Linux Server with Non-transparent Bridges (Ntbs) Regula (2004): The Memory Regions of Servers a And b Can Be Inter-mapped Across Platforms to Appear as Their Own Physically Addressed Memory Regions. An Ntb Physically Interconnects Platforms in 1 : 1 Fashion Through a Pcie Physical Interface. In Contrast to the Traditional Pcie Root Port (Rp) And Switch (Sw) Based I/O Device Connectivity, the Ntb from One Platform Connects Non-transparently to the Ntb Interface On Another Platform, Which Means That Either Side of the Ntb Appears As End-point to Each Other, Supporting Memory Read and Write Operations, Without Having Transparency on Either Side. In Contrast, a Normal Pcie Switch Functions Essentially as A Non-non-transparent-bridge, I.E., As a Transparent Bridge, By Giving Transparent Views (to Cpu) of I/O Devices, Pcie Root Port, And Switches. On the Other Hand, the Ntb Hides What Is Connected Beyond the Ntb, a Remote Node Only Sees the Ntb, and the Services Offered by the Ntb, Such as Reading and Writing to System Memory (Dram) or Disk (Ssd Pcie Endpoint) Without Exposure of the Device Itself.

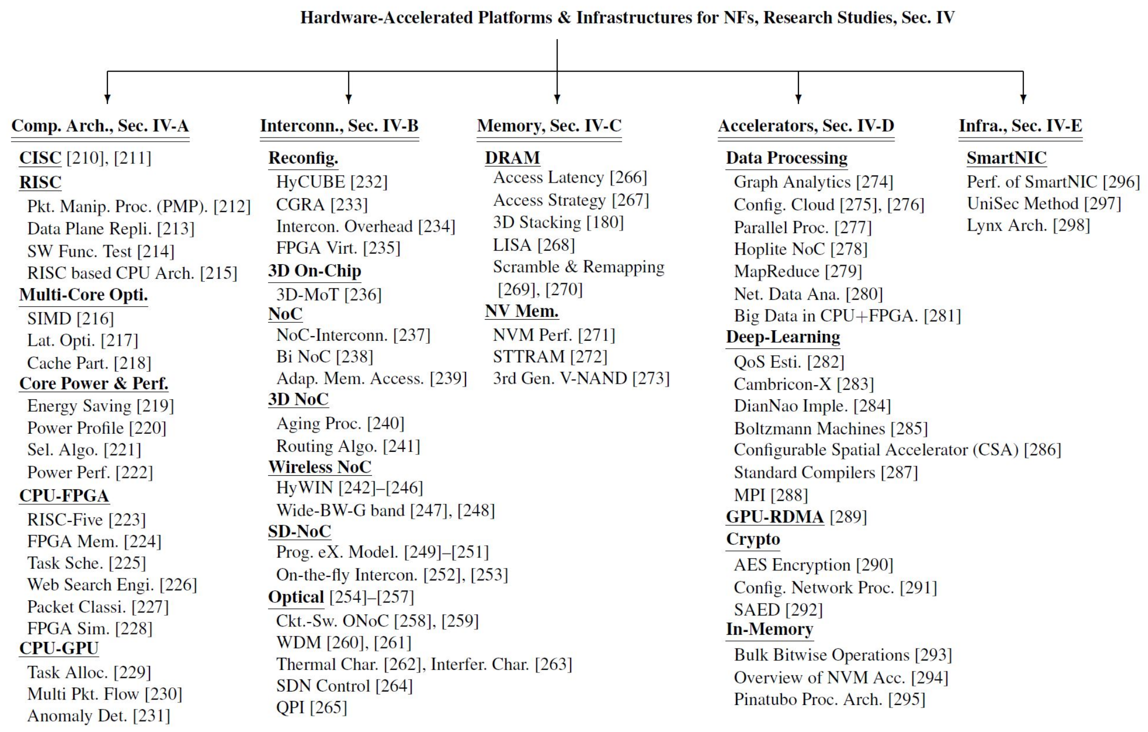
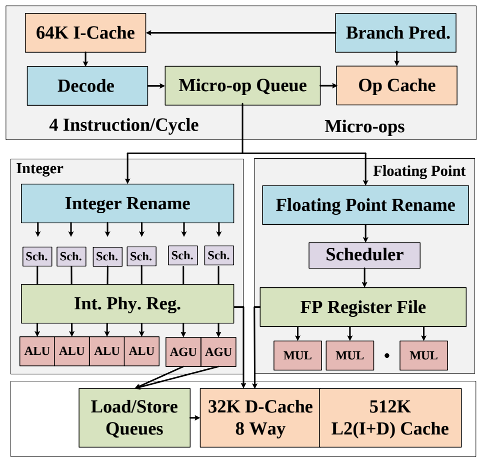
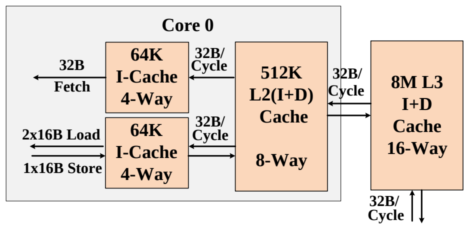


Figure 3.23: Classification Taxonomy of Research Studies On Hardware-accelerated Platforms and Infrastructures for Processing Softwarized Nfs.



(a) ZEN Micro Architecture



(b) ZEN Cache Hierarchy

Figure 3.24: (A) Overview of Zen Micro ArchitectureClark (2016); AMD (2020): The Zen Micro Architecture Has 3 Modules: *i*) Front End Module, *ii*) Integer And Floating-point Modules, and *iii*) Memory Subsystem Module. Each Core Performs Instruction Fetching, Decoding (Decodes 4 Instructions/Cycle into the Micro-op Queue), and Generating Micro-operation (Micro-ops) in the Front End Module. Each Core Is Independent with Its Own Floating-point and Integer Units. The Zen Micro Architecture Has Split Pipeline Design at the Micro-op Queue Which Runs Separately to the Integer and Floating Point Units, Which Have Separate Schedulers, Queues, and Execution Units. The Integer Unit Has Multiple Individual Schedulers Which Splits The Micro-ops and Feeds Them to the Various Alu Units. The Floating-point Unit Has a Single Scheduler That Handles All The Micro-ops. In the Memory Subsystem Module, the Data from The Address Generation Units (Agus) Is Fed into the Execution Units Via the Load and Store Queue. (B) the Zen Architecture Has A Single Pipeline Cache Hierarchy for Each Core Which Reduces The Overall Memory Access Latency.

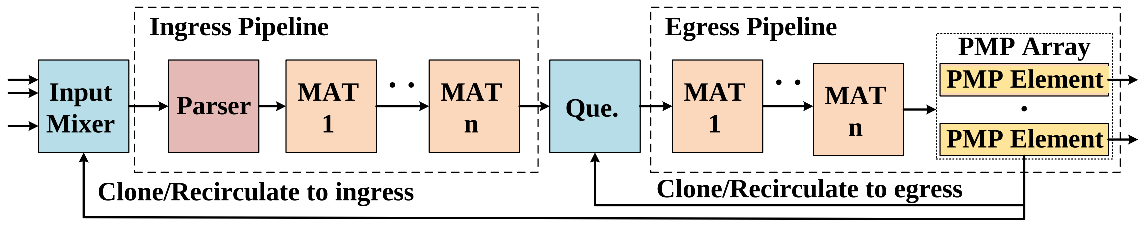


Figure 3.25: An Overview of the Risc Based Packet Manipulation Processor (Pmp) Pontarelli *et al.* (2019) Which Implements a Programmable Packet Header Matching Table Based on Atomic Operations. The Table Can Be Dynamically Updated by Multiple Processes Running on the Cpu Without Impacting the Matching Operations.

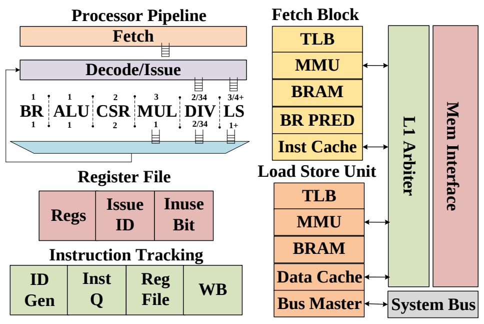


Figure 3.26: Taiga Computing Architecture with Reconfigurable Design Using Fpga Matthews and Shannon (2017). The Compute Logic Units, Such As Alu, Branch Unit (Br), Multiply (Mul), and Division (Div), Are Implemented with Independent Circuitry, I.E, with Instruction Level Parallelism (Ilp). Block Ram (Bram) and Branch Prediction (Br Pred) Assist in the Ilp Opcode Fetch. The Numbers on Top of the Logic Units Are Processing Latencies in Terms Clock Cycles, and below Are The Throughputs in Number of Instructions per Clock Cycle. The + Indicates That Numbers Shown Are Minimum Latency and Throughput Values, Whereas / Indicates Dual Instruction Flow Paths For Execution.

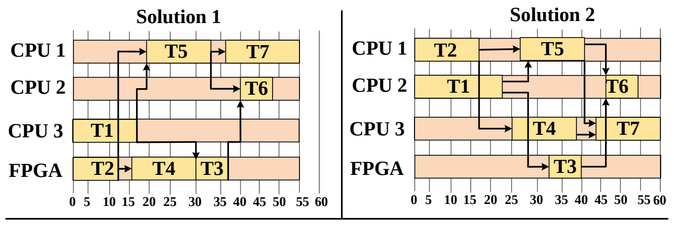


Figure 3.27: Illustration of Heterogeneous Scheduling Between Cpu and Fpga Where Different Tasks Are Commonly Scheduled Relative to Number Of Clock Cycles Abdallah *et al.* (2019): Solutions 1 and 2 Are Possible Scheduling Paths for Tasks (T1–t7) among Cpus and Fpga Homogeneously. The Optimization Algorithm Evaluates All Possible Paths and Estimates the Best Path in Terms of Lowest Overall Processing Latency and Power.

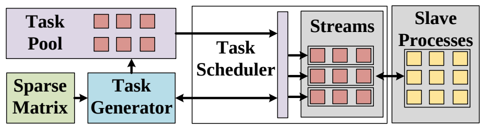


Figure 3.28: Overview of Dynamic Task Scheduling Framework Nie *et al.* (2019) for Cpu-gpu Heterogeneous Architecture: Tasks Are Partitioned into Parallel (Gpu) Execution Tasks and Single-threaded (Cpu) Execution Tasks and Are Then Scheduled as Gpu and Cpu Slave Processes. The Streams Organize The Tasks Such That the Inter-scheduling Intervals of Tasks Are Minimized.

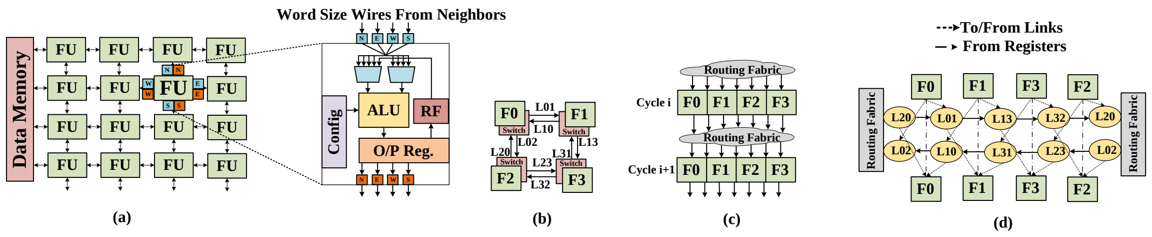


Figure 3.29: HyCUBE Karunaratne *et al.* (2017) Is an Extension To Coarse-grained Reconfigurable Arrays (Cgras) to Support Multi-hop (Distant) Single Clock Cycle Routing Between Functional Units (Fu): (A) Illustration of 4×4 Cgra Interconnected by 2d Mesh, (B) Fu Placement on Routing Fabric with Bidirectional Link Support. (C) Logical Overview of Routing Fabric Between the Fus In Hycube, Where Each Fu Node Can Communicate with Every Other Node Within 1 Clock Cycle. (D) Illustration of Routing Fabric Internals Showing Interconnect Links (E.G., L20, L02), and Their Interfaces to Fus. The Direct Paths from Top Fus to Bottom Fus Are Register Paths, and the Paths Between Link Interconnects And Fus Are “to” and “from” Interfaces to Link and Fus.

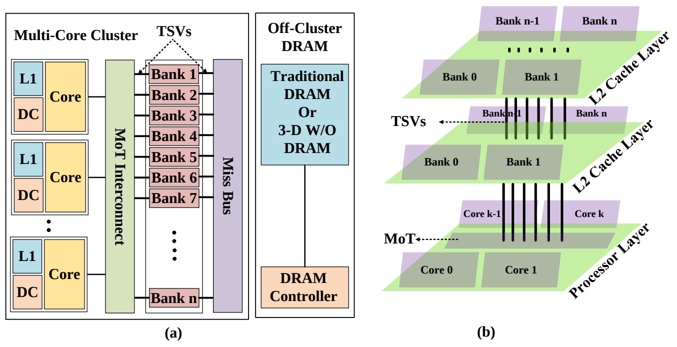


Figure 3.30: (A) Overview of Mesh-of-trees (Mot) Based Interconnect Over 3d Multi-core Cluster with L2 Cache Stacking Facilitated By Tsvs. Each Simple Core in the Multi-core Cluster Contains Its Own L1 Instruction Cache and Data Cache (Dc). Multiple Sram Banks That Are Connected with the 3d Mot Interconnect Through a Tsv Bus Form a Multi-bank Stacked L2 Cache. The Miss Bus Handles Instruction Misses in a round Robin Manner. (B) Geometric View Of 3d Multi-core Cluster Which Balances the Memory Access Latency From Each Core by Placing the Mot Interconnect in the Center Of The Cores Kang *et al.* (2016b).

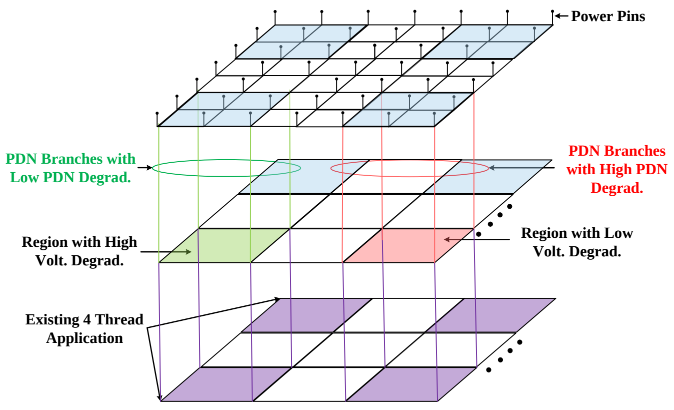

Power Pins
PDN Branches with Low PDN Degrad.
PDN Branches with High PDN Degrad.
Region with High Volt. Degrad.
Region with Low Volt. Degrad.
Existing 4 Thread Application

Figure 3.31: An Example of a 3d Package of a 12 ($2 \times 2 \times 3$)-core Chip Multiprocessor (Cmp) with a Regular 3d Power Grid: A 4-thread Application Is Running in the Four Bottom Layer Cores and a New 1-thread Application Is to Be Mapped. Suppose a High Power Delivery Network (Pdn) Degradation (High Resistance of Pdn Pillars Due to High Currents That Supported Prior Workloads), but Also a Low Circuit-threshold Voltage Degradation (I.E., Little Circuit Slowdown Due to Little Bias Temperature Instability Circuit Aging) Exists in the Red (Right Front) Region of the Middle Layer; Whereas the Green Region Has A Low Pdn Degradation, but High Voltage Degradation. The Artemis Aging-aware Runtime Application Mapping Framework for 3d Noc-based Chip Multiprocessors Raparti *et al.* (2017) Considers Both Pdn And Voltage Degradations.

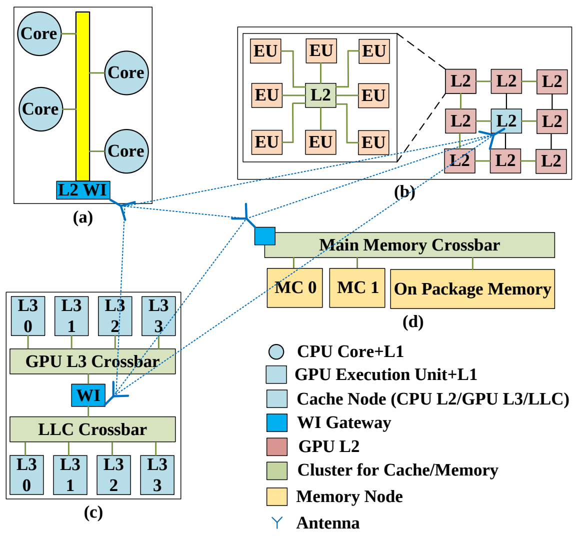


Figure 3.32: Illustration of Wireless Noc Hywin Gade and Deb (2016): (A) The Cpu Subsystem with Cpu Cores (along with Their Respective L1 Caches) Is Connected to a Bus Interface; The L2 Cache Is Shared Between All Cpu Cores. (B) the Gpu Subsystem with Shared L2 Cache At the Center Connects Multiple Execution Units in a Star Topology; All Shared L2 Caches Are Connected Through a Mesh Topology. The Wi Gateway at the Center Initiates the Communication Between The Blocks. (C) and (D) the Required Program Data Are Stored in The Shared Cache Subsystem and Main Memory Subsystem.

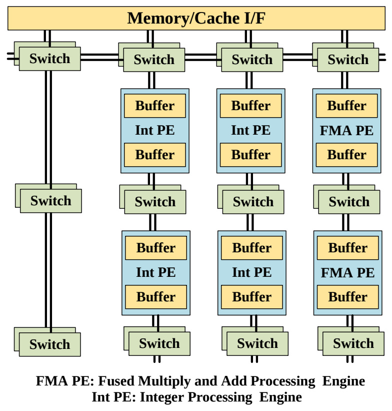

Memory/Cache I/F
Switch
Buffer
Int PE
FMA PE
FMA PE: Fused Multiply and Add Processing Engine
Int PE: Integer Processing Engine

Figure 3.33: Overview of Configurable Spatial Accelerator (Csa) For Supporting Cpus with Large Data Graph Computations as Required For Nf Applications Related to Deep Learning, Data Analytics, And Database Management Intel Corporation (2020b): Highly Energy-efficient Data-flow Processing Elements (for Integer and Fused Multiply-add (Fma) Operations) with Independent Buffers Are Interconnected By Multiple Layers of Switches.

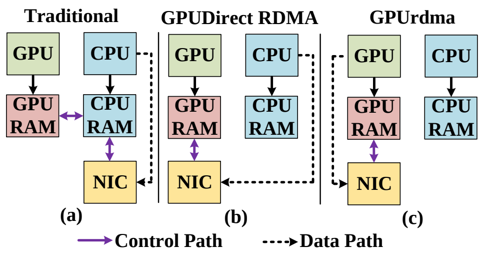


Figure 3.34: Evolution of Gpu-rdma Techniques Daoud *et al.* (2016): (A) Traditional Method of Gpu Accessing Rdma with Assistance from Cpu, (B) Gpu Accesses Rdma Directly from Nic, but Cpu Still Performs The Connection Management for the Gpu, and (C) Gpu Interacts with Nic Independent of Cpu, Thereby Reducing the Cpu Load for Gpu Rdma Purposes.

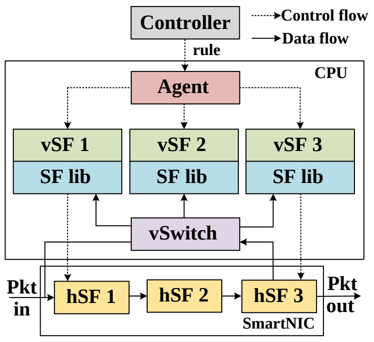


Figure 3.35: Unisec Implements a Unified Programming Interface To Configure and Utilize the Security Functions Implemented On Smartnics Yan *et al.* (2019). Smartnics Implement Hardware Security Functions (Hsf) Which Are Exposed to Applications by Virtual Security Functions (Vsf) Through a Unisec Security Function (Sf) Library.

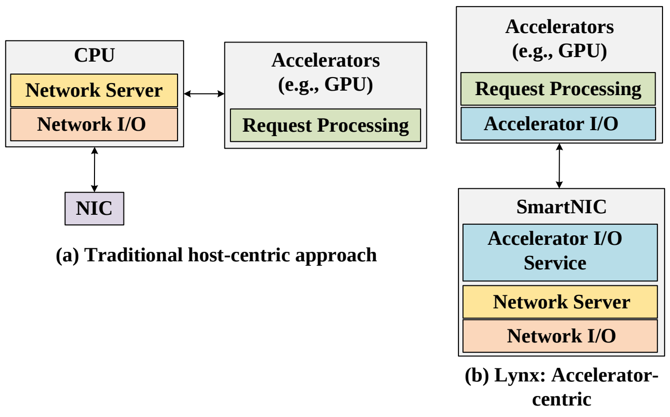


Figure 3.36: (A) Overview of Traditional Host-centric Approach: Cpu Runs the Software That Implements Network-server Function (For Interacting with Remote Client Nodes) to Process the Requests From A Remote Node (over the Network). Cpu Also Runs the Network I/O, Which Implements the Network Stack Processing; (B) Overview Of Lynx Architecture Tork *et al.* (2020): Smartnic Implements The Network-server (Remote Requests Processing), Network-i/O (Network Stack Processing), and Accelerator I/O Service (Accelerator Scheduling) Such That the Cpu Resources Are Freed From Network-server, Network I/O, and Accelerator I/O Services.

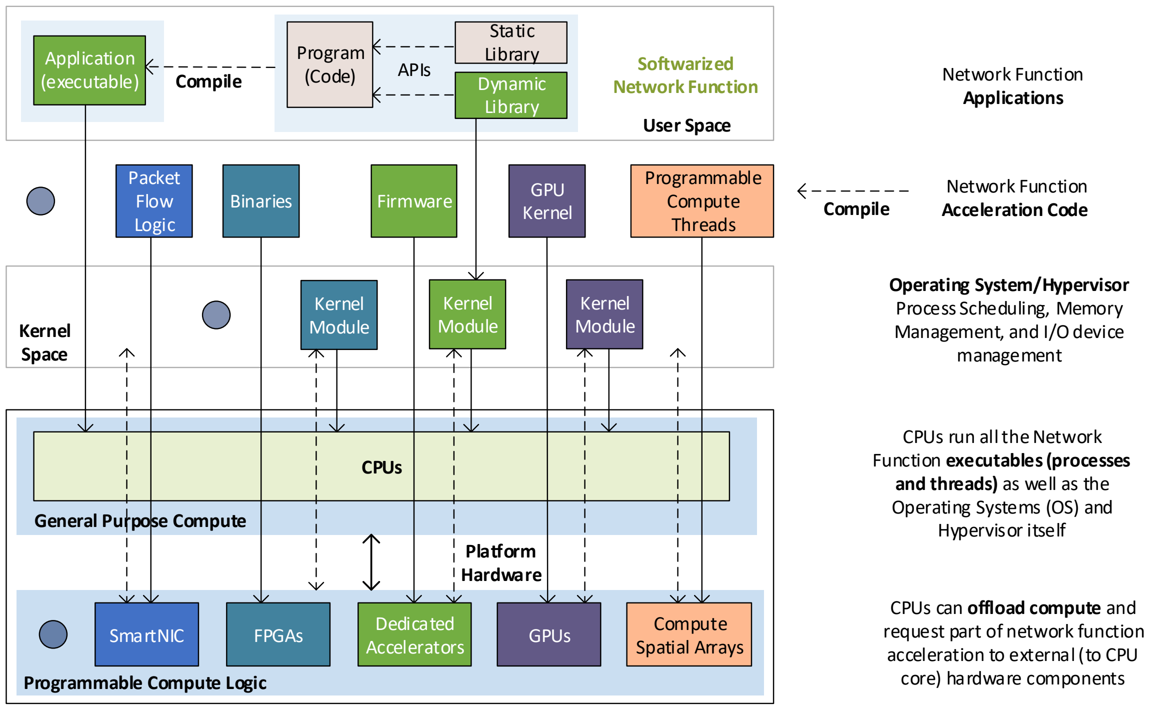

Application (executable)
Compile
Program (Code)
APIs
Static Library
Dynamic Library
Softwarized Network Function
User Space
Network Function Applications
Packet Flow Logic
Binaries
Firmware
GPU Kernel
Programmable Compute Threads
Compile
Network Function Acceleration Code
Kernel Space
Kernel Module
Kernel Module
Kernel Module
Operating System/Hypervisor Process Scheduling, Memory Management, and I/O device management
CPUs
General Purpose Compute
Platform Hardware
CPUs run all the Network Function executables (processes and threads) as well as the Operating Systems (OS) and Hypervisor itself
SmartNIC
FPGAs
Dedicated Accelerators
GPUs
Compute Spatial Arrays
Programmable Compute Logic
CPUs can offload compute and request part of network function acceleration to external (to CPU core) hardware components

Figure 3.37: Software Components in the Softwarized Network Component.

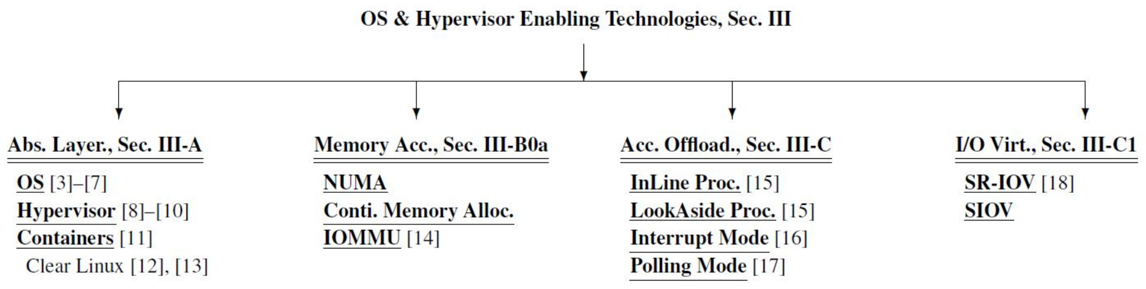


FIGURE2: Classification Taxonomy of Survey of Research Studies on OS and Hypervisors

Figure 3.38: Classification Taxonomy of Survey of Research Studies on Os and Hypervisors

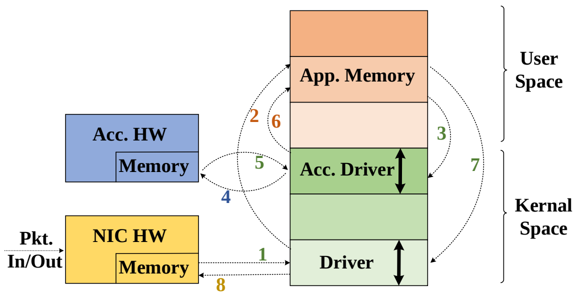


Figure 3.39: Packet Data Traversal Through Memory of Nic Device, Kernel And User Space of System Memory, and Accelerator Device in Tradition Application Processing by the Os.

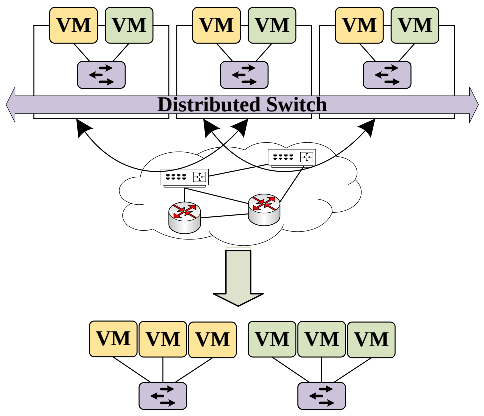


Figure 3.40: Logical Distributed Switch (Lds) Spans Across Multiple Compute Nodes, Where Vms Across Multiple Compute Node Belonging to Same Network Are Able to Communicate over a Logical Switch Transparently. Each Compute Node Has the Context of Lds Which Integrates with Lds Component on Other Compute Nodes to Extend as a Distributed Switch.

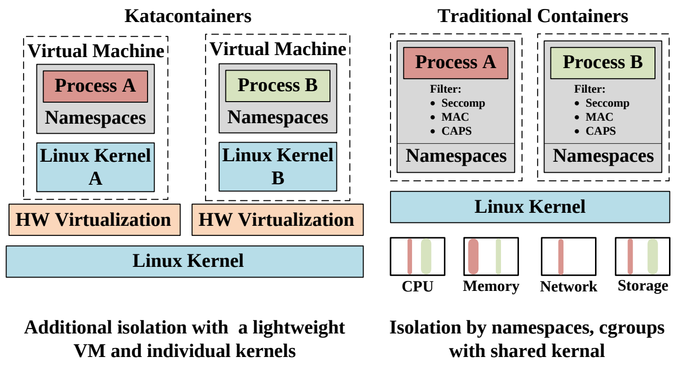


Figure 3.41: Kata Containers Achieves the Benefits of Both Virtual Machine (Vm) and Containers by Extending the Vm Level Isolation to Containers.

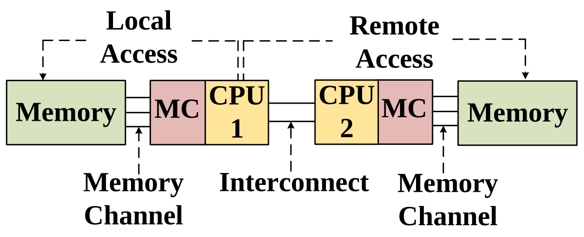


Figure 3.42: Non-uniform Memory Access (Numa) Memory Access Types: *i*) Local Memory Access Which Corresponds to Memory Access On The Same Socket of Cpu, and *ii*) Remote Access Which Corresponds To Memory Access on a Different Socket of Cpu.

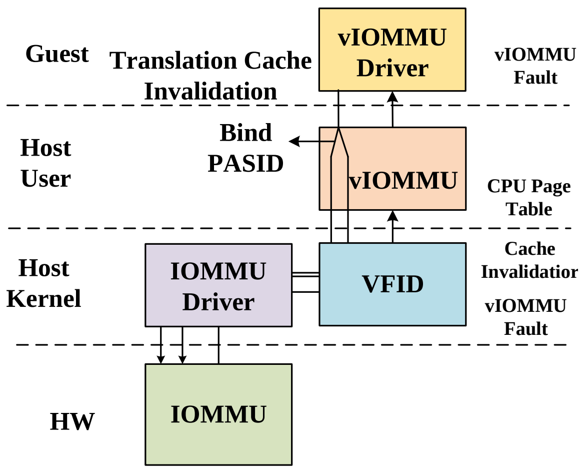


Figure 3.43: Shared Virtual Memory Enabled Vm to Share User Space Memory Location with an I/O Device Based on Iommu Feature. [Bottom up] Iommu Performs the First Level Translation from Physical Memory To Guest-physical (Host-virtual) at the Vmm Level, and Viommu Performs The Second Level Translation from Guest-physical to Guest-virtual. Process Address Space Identifiers (Pasid) Are Used to Isolate The Share Virtual Memory Between Different Vms.

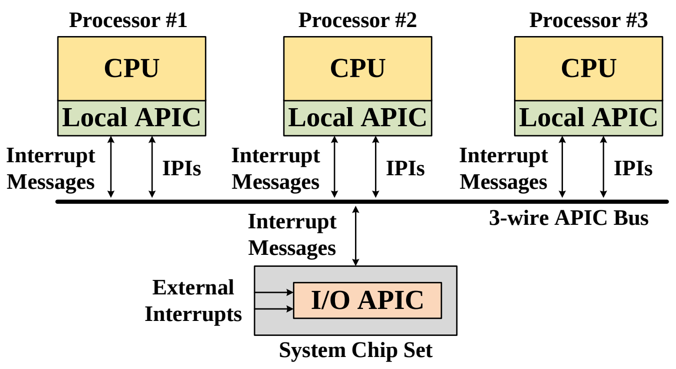


Figure 3.44: Overview of Local Advanced Programmable Interrupt Controller (Aipc) and I/O-aipc: Local Apic Is Core Specific and Enables Local Interrupt Pints(Lint0 And Lint1). Inter-process Interrupts (Ipi) Use System Bus to Forward An Incoming Interrupt or to Generate New Interrupt Targeted to Another Local Apic for Applications, Such as High Resolution Timer, Performance Monitoring Counters, and Thermal Sensors. I/O Apic Extends the Interrupts to External I/O Devices.

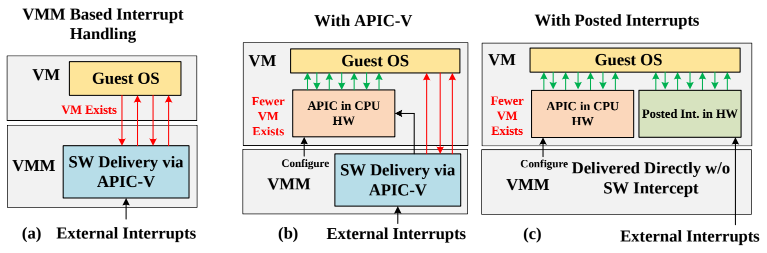


Figure 3.45: Interrupt Delivery Methods to Guest Os Running on A Vmm Corporation and Mulnix (2020): (A) in Traditional Methods, External Interrupts Are Captured by Advanced Programmable Interrupt Controller Virtualized (Apicv) on the Hypervisor and Send A Software Interrupt to the Guest Os. (B) Cpu Implements Apicv Hardware Which Is Assigned to Guest Os, External Interrupts Captured By Vmm Are Delivered to Apicv of the Guest Os. (C) External Interrupts Are Directly Captured by Guest Os Without Any Software Intervention (of Vmm) in the Form of Message Writes to A Memory Region Resulting in Posted Interrupts.

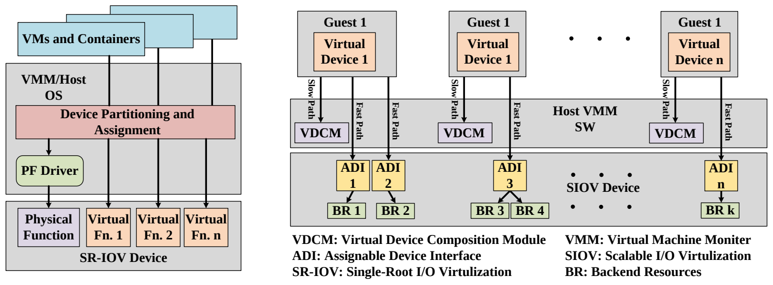


Figure 3.46: Single Root I/O Virtualization (Sriov) And Scalable I/O Virtualization (Siov) Overview Intel Corp. (2018). Sriov Provides Fixed Resource Splitting for Virtualizing Hardware Functions, Whereas Siov Provides a More General And Flexible Resource Splitting Ways for Virtualizing Of Hardware Resources.

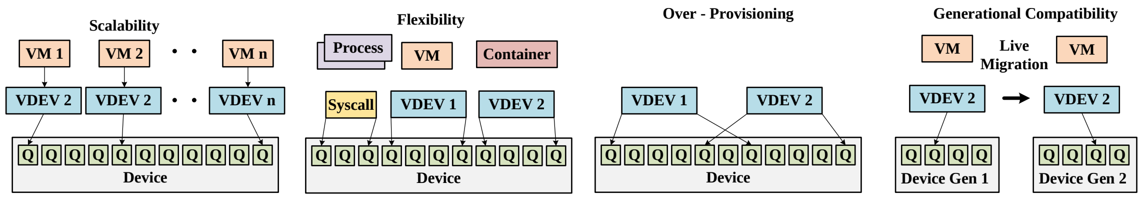


Figure 3.47: Siov Hardware Virtualization Enables Resource Slicing That Is, *i*) Scalable with Respect to the Number of Virtual Functions, *ii*) Flexible in Terms Allocating Virtualized Resources To Different Software Entities Such as Processes, Threads, Applications, and Vms, *iii*) Over-provisioning with Respect To Resource Sharing and Re-use, and *iv*) Compatibility with Respect To Live-migration of Services and Vm Intel Corp. (2018).

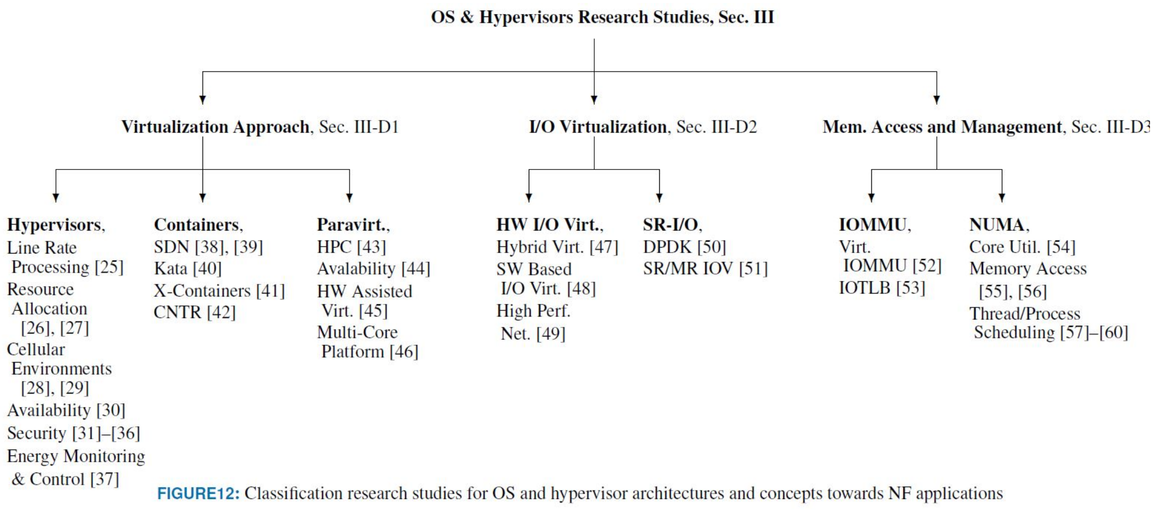

OS & Hypervisors Research Studies, Sec. III
Virtualization Approach, Sec. III-D1
I/O Virtualization, Sec. III-D2
Mem. Access and Management, Sec. III-D3
Hypervisors, Line Rate Processing [25] Resource Allocation [26], [27] Cellular Environments [28], [29] Availability [30] Security [31]–[36] Energy Monitoring & Control [37]
Containers, SDN [38], [39] Kata [40] X-Containers [41] CNTR [42]
Paravirt., HPC [43] Avalability [44] HW Assisted Virt. [45] Multi-Core Platform [46]
HW I/O Virt., Hybrid Virt. [47] SW Based I/O Virt. [48] High Perf. Net. [49]
SR-I/O, DPDK [50] SR/MR IOV [51]
IOMMU, Virt. IOMMU [52] IOTLB [53]
NUMA, Core Util. [54] Memory Access [55], [56] Thread/Process Scheduling [57]–[60]
FIGURE12: Classification research studies for OS and hypervisor architectures and concepts towards NF applications

Figure 3.48: Classification Research Studies for Os and Hypervisor Architectures and Concepts Towards Nf Applications

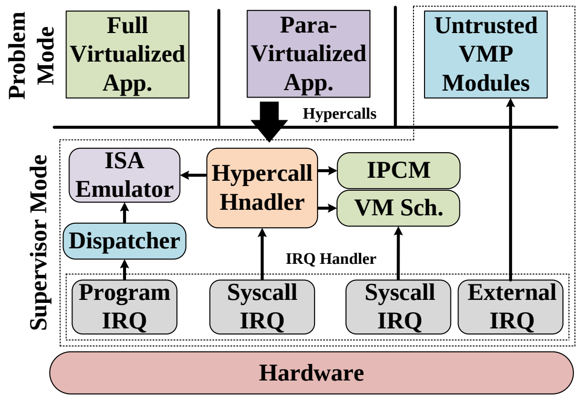


Figure 3.49: Proteus Architecture Where Privileged Code Is Executed in the Supervisor Mode Layer While the Problem Mode Handles Vm Related Communication Such as I/O Device Drivers, Etc. Gilles *et al.* (2013)

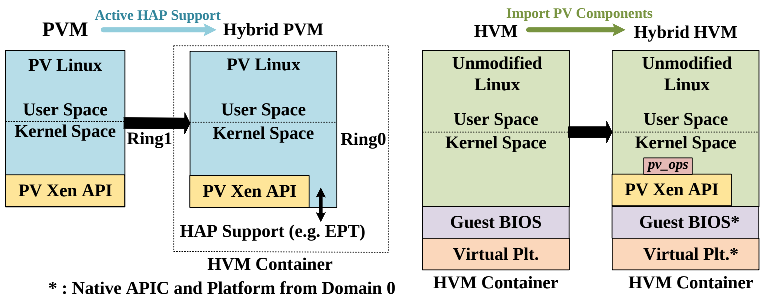


Figure 3.50: Hybrid Virtualization Architecture: (A) Adding Hardware Assisted Paging (Hap) Support to a Paravirtualized Linux Vm, and (B) Showing the Hardware Assisted Virtualization Modifications to Import Paravirtualization Components Nakajima *et al.* (2011).

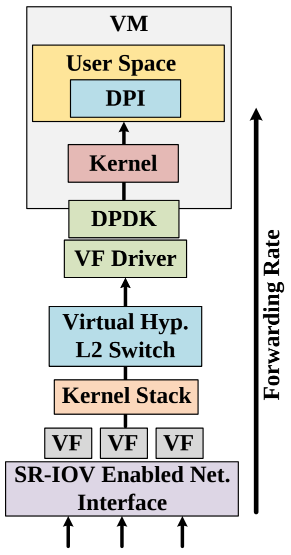


Figure 3.51: Sr-iov Combined with Dpdk Allowing High Throughput For Virtual Nf Applications Kourtis *et al.* (2015).

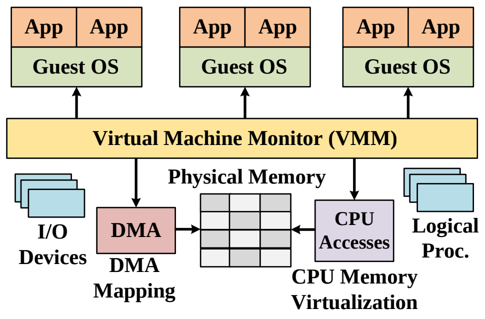


Figure 3.52: Interaction Between I/O, Processor, and Hypervisor Virtualization System.

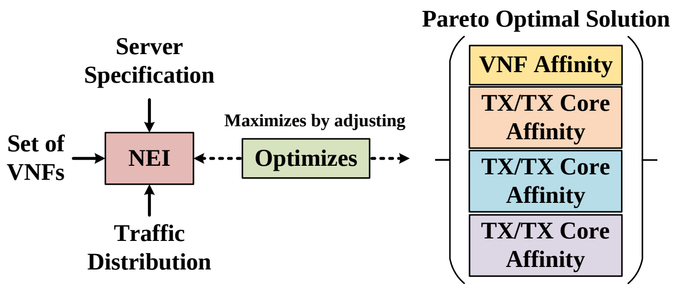


Figure 3.53: Optimization Illustration That Shows the Procedure of Optimizing Nei to Measures the Cost of Vnf Placement on Numa Based Virtualized Systems for Packet Processing Workloads Sieber *et al.* (2017).

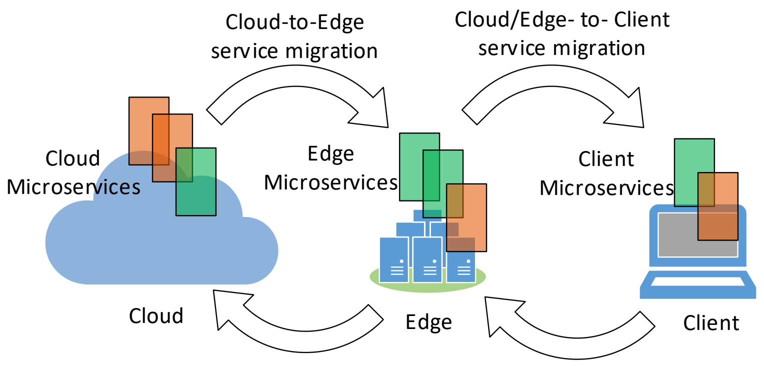


Figure 4.1: Illustration of Cloud-to-edge and Cloud/Edge-to-client Service Migrations. Containers Can Implement the Micro-services, And Multiple Micro-services Can Be Interconnected to Form a Fully Functional Service. Micro-services Are Typically Moved From Cloud-to-edge or Cloud/Edge-to-client so as to Achieve the Required Service Response Times.

Chapter 4

HARDWARE ACCELERATION FOR RESOURCE CONSTRAINED PLATFORMS

4.1 Introduction

4.1.1 Motivation and Background

Containers provide a light-weight cloud-native framework for implementing services that can be separated into several micro-services Kratzke and Quint (2017); Sharma *et al.* (2016); Sotomayor *et al.* (2006). The micro-services can be scaled to demand and migrated between nodes according to application needs Abdah *et al.* (2019); Addad *et al.* (2020); Bellavista *et al.* (2019); Bulkan *et al.* (2018); Ma *et al.* (2020); Wang *et al.* (2018).

Client devices with extensive computing capabilities and fast high-bandwidth

5G wireless connectivity are becoming ubiquitous. However, there is also a trend towards ultra-low latency (ULL) user applications Huang *et al.* (2020); Nasrallah *et al.* (2018); Sharma *et al.* (2020); Xiang *et al.* (2019). One strategy to support ULL applications is to migrate some of the container based micro-service computing to the computationally powerful client devices. Essentially, the cloud and the Multi-access Edge Computing (MEC) Shah *et al.* (2020); Shantharama *et al.* (2018b); Zhang *et al.* (2020) are extended by the client devices Ferrer *et al.* (2019); Goyal (2014); Mehrabi *et al.* (2019); Xie *et al.* (2020), as illustrated in Fig. 4.1. This participation of the client devices in the micro-service computing requires flexible container migrations between cloud, MEC, and client devices Morabito (2016); Cepa (2005).

Container storage images (for brevity "container images") are instantiated and migrated by a Container Engine (CE), which is a container management framework. The container images transition through so-called storage registries Littley *et al.* (2019), which are memory regions on a file system, either on a local system or on a remote server. The storage registry stores the container images in compressed form. Accordingly, the CE has to compress container images before storage them; also, the CE has to decompress container images after retrieving them from a storage registry. If the storage registry is on a remote server, then in addition compressing and decompressing the container images, the GE also has to (*i*) encrypt the container images before sending them over the network to the storage registry, and (*ii*) decrypt the container images received over the network. Typically, the connection between the storage registry on a remote server and the CE is secured through RESTful connections, such as HTTPS.

Client devices are typically resource constrained for workloads that require large I/O transactions, e.g., NFV packet processing and virtualization (abstraction), for the resource sharing among multiple isolated environments, such as VMs and containers.

Hence, applications on client devices depend on limited computing, memory, networking, and power resources to meet the resource needs of the specialized workloads, such as NFV requiring both large I/O transactions and virtualization. While client devices typically do not run workloads that demands large compute, memory, and I/O requirements, specialized applications, such as softwarized network functions often exceed the requirements for resources on the client devices. Dedicated (e.g., ASIC) and custom (e.g., GPU and FPGA) accelerators are specialized hardware designs that can be integrated onto client platforms to enhance the platform resources on-demand to meet application requirements. However, the downsides of using an accelerator are: *i*) offload overhead involving memory transactions (i.e., copies) resulting in increased processing latency, *ii*) CPU overheads for coordinating the task offload, e.g., polling the accelerator for task completion (memory copy operations are covered in [*i*]), and *iii*) I/O [Double Data Rate (DDR)] bandwidth resource overhead from the increased memory transactions. In contrast to client devices, server platforms typically have enough resources to accommodate accelerator offload overheads. Running large workloads on accelerators can achieve high efficiency gains, thus reducing the impact of overheads. Therefore, challenges of accelerator adoption include: *i*) limited CPU and I/O bandwidth resources, and *ii*) small workloads on the client platforms. To this end, we comprehensively investigate the application performance gains, compute and I/O bandwidth requirements, along with power implications on the client devices in the context of the container migration framework.

4.1.2 Use-Case Examples

The container migration framework on client devices is expected to play an important role in Multi-Access Edge Computing (MEC). Generally, client devices should leverage local computing on the client devices whenever possible on opportunistic chances, especially when there are enough computing and networking resources. An efficient container migration framework can assist client devices to manage the resources during instantiation and tear-down of container sessions, as well as state updates to a logically centralized database (e.g., edge or cloud servers).

Speech/Text Recognition

Applications that depend on learning and inference, such as Natural Language Processing (NLP) and handwriting recognition, requires large computing and memory resources as well as privacy due to user data association. If client devices were to operate on large data sizes with frequent interactions to edge or cloud servers, it may be economical to containerize the server operations, transfer the container image to the client device, and run the container image locally on the client device, provided that there are enough client resources to support the containerized application.

Enterprise Remote Desktop

Enterprise client devices are diverse and heterogeneous, whereby an employee of an enterprise or large corporation will typically interact with multiple devices at different locations such as home, office, and in transit. The challenge here is to synchronize the applications and device states across multiple devices. One way to synchronize is to containerize the applications and migrate the container images to the device of choice based on the user request.

4.1.3 Related Work

Today, most Internet services are supported by cloud-native applications Kratzke and Quint (2017). The cloud-native applications are typically implemented on a virtualized data center environment, such as virtual machines (VMs) or containers.

VMs have relatively high overheads Kratzke and Quint (2017); Scheepers (2014); Sharma *et al.* (2016); Sotomayor *et al.* (2006) and are therefore not considered in detail in this study; instead we focus on low-overhead container implementations. Virtualized environments, e.g., the virtualization of data center hardware resources are enabled by abstraction technologies, such as Intel Virtualization Technologies (VT) Neiger *et al.* (2006), that reduce the overhead from software virtualization. While there exist techniques to mitigate the overhead arising from virtualization abstraction, there exists no support from the platform or hardware to reduce the overhead on the containers.

Traditional service migration between the edge and cloud moves VMs, which incurs relatively high computing and networking overheads. A container based framework is better suited for client devices with limited computing and network resources. While there exist soft-handoffs of VM based services through live-migration strategies; to the best of our knowledge, there is currently no method that natively supports live container migration. Therefore, container migration currently requires hard handoffs with offline store-and-forward of container images through storage registries. Ma et al. Ma *et al.* (2017) have addressed this limitation by proposing a migration through a layered storage system that reduces the file system synchronization overhead. Evaluations from Ma et al. Ma *et al.* (2017) show 80% reduction of the handoff latency. Nadgowda et al. Nadgowda *et al.* (2017) proposed a complete state container migration with Checkpoint/Restore In Userspace (CRIU)-based memory migration, minimizing the overhead to be less than 3% of the actual workload.

The service migration from cloud to MEC, cloud to client, MEC to client, and client to client requires the movement of VMs and containers among cloud, MEC, and clients in a heterogeneous cloud-first computing approach. Shen et al. Shen *et al.* (2016) have presented VM based application migration methods based on the geographical movement of workloads.

Aside from short handoff latencies, low energy consumption is a key goal for VM or container migration. Kaur et al. Kaur *et al.* (2017) have examined the energy consumption of migrations and found that utilizing containers instead of VMs reduced the energy consumption by 21.75%. Overall, container based services currently achieve the lowest overhead, shortest latency, and lowest energy consumption.

Several hardware and software strategies can accelerate cloud services Linguaglossa *et al.* (2019); Shantharama *et al.* (2020). Hardware acceleration can employ custom accelerators, such as FPGA Lallet *et al.* (2018) and GPU Giunta *et al.* (2010), and dedicated accelerators for service-specific hardware functions, such as DLBoost on Intel Cascade Lake processors Arafa *et al.* (2019), which can accelerate deep learning algorithms on both containers and VMs. While container based services provide significant benefits over VM based services, containers do not strongly isolate the services. The vulnerability of containers for both cloud and client devices has been assessed in Martin *et al.* (2018).

In contrast to containers, Container Engine (CE) accelerations speed up the management of containers, including the storage, retrieval, instantiation, building, and tear-down of container images. Lu et al. Lu *et al.* (2019) have developed an acceleration method for container image updates across different versions on the CE by avoiding the duplication of container data between image versions, achieving a 7-fold improved CE image update performance. The container instantiation time depends on the residency of the container image in the system memory, hence Gu et al. Gu *et al.* (2019) have developed a method to store container image layers on Non-Volatile Memory (NVM) so as to accelerate container deployments. Huang et al. Huang *et al.* (2019) have accelerated the building of containers by caching the necessary files for incremental builds, reducing data download times up to 70%. Importantly, all of these approaches require

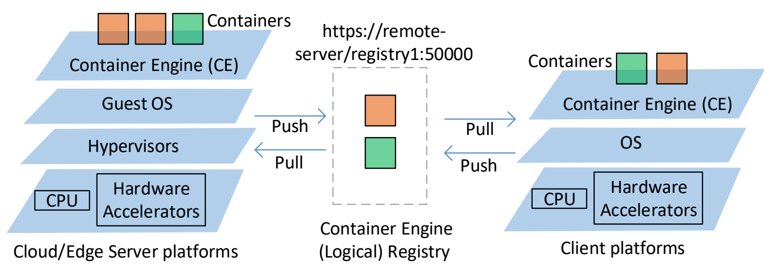


Figure 4.2: Container Migration Through a Container Engine (Ce) Registry: Container Images Are Pushed from a Node to a Registry, and Pulled From the Registry by Another Node. The Pushing of a Container Image To the Registry Involves Saving the Running Container Context as A New Container Image, Compressing the Image, and Then Uploading The Image to the Registry. Pulling a Container from the Registry Involves Downloading the Image from the Registry and Running It on A Container to Resume the Service.

the compression/decompression and encryption/decryptionAkhilesh S. Thyagaturu (2021) of container images; the acceleration of the compression/decompression and encryption/decryption of container images is therefore critical for efficiently transitioning container between computing nodes.

4.1.4 Contributions

Although several existing strategies can accelerate CE container management functions, such as the storage, instantiation, and building of container images, the critical aspect of CE acceleration for encryption and compression of container images has not yet been investigated in detail. We address the open topic area by evaluating the CE acceleration for compression and decompression of container images for transitioning through CE storage registries. More specifically, we compare the CPU utilization for CE processes for executing compression and decompression in software (SW) on the CPU and for executing compression and decompression in an Intel

QuickAssist Technology (QAT) hardware accelerator. To understand the implications of accelerator offloading on memory and I/O interactions, we quantify the relative memory read/write activity for compression and decompression of container images. We also quantify the overall task completion performance with and without hardware acceleration. We believe that the quantitative evaluation data presented in this article will serve as an important reference benchmark for container migration framework designs as well as for container management involving compression and decompression of container images on client devices.

To investigate the implications of network bandwidth on the use of hardware acceleration, we employ a container migration framework between two client devices connected over a network. The access network (e.g., cellular, cable, WLAN) that connects client devices to an external network are typically unreliable. For instance, even with 5G connectivity, mobility scenarios and loss of line-of-sight with base stations could significantly impact bandwidth and latency of the network connectivity. We quantify the performance implications of container migrations with and without hardware acceleration as a function of the network bandwidth. Our evaluations indicate that hardware accelerations with low network bandwidth can negatively impact the overall application performance. Additionally, we quantify the power consumption for applications with and without hardware accelerations as a function of the network bandwidth. We find that hardware accelerations show significant power savings which is critical for client devices. Our results serve as a reference profile for future designs of client devices in adopting hardware accelerators, especially in resource-constrained environments.

4.2 System Model

4.2.1 Cloud and Client Nodes

In the cloud context, micro-services and applications are deployed as containers in an isolated environment managed by a CE, such as Docker or CRI-O. The data center server platforms are typically virtualized by a hypervisor, such as Hyper-V or VMware ESXi; the hypervisor abstracts and slices hardware for the guest OSs. Containers typically run on Guest OSs and are managed by CEs. Virtualization can be nested to achieve multi-level abstractions and desired levels of isolation between different services and guest OSs. Server platforms can be equipped with hardware accelerators to assist software functions through custom (e.g., GPU, FPGA) and dedicated hardware (e.g., Intel QuickAssist Technology (QAT)). Hypervisors, guest OSs, and containers commonly access the hardware accelerators through abstraction methods and IO virtualization technologies, such as Single Root-IO Virtualization (SR-IOV) Kutch (2011) and S-IOV **?**. To avoid overheads from the abstractions and IO virtualization, an accelerator can also be statically allocated to a hypervisor, guest OS, or container through so-called fixed or pass-through allocation.

In contrast to the cloud, client devices have generally only limited by CPU and hardware resources, including very limited support for general hardware acceleration. Container workloads have traditionally been developed to run on cloud server platforms and are therefore not optimized to run on client devices. Applications and services on client devices typically operate directly (in bare metal fashion, without virtualization) on the hardware to achieve high efficiency. As a result, applications and containers running on client devices should be aware of the hardware resource constraints. Therefore, hardware acceleration on client devices is strongly recommended. Hardware acceleration on client devices can help to minimize disruptions and to mitigate workload congestions when computing container-based micro-services, which require frequent container instantiations and tear-downs. In order to understand the impact of hardware acceleration, we evaluate the implications of container migration workloads on client devices operating with and without hardware acceleration.

4.2.2 Container Migration

The service deployment for latency sensitive applications can implement a micro-service on a client device. This client based implementation logically extends the cloud end-to-end service to the client device. The client based micro-service implementation requires the migration of containers that contribute to the cloud service to client nodes. After some client updates, the container may migrate back to the cloud/edge. Thus, the implementation of container based micro-services requires client devices to perform the container migration functions.

We evaluate the offline container migration method which transports container images between nodes; this offline container migration results in a hard service handoff. The hand-off process and the service interruption delays are not our main concern and we do not propose a novel container migration strategy. Instead, our goal is to evaluate the implications of the hardware acceleration for container migration on the computing and I/O characteristics on client devices. We model the container migration based on the logical registry method. As shown in Fig. 4.2, each node (i.e., client and server) implements a physical registry specific to each node and exposes secure connections to external nodes for the pushing and pulling of the containers to and from the registry. For example, migrating a container from the cloud to a client proceeds as follows: *i*) the cloud server pushes the currently running container to a remote (relative to the cloud server) physical registry of the client (that is local to the client), and *ii*) the client pulls the container from the local (to the client) registry and executes the container.

A drawback of the registry based container migration is that all nodes must implement a physical registry and maintain a dictionary of external HTTPS connections to each node's registry for the pushing and pulling of containers. This requires a full mesh topology, which severely limits the scalability. Our evaluation focuses on one client device that connects to one cloud (server) node, resulting in an elementary `1:1` topology. Thus, the full-mesh requirement does not impact our evaluation and the registry method of container migration is well suited for evaluating the computing implications due to container migration on a single client device.

4.2.3 Acceleration of Container Engine (CE)

At microscopic levels, the instantiation of a container (either through migration or for the first time) proceeds as follows. The CE downloads the container image (which is in a compressed form, and encrypted if downloaded from a remote server), decrypts the image (if downloaded from a remote server), uncompresses the image, and schedules the (decrypted and) uncompressed image to run. This container instantiation effectively requires two CPU intensive tasks: *i*) the decryption of a container image that arrives through a network interface, and *ii*) the decompression of the container image. Conversely, for uploading a container image, the clients needs to compress and encrypt the container image. In general, both compression (and decompression) as well as encryption (and decryption) are highly CPU intensive tasks which impose both heavy power consumption and computing loads on the client devices. Hence, we believe that encryption/decryption and compression/decompression acceleration are critical for client devices.

Usually, encryption and decryption of a container image are provided by an application layer security framework, such as TLS/SSL through HTTPS. End-to-end TLS/SSL and HTTPS connections typically employ the AES-GCM encryption algorithm, which is highly optimized through CPU instruction set acceleration, such as AES-NI Hofemeier and Chesebrough (2012). However, compression and decompression accelerations are not natively provided by the CPU instruction set. Therefore, client devices have to depend on auxiliary acceleration, e.g., custom hardware accelerators (e.g., FPGA) or dedicated hardware accelerators (e.g., Intel QAT), to speed up the compression and decompression.

4.2.4 Compression/Decompression Acceleration

Compression/decompression acceleration can be achieved in both software and hardware.

Software Implementation

GZIP Deutsch (1996) efficiently compresses and decompresses on single-core computers. PIGZ Adler (2014) parallelizes GZIP so that multiple threads running on different cores can exploit parallel computing capabilities. However, PIGZ implements only the decompression functionality, complimentary to GZIP for compression. There exist 10 compression levels, L0–L9, whereby L0 indicates no-compression, and L9 indicates the maximum compression level. The default compression level for GZIP is L6. The implications of the compression level on the computing are the resource requirements; whereby, the higher the compression level, the higher is the compression ratio, but also the larger are the computing and memory requirements, and hence the longer the duration for completing the compression task in resource-constrained devices. For network applications, higher compression levels could result in lower data sizes for transmissions over the network as compared to lower compression levels.

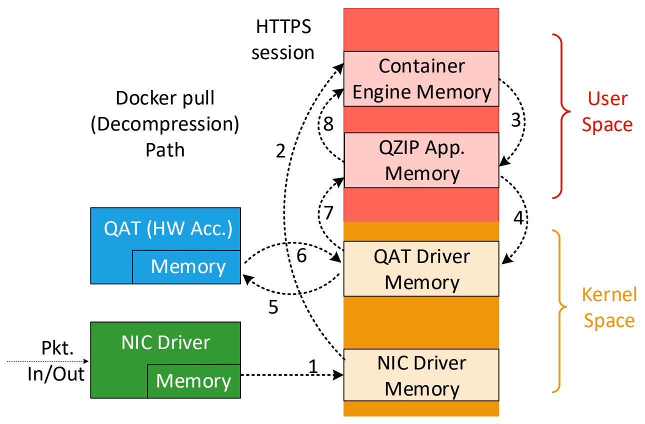


Figure 4.3: Overhead Illustration of Memory Transaction for Hardware Acceleration of Decompression for Docker Pull Operation Using Qat. The Data from Https Session Is Decompressed by Qzip Application Using Qat Driver and Delivers Uncompressed Data To Docker Container Engine (Ce) for Container Image Instantiation.

Hardware Acceleration

The Intel QuickAssist Technology (QAT) Intel Corp. (2019c); Lin (2018) is an ASIC based dedicated hardware accelerator for cryptographic and compression computations. We use the QATzip (QZIP) qatzip (2020) software library to interact and forward the compression and decompression CPU calls from the CE to the QAT hardware. The default compression in the QZIP is L1 to which the hardware accelerator QAT is initialized with based on recommended configuration from the open-source library. We employ both default configuration, i.e., L1 and maximum supported compression level L4 for QZIP, whereas the default configuration of software is L6 during the evaluations of hardware acceleration of CE.

Acceleration Overhead

The downside of hardware acceleration is the overhead associated with the offloading to the hardware accelerator. The primary overhead components are the data transactions between applications, accelerator drivers, and accelerator device-internal memory. Figure 4.3 illustrates the data movements across the various components involved in hardware acceleration for the Docker pull (decompression) processing. The container image, which is stored in the registry in compressed form, is transferred to the Docker host through a network interface and HTTPS session. HTTPS employs transport layer security using TLS/SSL which involves encryption using the Advanced Encryption Standard (AES), such as AES-GCM-128 algorithms. Once the data is decrypted by the CE application, the data is transferred to the QAT and the computing is offloaded to the QAT for decompressing the data. This processes involves two main overhead components due to OS and HV principles: *i*) data copy from Docker context to QAT context, and *ii*) CPU context switching between Docker and QZIP applications for data processing.

Memory Transactions For illustration, we have considered the data transactions of the Docker pull operation in Fig. 4.3 as logical Steps 1–8 between the different memory regions of the Docker and hardware accelerator (QZIP) applications. The container image, which is stored in compressed form at the registry, is transferred through the network interface, and hence the data is copied from the network driver kernel space memory region to the Docker CE application space. Once SSL/TLC decryption of the data is completed, QZIP is invoked to perform the decompression. The data is not only copied from application-to-application, i.e., Docker CE to QZIP, but once again copied from the user-space to the kernel space of the QAT hardware driver, and the other-way once the QAT computing is completed. As a result, each QZIP request for decompression involves four extra data copies due to the use of a hardware accelerator (whereas only four copies would be required for a conventional SW implementation). before the final data can be interpreted by the CE. That is, overall, the use of an accelerator device requires eight copies compared to four copies for a conventional SW implementation. Although data copying from the kernel space memory to the user space memory can be avoided using advanced data processing techniques, such as kernel bypass Høiland-Jørgensen *et al.* (2018); Lettieri *et al.* (2017), shared virtual memory using Process Address Space ID (PASID) Huang *et al.* (2016), and Poll Mode Drivers (PMD) (e.g., Data Plane Development Kit (DPDK)), these mechanisms require application and OS/hypervisor kernel modifications which are typically not considered in the networking stack on resource-constrained client devices. Therefore, our evaluations are based on standard techniques and mechanisms that are typically used in resource-constrained client devices. Techniques, such as PMD, do not fit well for client devices due to the required dedicated core assignments for the continuous polling of network interfaces; this polling is both compute and power inefficient.

Process/Thread Context Switching The concurrent processing of multiple applications results in CPU context switching between processes and threads. As the number of applications and contexts increases, especially from the hardware acceleration components, the overall context switching overhead can become significant. For instance, the Docker CE application requests the QZIP application to processes the data for compression and decompression, resulting in the context switching between the threads and processes of the Docker CE application and the QAT hardware accelerator. These context switches reducing the overall acceleration gain from the hardware accelerator. Therefore, in this study we focus on the effects of hardware accelerator offload overhead especially for network links with low bandwidth and high latencies which regulate the instantaneous amount of compute (data) offloading from the CPU to the QAT hardware.

4.2.5 Container Engine Modifications

We employ the Docker code of the open source project Moby Moby (2020). Due to the structural implementation of the serialized compression framework in the Moby Docker, we have retained GZIP for compression. However, for decompression, we employ the PIGZ software library, which utilizes multiple parallel threads for the decompressing the downloaded container image. The code changes for the QAT adaptation for Docker are freely available from Github Akhilesh (2020).

4.2.6 Execution Algorithm

Algorithm 6 illustrates the execution flow for the SW and QAT (hardware accelerator) implementations of the Docker CE migration. We consider two different evaluation methods: *i*) Local network interface, where the container registry is implemented on the same client device, as considered in the preliminary study Chhajer *et al.* (2020), and *ii*) External network (over-the-network) interface, where the container registry is implemented on a different client device connected via an Ethernet link. The local network interface evaluations focus on the impact of the hardware acceleration on the container migration processing without any network link constraints. In contrast, the external network interface evaluations focus primarily on the impact of the network link (bandwidth and latency) characteristics on the hardware acceleration.

4.3 Evaluations over Local Network Interface

4.3.1 Experimental Setup

Figure 4.4 illustrates the two compared experimental conditions: the benchmark software (SW) implementation compresses with GZIP and decompresses with PIGZ on the client device CPU; the QAT implementation employs the QATzip library to

Algorithm 6: Execution sequence flow for software and hardware acceleration of CE for compression and decompression.

```
Input : iter = $N$, Disable: Turbo, Hyper Threading, and fixed CPU frequency (2.3 and
        4.2 GHz).
1  while $N \geq 0$ do
       Input : Initialize Docker CE
2      if run == QAT then
3          Compile Docker with QAT adaptation;
4          Instantiate new Docker with QAT adaptation;
5      else if run == SW then
6          Instantiate default Docker;
7      Setup link bandwidth and latency settings;
8      Start local/remote registry;
9      Delete all preexisting containers;
10     Download and run Fedora 26 container image;
11     Create desired size data layer;
12     Append the data layer to base container layer;
13     if run == Local Network Interface then
14         Tag the container to local registry (https://localhost/registry1:50000);
15     else if run == External Network Interface then
16         Tag the container to remote registry (https://192.168.1.3/registry:50000);
17     Start timer and SoCWatch;
18     if run == Compression then
19         Delete all preexisting remote containers;
20         Push the container to the registry;
21     else if run == Decompression then
22         Delete all preexisting local containers;
23         Pull the container from the registry;
24     Stop timer and SoCWatch;
25     Collect_statistics();
26     iter ← iter−1;
27 end
```

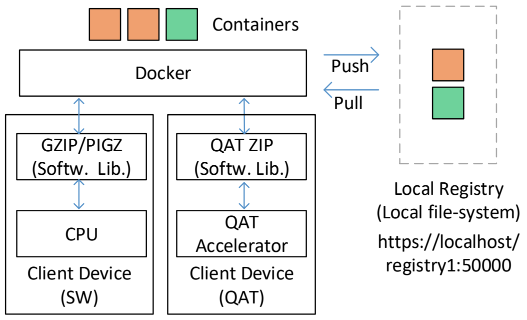


Figure 4.4: Test Setup Overview: Docker Engine Is Modified to Use Intel Quick Assist Technology (Qat) Hardware Accelerator for Compression and Decompression Tasks During Push and Pull of Container Images to the Docker Registry.

Table 4.1: Test Setup Parameters.

Setup Env.	**Details**
CPU Info.	Intel Core™ i7 − 7700K
Num. Cores	4C/8T
Microcode Ver.	`0xd6`
Base Freq.	4.2 GHz
DRAM	32 GB (DDR4)
LLC Size	8192 KB (L3)
Storage	2 TB HDD
OS	4.15.0 − 74/Ubuntu(16.04.1)
Docker	19.03.4, build 9013bf583a
QAT HW Ver.	82C628/QAT1.7.L.4.5.0 − 00034

offload compression and decompression to the QAT hardware accelerator on the client device. We have evaluated the performance on an Intel Core™ i7 − 7700K client platform with traditional Linux kernel of version 4.15.0 − 74 on Ubuntu (16.04.1). The performance evaluations for this study have been conducted in accordance with the guidelines presented in Intel (2019b).

We implemented the Docker local registry within the same client device and performed repeated container push and pull operations using a local network interface (e.g., `https://localhost/registry1:50000`) such that there is minimal to no impact from the external network characteristics on our evaluations. The push and pull operations of containers to and from the registry are performed more than 10 times for a given container size to record the completion times, CPU core utilization, and memory access rates with statistical reliability. Over 97% of the thus sampled performance values are within 5% of the plotted sample means. We performed the push, pull, and registry interactions with a script to simulate the synchronized behaviors of offline container migration on the same node without any additional management overhead, such as end-to-end connection setup, migration trigger, integrity check, or tear-down migration.

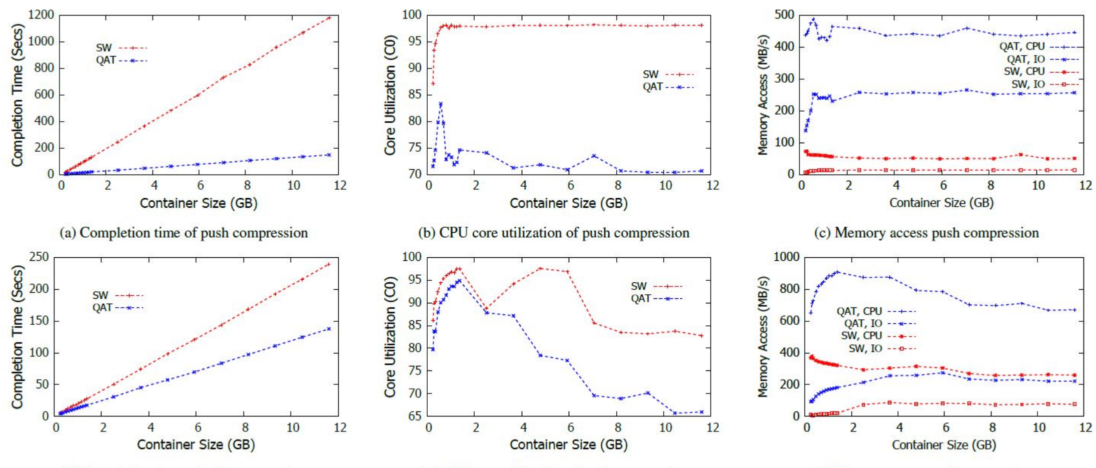
We vary the container sizes from 100 MB – 1 GB with a step size of 100 MB to evaluate the performance for small container sizes, and 1 GB – 10 GB with steps of 1 GB to evaluate the performance for large container sizes. We maintained uniformity in the compression and decompression across the various container sizes by synthetically generating the containers from a single base container image. In particular, we generate a text file with the English dictionary package on Linux for the size determined by the step size of the container sizes, and recursively append to the data layer of the container to obtain the desired container image size. With this method, we aim to retain the entropy of the containers linear with their incremental versions such that the amount of computing required for compressing the container sizes varies linearly. The achieved compression ratio is around 48%.

We have measured the CPU residency durations (for task completion time), CPU core utilization, and memory access (DDR IO) bitrates with the SoCWatch tool. To minimize the variability of the system characteristics during our measurements, we have disabled Hyper Threading (HT) and turbo frequency in the system BIOS. We have set the system core to a fixed frequency of 2.3 GHz throughout the local network interface evaluations. The Docker process is pinned to a single physical core, namely core 0 to avoid context switches with OS internals and other application processes of the system. The QAT accelerator is an add-on PCIe card plugged into the platform board with the capability of 8 lanes width. The applications and system software on the OS access the QAT natively (no virtualization) through direct interactions with the device. Also, the Intel Virtualization Technologies for Directed IO (VT-D) is turned off in the BIOS to avoid IO-Memory Management Unit (IO-MMU) transactions. Table 4.1 summarizes the configuration of the test setup.

4.3.2 Results

Completion Times

Figure 4.5 (a) shows the compression completion time as a function of the container size for the GZIP software (SW) and for the QAT hardware acceleration, while Fig. 4.5(d) shows the decompression completion times. For the compression evaluations, the Docker process compresses the container image and pushes the compressed image to a local registry. The completion times are evaluated based on the total CPU residency time for the Docker process on the core `C0`. As expected, we observe a linearly increase in the completion time with increasing container sizes. The higher linear slope of approximately 100 s/GB for SW compared to the slope of 13 s/GB for QAT indicates that QAT hardware compression acceleration achieves significant reductions of the completion times. Likewise, for decompression acceleration, the SW slope of approximately 20 s/GB compared to 12 s/GB for QAT indicates a substantial decompression time reduction with the QAT hardware acceleration compared to the



(a) Completion time of push compression (b) CPU core utilization of push compression (c) Memory access push compression

(d) Completion time of pull compression (e) CPU core utilization of pull compression (f) Memory access pull compression

Figure 4.5: (A) and (D) Completion Time of Docker Registry Push (Compression) and Pull (Decompression) as a Function of Container Size with Software (Sw) Implementation and with Hardware (Qat) Acceleration (B) and (E) Cpu Core Utilization [Percent] of Docker Registry Push (Compression) and Pull (Decompression) as a Function Of Container Size for Software (Sw) Implementation and for Hardware (Qat) Acceleration. (C) and (F) Memory Bandwidth Consumption From Cpu Core and Io Devices (Qat, Network Interface) for Docker Registry Push (Compression) and Pull (Decompression) as a Function Of Container Size for Software (Sw) Implementation and for Hardware (Qat) Acceleration. Lower Values Are Better for Both Completion Times and Cpu Utilizations. The Memory Access Rate Is an Observable Characteristic Due to Changes in the Completion times and Cpu Utilizations, I.E., Both Higher and Lower Memory Access Rate Values Are Neither Better nor Worse.

multi-thread PIGZ software implementation. Importantly, our results indicate that hardware acceleration is especially critical to keep the compression times for small to moderate sized container images below 100 seconds.

Core Utilization

Figs. 4.5 (b) and (e) present the CPU core utilization for compression and decompression, respectively, whereby the core `C0` utilization includes all the PIGZ threads. We observe a rapid increase of the CPU utilization as we increase the container sizes up to about 500–600 MB, with software based compression showing greater CPU utilization as compared to QAT. For instance, for a container size around 500– 600 MB (where the QAT core utilization peaks), the SW CPU utilization is approximately 15% higher than the QAT CPU utilization. As the container sizes are increased above 600 MB, we observe that the QAT CPU utilization tapers off to values below 75% while the SW utilization saturates near 98%.

For decompression, we observe a non-linear behavior in the CPU utilization originating from the parallelized implementation of the decompression through the PIGZ library. For container sizes up to around 1.4 GB, PIGZ decompression has only very slightly higher core utilization than QAT. For container sizes above 1.4 GB, the QAT utilization tapers rapidly down to nearly 65% while the SW utilization stays above 80% for large container sizes approaching 12 GB. The CPU utilization is defined as the CPU execution time to the total execution time (= accelerator execution time + CPU execution time). For larger container sizes, the accelerator execution time is significantly longer than the CPU execution time, reducing the CPU utilization. Essentially, the CPU off-loads the task to the accelerator and waits idle for completion. Overall, it is important to interpret these CPU utilization results in conjunction with the completion time results: the QAT dramatically reduces the completion time compared to SW (see Fig. 4.5), and during this dramatically shorter QAT completion time, the CPU utilization is still somewhat lower than with SW (see Fig. 4.5). These CPU utilization results provide guidelines for the required CPU capacity for achieving the acceleration benefits.

Memory Access

Figs. 4.5 (c) and (f) present the memory (DDR) read and write access bitrates of the CPU and the IO devices. The IO devices include hardware accelerators, e.g., QAT, and disks. We observe that in the tests with the QAT, the IO memory access rates for compression and decompression are below 900 MB/s and thus well within the access rates supported by contemporary memory technology. These QAT IO memory access rates are mainly due to the QAT accessing the container image data as an IO device. On the other hand, the IO accesses for the SW case are mainly due to background tasks that are not directly related to the compression/decompression.

We also observe from Figs. 4.5 (c) and (f) that the QAT CPU memory access rates are generally higher (roughly 9 times higher for compression, and 3 times higher for decompression) than the corresponding SW rates. The higher QAT CPU memory access rates are mainly due to the CPU feeding the container image data to the QAT as well as receiving and processing the QAT results. We further observe from Figs. 4.5 (c) and (f) that the QAT memory access rates for decompression are roughly twice the rates for compression; whereas the SW memory access rates for decompression are roughly four times the rates for compression. This indicates that the data processing memory transactions for the decompression are substantially more demanding than for compression. The memory transactions complexity increase of decompression is relatively more pronounced for SW processing. The presented memory access rates can inform future hardware acceleration designs, e.g., whether inline or lookaside acceleration should be pursued to operate within the chip-to-chip interconnect and IO fabric limits.

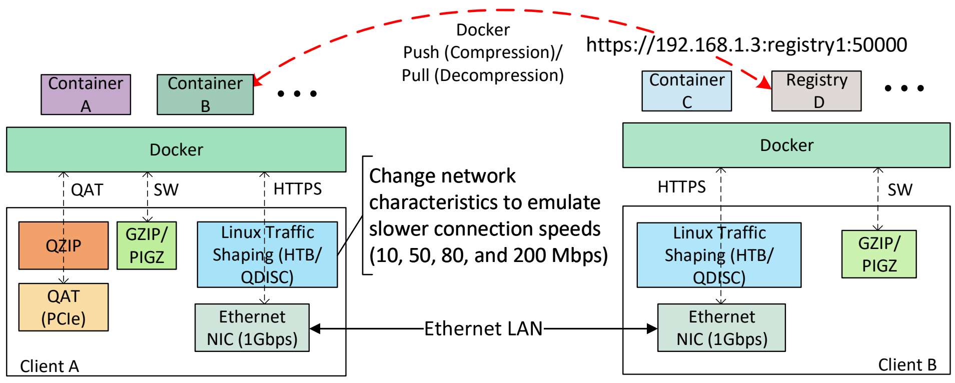


Figure 4.6: Test Setup Overview for Network Interface Based Evaluation: *i*) Registry Implementation Is Moved from the Local Host (Resource-constrained Client a) to an External Node (Client b With Abundant Resources) via a Network Interface (Ethernet Lan), *ii*) The Hierarchical Token Bucket (Htb) of Linux Is Used to Throttle The Network Bandwidths for 10, 50, 80, and 200 mbps, While The Default Link Bandwidth Is 1 gbps, and *iii*) the Queuing Discipline (Qdisc) of Linux Is Used to Control the End-to-end Latencies to 50, 100 ms in Either Direction of the Link (I.E., 50 ms Is Equal to 100 ms of round Trip Time [Rt]).

Table 4.2: Comparisons of Sw and Qat Hardware Acceleration Compression Ratios: Total Container Size = Base Container Layer (Fedora 26) + Data Layer (Random Dictionary Words); Compression Ratio = Compressed Size / Total Container Size.

Compre. Engine	**Data Layer Size** [MB]	**Total Cont. Size** [MB]	**Compre. Size** [MB]	**Compre. Ratio** [%]
QAT L4	200	419	198	44.5
GZIP L6	200	419	173	41.3
QAT L4	1024	1328	652.4	49.1
GZIP L6	1024	1328	605.4	45.6

4.4 Evaluations over External Network Interface

4.4.1 Experimental Setup

Figure 4.6 illustrates the overall setup for the external network interface (over-the-network) evaluations. To understand the performance implications of the network characteristics (such as effective bandwidth and latency) on the hardware acceleration, the container registry is moved from the local host (Client A) to an external host (Client B) that is connected over an Ethernet LAN. The Ethernet Network Interface Card (NIC) and the physical Ethernet link support a bandwidth of 1 Gbps and latency of < 1 ms. The external host that implements the registry (e.g., `https://192.168.1.3/registry:50000`) is also a client device (Client B in Fig. 4.6), but not constrained with abundant resources, and hence can support the registry services for the requesting Client A with QAT without creating a bottleneck.

Network Characteristics The client devices are typically connected to access networks that are unreliable due to (device) mobility, subscriptions, and shared resources (e.g., best effort Ethernet services), which results in constrained network resources impacting the effective end-to-end bandwidth and latencies. To understand the impact of the network characteristics on the hardware acceleration, based on the standard broadband connectivity profiles Xu *et al.* (2019), we consider 5 levels of bandwidth: 10, 50, 80, 200 Mbps, and 1 Gbps (i.e., default link speed) and 3 levels of latency: < 1, 50, and 100 ms in either direction (downlink and uplink) of the link. We use the Linux based Queuing Discipline (QDISC) and Hierarchal Token Bucket (HTB) Devera and Cohen (2002); Balan and Potorac (2009) to emulate the network traffic speeds and latencies on the client devices. Container migrations between Docker CE (Client A) and registry (Client B) are performed over the emulated network speeds over HTTPS connections.

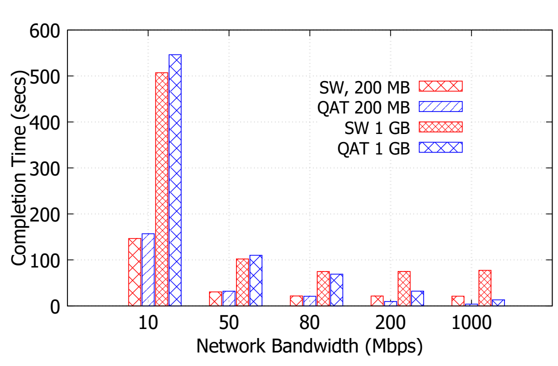
Container Sizes In a typically deployment of containerized micro-services Taherizadeh and Grobelnik (2020), container sizes vary dynamically based on changes in the container contexts involving multiple layers of data as well as a base layer of the container. Table 4.2 compares the container sizes used in our evaluations in their default and compressed forms, along with the compression ratios achieved from software (GZIP L6) and QAT (QAT L4) based compressions. For instance, a total container size of 419 MB results from the base layer Fedora 26 of 219 MB and data layer of 200 MB. Docker compresses these individual layers (base and data) independently, and transfers these layers to a registry. In an actual container deployment and migration processes, layer-aware mechanisms, such as caching and data movements, are performed to improve the efficiency Junping *et al.* (2020). However, in our evaluations we delete all the cached container layers before the container is migrated to and from the registry to quantify the actual gains from the hardware accelerations relative to the container size. We consider two data layer sizes, 200 MB and 1 GB, specifying a small and a large data container layer relative to the base container layer (i.e., Fedora 26 of 219 MB).

In contrast to Sec 4.3, in this section we present the network bandwidth and latency implications on the hardware acceleration of container migration. The setup parameters are maintained constant between the evaluations of local and external network interfaces, except for the operational CPU frequency which is fixed at 4.2 GHz (a base frequency of 2.3 GHz was used for the local interface evaluations) to support the requirements for hardware accelerations overhead and emulating networking bandwidth speeds and latencies.

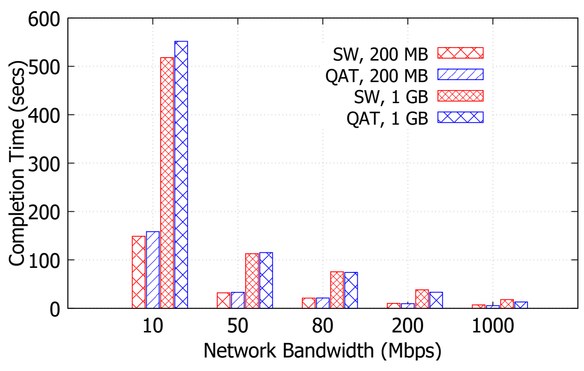
4.4.2 Bandwidth and Latency Evaluations

Completion Time

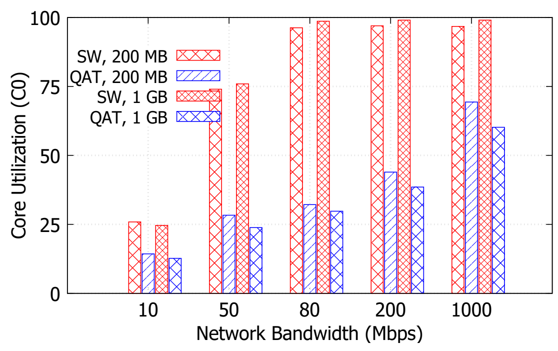
Figure 4.7(a) presents the completion time as a function of the network bandwidth for Docker push (compression) operations for container data layer sizes of 200 MB



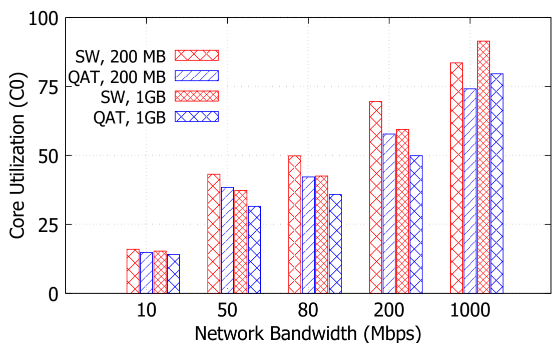
(a) Compression



(b) Decompression

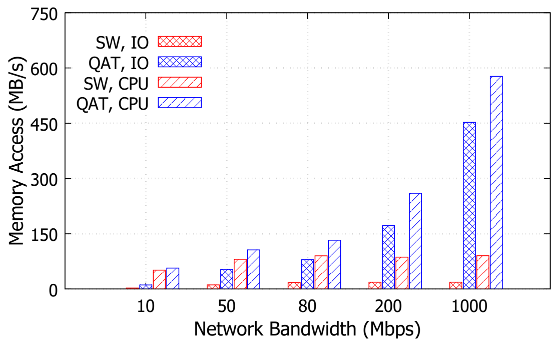


(c) Compression

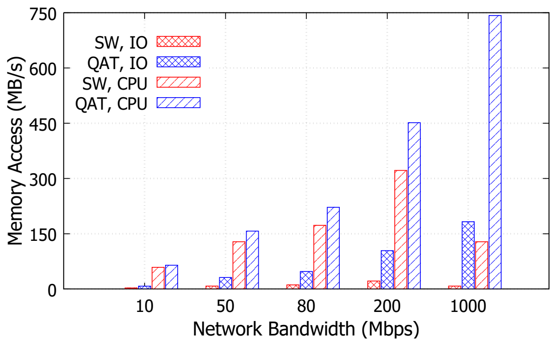


(d) Decompression

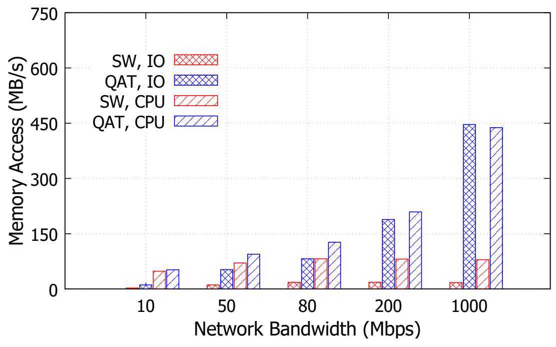
Figure 4.7: Container Migration Performance Comparison as a Function Of Network Bandwidth (Mbps) for Container Data Layer Sizes 200 Mb And 1 Gb with Software (Sw) Implementation and with Qat Hardware Acceleration. (A) Completion Time [Seconds] of Docker Registry Push (Compression); (B) Completion Time [Seconds] of Docker Registry Pull (Decompression); (C) Cpu Utilization [Percent] of Docker Registry Push (Compression); (D) Cpu Utilization [Percent] of Docker Registry Pull (Decompression); Lower Values Are Better for Both Completion Times And Cpu Utilizations. The Memory Access Rate Is an Observable Characteristic Due to Changes in the Completion times and Cpu Utilizations, I.E., Both Higher and Lower Memory Access Rate Values Are Neither Better nor Worse.



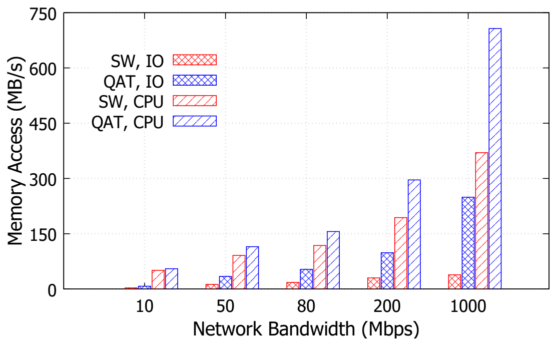
(e) Compression, 200 MB



(f) Decompression, 200 MB

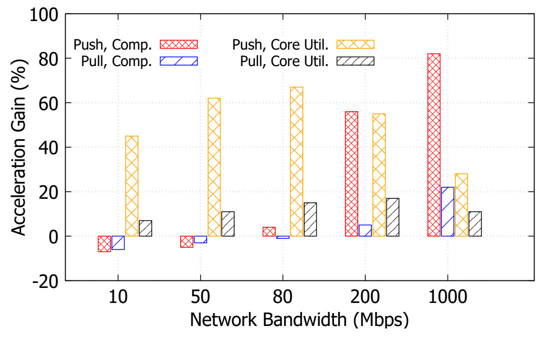


(g) Compression, 1 GB

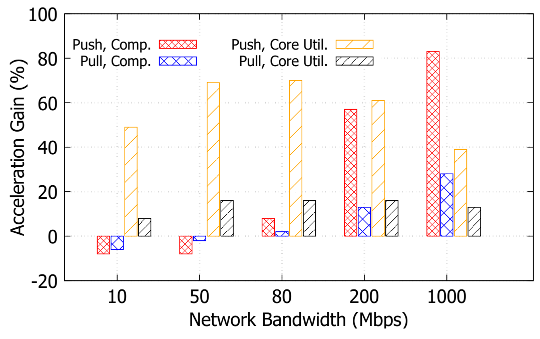


(h) Decompression, 1 GB

Figure 4.8: Container Migration Performance Comparison as a Function Of Network Bandwidth (Mbps) for Container Data Layer Sizes 200 Mb And 1 Gb with Software (Sw) Implementation and with Qat Hardware Acceleration. (E) and (G) Memory Access by Cpu and Qat (And Network Interface) of Docker Registry Push (Compression), For 200 mb and 1 gb, Respectively; And (F) and (H) Memory Access By Cpu and Qat (and Network Interface) of Docker Registry Pull (Decompression). Lower Values Are Better for Both Completion Times And Cpu Utilizations. The Memory Access Rate Is an Observable Characteristic Due to Changes in the Completion times and Cpu Utilizations, I.E., Both Higher and Lower Memory Access Rate Values Are Neither Better nor Worse.



(a) 200 MB



(b) 1 GB

Figure 4.9: Summary of Hardware Acceleration (Qat) Gains as a Function Of Network Bandwidth: (A) for 200 mb, and (B) 1 gb Container Data Layer Sizes. Hardware Acceleration Results in Negative Performance Gains for Low Network Speeds, < 80 mbps. Larger and Positive Values Are Better for Acceleration Gains.

Table 4.3: Power Consump.: Push (Comp.), 200 MB.

B/W (Mbps)	QAT (mW)	SW (mW)	QAT (Hrs.)	SW (Hrs.)	Bat. Sav. (Hrs.)
10	50009.49	6951.8	7.98	5.75	2.23
50	7462.23	15399.85	5.36	2.60	2.76
80	7937.9	19316.13	5.04	2.07	2.97
200	10446.22	19436.41	3.83	2.06	1.77
1000	14321.38	19405.99	2.79	2.06	0.73

Table 4.4: Power Consump.: Pull (Decomp.), 200 MB.

B/W (Mbps)	QAT (mW)	SW (mW)	QAT (Hrs.)	SW (Hrs.)	Bat. Sav. (Hrs.)
10	5623.97	5858.32	7.11	6.83	0.28
50	12030.12	12740.81	3.32	3.14	0.19
80	12812.35	13884.32	3.12	2.88	0.24
200	14531.14	15734.84	2.75	2.54	0.21
1000	15712.86	17115.62	2.55	2.34	0.21

Table 4.5: Power Consump.: Push (Comp.), 1 GB.

B/W (Mbps)	QAT (mW)	SW (mW)	QAT (Hrs.)	SW (Hrs.)	Bat. Sav. (Hrs.)
10	4754.71	6772.3	8.41	5.91	2.51
50	6665.36	15737.87	6.00	2.54	3.46
80	7683.24	19717.66	5.21	2.03	3.18
200	9442.32	19779.24	4.24	2.02	2.21
1000	12971.61	19738.47	3.08	2.03	1.06

Table 4.6: Power Consump.: Pull (Decomp.), 1 GB.

B/W (Mbps)	QAT (mW)	SW (mW)	QAT (Hrs.)	SW (Hrs.)	Bat. Sav. (Hrs.)
10	5754.3	5708.8	6.95	7.01	−0.06
50	9785.47	10801.39	4.09	3.70	0.38
80	10811.59	11716.92	3.70	3.41	0.29
200	12658.16	13746.31	3.16	2.91	0.25
1000	17073.72	18415.16	2.34	2.17	0.17

Table 4.7: Impact of Link Latency on Completion Time, 200 MB.

B/W	**Lat.**	**SW (secs)**		**QAT (secs)**		**Time Sav. (secs)**	
		Push	**Pull**	**Push**	**Pull**	**Push**	**Pull**
50	< 1	30.39	31.91	31.83	32.98	−1.44	−1.07
	50	32.79	33.16	34.87	34.13	−2.08	−0.97
	100	35.95	35.97	37.71	38.27	−1.76	−2.30
200	< 1	21.46	10.03	9.35	9.51	12.11	0.52
	50	24.42	13.46	11.94	12.51	12.47	0.94
	100	27.26	15.45	16.75	13.65	10.51	1.80

Table 4.8: Impact of Link Latency on Core Utilization, 200 MB.

B/W	**Lat.**	**SW (secs)**		**QAT (secs)**		**Core Sav. (%)**	
		Push	**Pull**	**Push**	**Pull**	**Push**	**Pull**
50	< 1	73.99	43.18	28.31	38.42	45.68	4.76
	50	73.19	40.75	38.07	35.96	35.12	4.79
	100	67.77	35.67	34.71	34.34	33.06	3.33
200	< 1	96.97	69.51	43.95	57.73	53.02	11.78
	50	87.57	56.04	47.27	45.17	40.3	10.87
	100	78.48	49.32	36.16	41.11	42.32	8.32

Table 4.9: Impact of Link Latency on Completion Time, 1 GB.

B/W	**Lat.**	**SW (secs)**		**QAT (secs)**		**Time Sav. (secs)**	
		Push	**Pull**	**Push**	**Pull**	**Push**	**Pull**
50	< 1	102.02	112.84	110.04	115.13	−8.02	−2.29
	50	104.79	116.32	113.02	117.74	−8.23	−1.42
	100	107.88	115.69	116.13	117.74	−8.25	−2.05
200	< 1	74.77	38.16	31.92	33.18	42.85	4.98
	50	77.90	39.55	33.94	34.57	43.96	4.97
	100	80.69	56.96	49.10	54.95	31.58	2.00

Table 4.10: Impact of Link Latency on Core Utilization, 1 GB.

B/W	**Lat.**	**SW (secs)**		**QAT (secs)**		**Core Sav. (%)**	
		Push	**Pull**	**Push**	**Pull**	**Push**	**Pull**
50	< 1	75.92	37.33	23.83	31.54	52.09	5.79
	50	78.74	37.34	39.82	32.66	38.92	4.68
	100	81.12	37.92	40.61	33.4	40.51	4.52
200	< 1	99.04	59.41	38.52	49.9	60.52	9.51
	50	95.58	56.90	50.37	47.46	45.21	9.44
	100	92.45	47.22	41.06	40.13	51.39	7.09

and 1 GB. The completion time of the 200 MB (actual [data+base layers] size = 419 MB) container size is 505.045/146.534 = 3.46 times of the completion time of the 1 GB (actual size = 1328 MB) container size. The ratio of actual container sizes is 1328/416 = 3.19, which indicates that the completion time for compression scales approximately linearly with the container size.

Comparing the completion times of SW and QAT for compression, we observe that QAT performs worse as compared to SW for low bandwidths (< 80 Mbps), and QAT starts performing better as the bandwidth increases (≥ 80 Mbps). For instance, with 10 Mbps bandwidth, the QAT completion time for the Docker push is $156.91 - 146.53 = 10.38$ seconds longer than the SW completion time for the 200 MB container size, and correspondingly $546.21 - 507.04 = 39.17$ seconds longer for the 1 GB container size. Whereas, with the 200 Mbps bandwidth, the QAT completion time is $30 - 3.71 = 17.29$ seconds faster (shorter) than the SW completion time for the 200 MB container size, and $74.77 - 31.92 = 42.85$ seconds faster for the 1 GB container size. Hence, hardware acceleration is more effective when the link bandwidth is sufficient enough to utilize the benefits of the hardware accelerator. Higher completion times with QAT as compared to SW for low bandwidth (< 80 Mbps) are associated with the overhead of memory copies and context switching, see Sec. 4.2.4.

The docker pull (decompression) completion times are presented in Figure 4.7(b) for the container sizes 200 MB and 1 GB. The completion times for Docker pull (decompression) with QAT are similar to Docker push (compression) in terms of hardware acceleration as a function of network bandwidth. Whereby, lower bandwidth results in worse performance, i.e., longer times completion for QAT as compared to SW. For instance, with 10 Mbps bandwidth, the QAT completion time for Docker pull is $158.498 - 148.852 = 9.646$ seconds higher than SW for the 200 MB container size, and $551.727 - 518.09 = 33.637$ seconds for the 1 GB container size.

In general, Docker push (compression) achieves shorter completion times than Docker push (decompression) because of the algorithmic complexity of compression. Hence, in general compression benefits more than decompression from hardware acceleration. However, a hardware accelerator roughly takes the same time for compression and decompression, whereas, SW takes longer for compression than for decompression. For example, Docker push (compression) at 1 Gbps takes 77.26

seconds with SW for the 1 GB container size, while it takes only 18 seconds with SW for Docker pull (decompression). Whereas, QAT takes nearly 13 seconds for both compression and decompression for the same execution. Therefore, to optimize the completion times to be lower, a hybrid approach of SW execution for slower network speeds, and hardware acceleration for higher network speeds appears prudent.

Core Utilization

Figs. 4.7(c) and (d) show the core utilization as a function of network bandwidth for QAT and SW based compression and decompression for container (data layer) sizes of 200 MB and 1 GB. The core utilization is the percentage of the core active time to the total execution duration, and core utilization savings is difference from SW to QAT. Due to the compute offloading to the hardware accelerator, we observe positive core utilization savings regardless of the network bandwidth and container sizes for both Docker push (compression) and Docker pull (decompression).

The margin of the core utilization savings is lowered as the network bandwidth gets lower. This is due to the fact that for the slower network speeds, the overall acceleration again is reduced when averaged over longer upload and download times. The CPU utilization is evaluated over the total duration of the Docker push/pull operations, which includes upload/download time as well as compression/decompression time. For example, for the 10 Mbps network speed, we see 11.92 and 1.2% of core utilization savings for the compression and decompression of the 1 GB container size, respectively. However, when the network speeds are increased, we observe higher core utilization savings, resulting from the balanced upload/download time relative to the compression/decompression times. For instance, when the network speed is 200 Mbps, we observe core utilization savings of 68.52 and 9.51% for the compression and decompression of the 1 GB container size, respectively.

When the network speeds are very high, i.e., 1 Gbps, the download/upload times are very short, and the CPU utilizations are dominated by the compression/ decompression operations. Whereby, during the compression/decompression accelerations, the core is actively engaged in hardware offloading operations, which results in higher core utilizations during the shorter (see Figs. 4.7(a) and (b)) duration of the Docker push/pull operations.

Overall, we observe that the effective core utilization savings from the hardware acceleration can vary based on the available network bandwidth.

Summary of Completion Times and Core Utilization

In Figs. 4.9(a) and 4.9(b) we present the overall gain summary of the completion times and core utilization as a function of the network bandwidth. To understand the impact of the QAT hardware acceleration on the Docker push (compression) and Docker (pull), we define the parameter Acceleration Gain as (1-(QAT/SW))$\times$100). A positive acceleration gain indicates that the QAT performs better than the SW, whereas a negative acceleration gain indicates the QAT performing worse than SW. From these charts, we can observe the following key characteristics of hardware acceleration relative to the network bandwidth.

i) For low network speeds, i.e., $<$ 80 Mbps, negative acceleration gains are observed for completion times of both Docker push (compression) and Docker pull (decompression). Gains improve as the network bandwidth increases, and we observe maximum acceleration gain in the completion times for 1 Gbps, and a marginal gain (nearly zero) for 80 Mbps.

ii) The core utilization exhibits positive gains for all evaluated network bandwidths, which shows that hardware accelerator can spare the core resources and thus free up core resources for general computing. For low network speeds, the core utilization gain is lower, but increases with the network bandwidth and then tapers off as the network bandwidth is further increased. We observe maximum core utilization gains for 80 Mbps.

iii) The gains for the 200 MB and 1 GB container sizes are very similar, whereby the behaviors of both completion time and core utilization are consistent with the container sizes. Hence, the behaviors indicate that the acceleration gain is independent of the container size.

Memory Access

Figures 4.7(e) and 4.7(f) show the DDR transaction rate [MB per second] from the IO devices (QAT, disk, network interface) and the CPU for both SW and QAT as a function of the network bandwidth for Docker push (compression) and Docker pull (decompression), for the 200 MB container size. Likewise, Figs. 4.7(g) and 4.7(h) show the transaction rate for the 1 GB container size. We observe a linear increase in the memory access rate with increasing network bandwidth which can be associated with the increase in the amount of data offloaded relative to total duration of Docker push/pull (compression/decompression) (whereby total duration = upload or download time + compression or decompression time). For instance, if the download time is short (for a high network bandwidth), the observed average memory access rate over the total duration is dominated by the hardware offload components whereby the QAT processing is much faster relative to the SW execution, thereby increasing the overall memory access rate.

For the container size of 200 MB, when the network bandwidth is 10 Mbps the CPU memory access rate is 56.6 MB/s and when the bandwidth is 200 Mbps, the

CPU memory access rate is 259.85 MB/s. In general, QAT IO and QAT CPU memory access rates are higher than the SW IO and SW CPU memory access rates, which can be associated with the fact that the QAT performs a higher number of operations per second, effectively reducing the overall completion time relative to SW. The behavior of memory access rates between container sizes of 200 MB (Figs. 4.7(e) and 4.7(f)) and 1 GB (Figs. 4.7(g) and 4.7(h)) are similar to each other, indicating that the memory access rates roughly remain the same for different container sizes. Likewise, the memory access rates of compression and decompression are similar, with decompression memory access rates being larger compared to compression. The maximum memory access rate is observed for decompression of the 200 MB container size with the network speed of 1 Gbps, resulting in a CPU memory access rate of 742.2 MB/s. This indicates that to effectively utilize the acceleration benefits, the DDR bandwidth of nearly 742.2 MB/s must be dedicated to the accelerator application.

Power Consumption

To understand the power implications of hardware acceleration, we present the total power consumption of Docker push (compression) and Docker pull (decompression) for both QAT and SW implementation. Tables 4.3 and 4.4 present the power consumption of QAT and SW for different network bandwidths for compression and decompression, respectively, for the container size of 200 MB. Likewise, Tables 4.5 and 4.6 present the power consumption for the 1 GB container size. With instantaneous power observed during the Docker operations, we estimate the total duration of battery savings that can be achieved with hardware acceleration relative to SW implementation based on the typical battery power source capacity of 40000 mWH. In general, QAT consumes less power when compared to SW for both compression and decompression for the different container sizes of 200 MB and 1 GB due to the compute offloading. For instance, when network bandwidth is 200 Mbps, QAT consumes 10446.22 mW of power for compression acceleration (i.e., the complete Docker push operation), while SW consumes 19436.41 mW which is nearly twice the QAT power consumption. However, for the same experiment but for decompression, we observe power consumption of 14531.14 mW for QAT and 15734.84 mW for SW showing a marginal improvement in consumption for the QAT. In terms of total power savings of battery (i.e., QAT − SW [Hour]), we observe 1.77 hours of savings for compression, whereas for decompression, we observe the savings of 0.21 hours.

We observe the following distinctive behavior in terms of power savings:

i) For low bandwidths and for compression, we observe high power savings because of increased opportunities for the core to sleep without having to offload or monitor the hardware accelerator very frequently.

ii) For decompression, the power savings is marginal, especially when the network bandwidth is low; this is commensurate with the lower decompression acceleration gain relative to compression. The power consumption is dominated by the overheads from the hardware offloading, which can be inferred from the 1 GB container size, where QAT shows negative power savings for the 10 Mbps network bandwidth in Table 4.6, indicating that the QAT consumes more power.

iii) As the bandwidth increases, the power savings improve because of more effective utilization of the hardware acceleration in terms of data availability to offload into the accelerator. For instance, for the 1 GB container size and compression operations, the battery savings is improved from 2.51 hours at 10 Mbps to 3.18 hours at 80 Mbps.

iv) However, when the bandwidth is further increased to 200 Mbps and 1 Gbps, the power savings start to reduce due to the large amount of instantaneous transactions relative to the total duration of the Docker push/pull operations due to the reduced upload/download time. This behavior is in line with the acceleration gains, as seen in Figs. 4.9(a) and 4.9(b), where the gain reduces as the bandwidth is increased to 1 Gbps.

Link Latency

The results showing the impact of the link latency on the hardware acceleration are summarized in Tables 4.7–4.10. Tables 4.7 and 4.8 present the completion times and CPU utilization as functions of the link latency for the container size of 200 MB; Tables 4.9 and 4.10 show the corresponding results for the 1 GB container size. We consider two network bandwidths, namely 50 and 200 Mbps, and vary the link latencies to < 1 (link default), 50, and 100 ms. In the evaluations, latency values indicate the unidirectional link latency such that 50 ms of latency indicates 100 ms of Round Trip Time (RTT).

In general, we observe that the completion times and core utilizations are relatively constant, with a minor increase in values as the link latency increases. For instance, when the network bandwidth is 50 Mbps, the completion times for Docker push (compression) are 30.39, 32.79, and 35.95 seconds for the < 1 (link default), 50, and 100 ms link latencies, respectively. The Docker push (compression) completion times for the 50 Mbps network bandwidth show negative acceleration time savings. When the link latency is modified while keeping other parameters constant, we observe that the negative savings are consistent across the different link latencies. In contrast, for the 1 GB container size and bandwidth of 200 Mbps, the acceleration time savings are consistently positive for all link latencies.

The core utilizations show positive acceleration gains (savings); we observe that these gains are consistent even as the link latency is modified. For instance, when the network bandwidth is 50 Mbps and the container size is 1 GB (Table 4.9), we observe core utilization savings of roughly 40% for all latency values for the Docker push operations. Whereas, for pull operations we observe CPU utilization savings in the typically in the 5–10% range. Overall, these results indicate that the end-to-end latency variations on resource-constrained devices do not strongly impact the overall acceleration gains.

4.5 Conclusions

We have quantified the performance of hardware acceleration for the compression and decompression of container images as required for container migration to and from client devices. Our extensive evaluations demonstrate that the Intel Quick Assist Technology (QAT) hardware acceleration of the container image compression and decompression achieves substantial speed-ups compared to software based compression (7 times) and decompression (1.6 times) with a local container registry. With a remote container registry, the compression gains reach 80% for high network bandwidths, while the decompression gains level out around 65% for moderately high network bandwidths. The QAT hardware acceleration also achieves some CPU core utilization reductions (even during the dramatically shorter time of utilizing the CPU) and has readily tolerable memory bandwidth consumption. Moreover, CPU memory access bitrates with QAT are approximately by factor of 9 higher than for GZIP compression and up to 3 times higher than for PIGZ decompression for a local container registry.

There are several important future research directions that can build on the results of this benchmark performance study of hardware accelerated container migration. One direction is to optimize the benefits of hardware acceleration with an orchestrator. The orchestrator could optimize the performance by dynamically and adaptively switching between execution on the CPU or on the hardware accelerators. The adaptive switched needs to satisfy the required task completion deadlines according to the quality of service stipulated by the application, under the constraints of available computing and energy resources at the client devices.

REFERENCES

Abbas, N., Y. Zhang, A. Taherkordi and T. Skeie, “Mobile edge computing: A survey”, IEEE Internet of Things Journal **5**, 1, 450–465 (2017).

Abdah, H., J. P. Barraca and R. L. Aguiar, “QoS-aware service continuity in the virtualized edge”, IEEE Access **7**, 51570–51588 (2019).

Abdallah, F., C. Tanougast, I. Kacem, C. Diou and D. Singer, “Genetic algorithms for scheduling in a CPU/FPGA architecture with heterogeneous communication delays”, Computers & Industrial Engineering **137**, 106006.1–106006.13 (2019).

Acevedo, J., R. Scheffel, S. Wunderlich, M. Hasler, S. Pandi, J. Cabrera, F. H. Fitzek, G. Fettweis and M. Reisslein, “Hardware acceleration for RLNC: A case study based on the Xtensa processor with the Tensilica instruction-set extension”, Electronics **7**, 9, 180 (2018).

Adamic, L. A. and B. A. Huberman, “Zipf’s law and the Internet”, Glottometrics **3**, 1, 143–150 (2002).

Addad, R. A., D. L. C. Dutra, M. Bagaa, T. Taleb and H. Flinck, “Fast service migration in 5G trends and scenarios”, IEEE Network **34**, 2, 92–98 (2020).

Adler, M., “PIGZ: Parallel GZIP”, (2014).

Advanced Micro Devices Inc., “AMD I/O Virtualization Technology (IOMMU) Specification, Rev. 3.00”, (2016).

Afolabi, I., T. Taleb, K. Samdanis, A. Ksentini and H. Flinck, “Network slicing and softwarization: A survey on principles, enabling technologies, and solutions”, IEEE Commun. Surveys & Tutorials **20**, 3, 2429–2453 (2018).

Ahmed, A. B. and A. B. Abdallah, “Adaptive fault-tolerant architecture and routing algorithm for reliable many-core 3D-NoC systems”, Journal of Parallel and Distributed Computing **93**, 30–43 (2016).

Akdemir, K., M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich and R. Zohar, “Breakthrough AES performance with Intel AES new instructions, Intel White Paper”, URL `https://software.intel.com/sites/default/files/m/d/4/1/d/8/10TB24_Breakthrough_AES_Performance_with_Intel_AES_New_Instructions.final.secure.pdf`, last accessed June 8, 2020 (2010).

Akhilesh, “Code base for a general discrete-event simulation framework for LayBack architecture”, URL `https://github.com/athyagat/layback.html` (2020).

Akhilesh S. Thyagaturu, V. G., “Efficient encryption in vpn sessions”, Patent, URL `https://patents.google.com/patent/US20210314359A1/en` (2021).

Akhilesh Thyagaturu, L. T., Hassnaa Moustafa, "Network flow-based hardware allocation", Patent, URL `https://patents.google.com/patent/US20210328933A1/en` (2021).

Akinaga, H. and H. Shima, "Resistive random access memory (ReRAM) based on metal oxides", Proceedings of the IEEE **98**, 12, 2237–2251 (2010).

Alberti, A. M., M. A. S. Santos, R. Souza, H. D. L. Da Silva, J. R. Carneiro, V. A. C. Figueiredo and J. J. P. C. Rodrigues, "Platforms for smart environments and future internet design: A survey", IEEE Access **7**, 165748–165778 (2019).

Aldwairi, M., T. Conte and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection", ACM SIGARCH Computer Architecture News **33**, 1, 99–107 (2005).

Alharbi, Z., A. S. Thyagaturu, M. Reisslein, H. ElBakoury and R. Zheng, "Performance comparison of r-phy and r-macphy modular cable access network architectures", IEEE Transactions on Broadcasting **64**, 1, 128–145 (2017).

Ali, A., G. A. Shah and J. Arshad, "Energy efficient resource allocation for m2m devices in 5g", Sensors **19**, 8, URL `http://www.mdpi.com/1424-8220/19/8/1830` (2019).

Alimi, I. A., A. L. Teixeira and P. P. Monteiro, "Toward an efficient C-RAN optical fronthaul for the future networks: A tutorial on technologies, requirements, challenges, and solutions", IEEE Communications Surveys & Tutorials **20**, 1, 708–769 (2018).

Allen, A. O., "Statistics and queuing theory with computer science applications, vol. 2", (1990).

Alsaeedi, M., M. M. Mohamad and A. A. Al-Roubaiey, "Toward adaptive and scalable OpenFlow-SDN flow control: A survey", IEEE Access **7**, 107346–107379 (2019).

Altman, E., K. Avrachenkov and S. Ramanath, "Multiscale fairness and its application to resource allocation in wireless networks", Comput. Commun. **35**, 7, 820–828 (2012).

Amazon Web Services, Inc., "AWS Graviton Processor", Https://aws.amazon.com/ec2/graviton/, last accessed June 2, 2020 (2020).

AMD, "ZEN Zeppelin block diagram", URL `https://pc.watch.impress.co.jp/video/pcw/docs/1108/270/p6.pdf`, last accessed Apr. 4, 2020 (2019).

AMD, "Zen Microarchitectures AMD", Https://en.wikichip.org/wiki/amd/microarchitectures/zen, last accessed June 2, 2020. (2020).

AMDl, "Infinity Fabric (IF) – AMD", URL `https://en.wikichip.org/wiki/amd/infinity_fabric`, last accessed June 2, 2020 (2020).

Ameigeiras, P., J. Ramos-Munoz, L. Schumacher, J. Prados-Garzon, J. Navarro-Ortiz and J. Lopez-Soler, “Link-level access cloud architecture design based on SDN for 5G networks”, IEEE Network **29**, 2, 24–31 (2015).

Amin, R., M. Reisslein and N. Shah, “Hybrid SDN networks: A survey of existing approaches”, IEEE Commun. Surv. & Tut. **20**, 4, 3259–3306 (2018).

Amit, N., M. Ben-Yehuda, D. Tsafrir and A. Schuster, “vIOMMU: efficient IOMMU emulation”, in “Proc. USENIX Ann. Techn. Conf. (ATC)”, pp. 73–86 (2011).

Amit, N., M. Ben-Yehuda and B.-A. Yassour, “Iommu: Strategies for mitigating the iotlb bottleneck”, in “International Symposium on Computer Architecture”, pp. 256–274 (Springer, 2010).

Anderson, J., H. Hu, U. Agarwal, C. Lowery, H. Li and A. Apon, “Performance considerations of network functions virtualization using containers”, in “Proc. IEEE Int. Conf. on Computing, Net. and Commun.”, pp. 1–7 (2016).

Andreades, P., K. Clark, P. M. Watts and G. Zervas, “Experimental demonstration of an ultra-low latency control plane for optical packet switching in data center networks”, Optical Switching and Networking **32**, 51–60 (2019).

Andrews, J., S. Singh, Q. Ye, X. Lin and H. Dhillon, “An overview of load balancing in HetNets: Old myths and open problems”, IEEE Wireless Communications **21**, 2, 18–25 (2014).

Ansaloni, G., P. Bonzini and L. Pozzi, “EGRA: A coarse grained reconfigurable architectural template”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **19**, 6, 1062–1074 (2010).

Arafa, M., B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman *et al.*, “Cascade Lake®: Next generation Intel Xeon® scalable processor”, IEEE Micro **39**, 2, 29–36 (2019).

Arena, F. and G. Pau, “An overview of vehicular communications”, Future Internet **11**, 2, URL `http://www.mdpi.com/1999-5903/11/2/27` (2019).

ARM Holdings, “AMBA AXI and ACE Protocol Specification”, URL `https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf`, last accessed June 9, 2020 (2019).

Arnold, O., S. Haas, G. P. Fettweis, B. Schlegel, T. Kissinger, T. Karnagel and W. Lehner, “HASHI: An Application Specific Instruction Set Extension for Hashing.”, in “ADMS@ VLDB”, pp. 25–33 (2014a).

Arnold, O., E. Matus, B. Noethen, M. Winter, T. Limberg and G. Fettweis, “Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs”, ACM Transactions on Embedded Computing Systems **13**, 3s, 107.1–107.24 (2014b).

Arnold, O., B. Noethen and G. Fettweis, “Instruction set architecture extensions for a dynamic task scheduling unit”, in “Proc. IEEE Computer Society Annual Symp. on VLSI”, pp. 249–254 (2012).

Auroux, S., M. Dräxler, A. Morelli and V. Mancuso, “Dynamic network reconfiguration in wireless DenseNets with the CROWD SDN architecture”, in “Proc. Eu. Conf. on Netw. and Commun. (EuCNC)”, pp. 144–148 (2015).

Bahena, V. R. and M. de Alba, “Embedded distributed systems: A case of study with Clear Linux Project for Intel® architecture”, Available from https://www.overleaf.com/articles/embedded-distributed-systems-a-case-of-study-with-clear-linux-projectfor-intel-r-architecture/wpcgjhykxdmj (2014).

Bahrami, B., M. A. J. Jamali and S. Saeidi, “A hierarchical architecture based on traveling salesman problem for hybrid wireless network-on-chip”, Wireless Networks **25**, 2187–2200 (2019).

Baktir, A. C., A. Ozgovde and C. Ersoy, “How can edge computing benefit from software-defined networking: A survey, use cases, and future directions”, IEEE Commun. Surveys & Tutorials **19**, 4, 2359–2391 (2017).

Balan, D. G. and D. A. Potorac, “Linux HTB queuing discipline implementations”, in “Proc. IEEE Int. Conf. on Networked Digital Techn.”, pp. 122–126 (2009).

Balasubramanian, V., M. Aloqaily and M. Reisslein, “An SDN architecture for time sensitive industrial IoT”, Computer Networks **186**, 107739 (2021a).

Balasubramanian, V., M. Aloqaily, M. Reisslein and A. Scaglione, “Intelligent resource management at the edge for ubiquitous IoT: An SDN-based federated learning approach”, IEEE Network **35**, 5, 114–121 (2021b).

Baldanzi, L., L. Crocetti, M. Bertolucci, L. Fanucci and S. Saponara, “Analysis of cybersecurity weakness in automotive in-vehicle networking and hardware accelerators”, in “Applications in Electronics Pervading Industry, Environment and Society, LNEE, Vol. 573”, pp. 11–18 (Springer, Cham, Switzerland, 2019).

Ball, M. O., T. Magnanti, C. Monma and G. Nemhauser, *Network Routing* (Elsevier, Amsterdam, 1995).

Ballani, H., P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas and G. O’Shea, “Enabling end-host network functions”, ACM SIGCOMM Computer Commun. Rev. **45**, 4, 493–507 (2015).

Banirazi, R., E. Jonckheere and B. Krishnamachari, “Heat diffusion algorithm for resource allocation and routing in multihop wireless networks”, in “Proc. IEEE GLOBECOM”, pp. 5693–5698 (2012).

Bao, W., C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello and P. Sadayappan, “Static and dynamic frequency scaling on multicore CPUs”, ACM Transactions on Architecture and Code Optimization (TACO) **13**, 4, 51:1–51:26 (2016).

Barach, D., L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, D. Rossi and J. Tollet, "Batched packet processing for high-speed software data plane functions", in "Proc. IEEE Conf. on Computer Commun. Wkshps", pp. 1–2 (2018).

Barakabitze, A. A., A. Ahmad, A. Hines and R. Mijumbi, "5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges", Computer Networks **167**, 1–40 (2020).

Bari, M. F., R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang and M. F. Zhani, "Data center network virtualization: A survey", IEEE Communications Surveys & Tutorials **15**, 2, 909–928 (2013).

Bartelt, J., N. Vucic, D. Camps-Mur, E. Garcia-Villegas, I. Demirkol, A. Fehske, M. Grieger, A. Tzanakaki, J. Gutiérrez, E. Grass, G. Lyberopoulos and G. Fettweis, "5G transport network requirements for the next generation fronthaul interface", EURASIP J. Wireless Commun. and Netw. **2017**, 89, 1–12 (2017).

Bashir, J., E. Peter and S. R. Sarangi, "A survey of on-chip optical interconnects", ACM Computing Surveys (CSUR) **51**, 6, 115.1–115.34 (2019).

Batmaz, B. and A. Doğan, "Hardware acceleration of FreeRTOS network stack for IoT edge devices", in "Proc. IEEE Int. Conf. on Electrical and Electronics Eng. (ELECO)", pp. 484–487 (2019).

Bauman, E., G. Ayoade and Z. Lin, "A survey on hypervisor-based monitoring: approaches, applications, and evolutions", ACM Computing Surveys (CSUR) **48**, 1, 1–33 (2015).

Beck, N., S. White, M. Paraschou and S. Naffziger, "'Zeppelin': An SoC for multichip architectures", in "Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)", pp. 40–42 (2018).

Begum, R., M. Hempstead, G. P. Srinivasa and G. Challen, "Algorithms for CPU and DRAM DVFS under inefficiency constraints", in "Proc. IEEE Int. Conf. on Comp. Design (ICCD)", pp. 161–168 (2016).

Bellavista, P., A. Corradi, L. Foschini and D. Scotece, "Differentiated service/data migration for edge services leveraging container characteristics", IEEE Access **7**, 139746–139758 (2019).

Belocchi, G., V. Cardellini, A. Cammarano and G. Bianchi, "Paxos in the NIC: Hardware acceleration of distributed consensus protocols", in "Proc. IEEE Int. Conf. on the Design of Reliable Communication Networks", pp. 1–6 (2020).

Ben Azzouz, L. and I. Jamai, "SDN, slicing, and NFV paradigms for a smart home: A comprehensive survey", Trans. on Emerging Telecommun. Techn. **30**, 10, e3744.1–e3744.13 (2019).

Ben-Yehuda, M., M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman and B.-A. Yassour, "The Turtles project: Design and implementation of nested virtualization.", in "Proc. USENIX Symp. on Operating Sys. Design and Impl. (OSDI)", vol. 10, pp. 423–436 (2010).

Benini, L., E. Flamand, D. Fuin and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator", in "Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)", pp. 983–987 (2012).

Bernal-Mor, E., V. Pla, J. Martinez-Bauset and D. Pacheco-Paramo, "A model of resource management in small cells with dynamic traffic and backhaul constraints", in "Proc. IEEE Eu. Wireless Conf. (EW)", pp. 1–6 (2013).

Bhamare, D., R. Jain, M. Samaka and A. Erbad, "A survey on service function chaining", J. Network and Computer Applications **75**, 138–155 (2016).

Biermann, T., L. Scalia, C. Choi, H. Karl and W. Kellerer, "CoMP clustering and backhaul limitations in cooperative cellular mobile access networks", Pervasive and Mobile Computing **8**, 5, 662–681 (2012).

Bikram Kumar, B., L. Sharma and S.-L. Wu, "Online distributed user association for heterogeneous radio access network", Sensors **19**, 6, URL `http://www.mdpi.com/1424-8220/19/6/1412` (2019).

Binsahaq, A., T. R. Sheltami and K. Salah, "A survey on autonomic provisioning and management of QoS in SDN networks", IEEE Access **7**, 73384–73435 (2019).

Blagodurov, S., A. Fedorova, S. Zhuravlev and A. Kamali, "A case for numa-aware contention management on multicore systems", in "2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)", pp. 557–558 (IEEE, 2010).

Blake, G., R. G. Dreslinski and T. Mudge, "A survey of multicore processors", IEEE Signal Processing Magazine **26**, 6, 26–37 (2009).

Blanco, B., J. O. Fajardo, I. Giannoulakis, E. Kafetzakis, S. Peng, J. Pérez-Romero, I. Trajkovska, P. S. Khodashenas, L. Goratti, M. Paolino, E. Sfakianakis, F. Liberal and G. Xilouris, "Technology pillars in the architecture of future 5G mobile networks: NFV, MEC and SDN", Computer Standards & Interfaces **54**, 216–228 (2017).

Blenk, A., A. Basta, M. Reisslein and W. Kellerer, "Survey on network virtualization hypervisors for software defined networking", IEEE Communications Surveys & Tutorials **18**, 1, 655–685 (2015a).

Blenk, A., A. Basta, J. Zerwas and W. Kellerer, "Pairing sdn with network virtualization: The network hypervisor placement problem", in "2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)", pp. 198–204 (IEEE, 2015b).

Blenk, A., A. Basta, J. Zerwas, M. Reisslein and W. Kellerer, “Control plane latency with sdn network hypervisors: The cost of virtualization”, IEEE Transactions on Network and Service Management **13**, 3, 366–380 (2016).

Bojkovic, Z., B. Bakmaz and M. Bakmaz, “Principles and enabling technologies of 5G network slicing”, in “Paving the Way for 5G Through the Convergence of Wireless Systems”, pp. 271–284 (IGI Global, Hershey, PA, 2019).

Bojnordi, M. N. and E. Ipek, “Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning”, in “Proc. IEEE Int. Symp. on High Perf. Computer Arch. (HPCA)”, pp. 1–13 (2016).

Bonfim, M. S., K. L. Dias and S. F. Fernandes, “Integrated NFV/SDN architectures: A systematic literature review”, ACM Computing Surveys (CSUR) **51**, 6, 114:1–114:39 (2019).

Bouras, C., A. Kollia and A. Papazois, “SDN & NFV in 5G: Advancements and challenges”, in “Proc. IEEE Conf. on Innov. in Clouds, Internet and Netw. (ICIN)”, pp. 107–111 (2017).

Bressoud, T. C. and F. B. Schneider, “Hypervisor-based fault tolerance”, in “Proceedings of the fifteenth ACM symposium on Operating systems principles”, pp. 1–11 (1995).

Broquedis, F., N. Furmento, B. Goglin, R. Namyst and P.-A. Wacrenier, “Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective”, in “Proc. Int. Workshop on OpenMP”, pp. 79–92 (Springer, Berlin, Heidelberg, Germany, 2009).

Bui, L. X., S. Sanghavi and R. Srikant, “Distributed link scheduling with constant overhead”, IEEE TON **17**, 5, 1467–1480 (2009).

Bulkan, U., T. Dagiuklas, M. Iqbal, K. M. S. Huq, A. Al-Dulaimi and J. Rodriguez, “On the load balancing of edge computing resources for on-line video delivery”, IEEE Access **6**, 73916–73927 (2018).

Bumgardner, V. C., V. W. Marek and C. D. Hickey, “Cresco: A distributed agent-based edge computing framework”, in “Proc. IEEE Int. Conf. Network and Service Management”, pp. 400–405 (2016).

Burr, G. W., R. S. Shenoy, K. Virwani, P. Narayanan, A. Padilla, B. Kurdi and H. Hwang, “Access devices for 3D crosspoint memory”, Journal of Vacuum Science & Technology B **32**, 4, 040802 (2014).

Casellas, R., R. Martínez, R. Vilalta and R. Muñoz, “Control, management, and orchestration of optical networks: Evolution, trends, and challenges”, IEEE/OSA Journal of Lightwave Technology **36**, 7, 1390–1402 (2018).

Catania, V., A. Mineo, S. Monteleone, M. Palesi and D. Patti, “Improving energy efficiency in wireless network-on-chip architectures”, ACM Journal on Emerging Technologies in Computing Systems (JETC) **14**, 1, 9.1–9.24 (2017).

Caulfield, A., E. Chung, A. Putnam, H. Angepat, J. Fowers, S. Heil, J. Kim, D. Lo, M. Papamichael, T. Massengill, D. Chiou and D. Burger, “A cloud-scale acceleration architecture”, IEEE Micro, in print pp. 1–1 (2020).

Caulfield, A., P. Costa and M. Ghobadi, “Beyond SmartNICs: Towards a fully programmable cloud”, in “Proc. IEEE Int. Conf. on High Perf. Switching and Routing (HPSR)”, pp. 1–6 (2018).

Cavaliere, F., P. Iovanna, J. Mangues-Bafalluy, J. Baranda, J. Núñez-Martínez, K.-Y. Lin, H.-W. Chang, P. Chanclou, P. Farkas, J. Gomes, L. Cominardi, A. Mourad, A. De La Oliva, J. A. Hernandez, D. Larrabeiti, A. DiGiglio, A. Paolicelli and P. Oedling, “Towards a unified fronthaul-backhaul data plane for 5G the 5G-Crosshaul project approach”, Comp. Standards & Interf. **51**, 56–62 (2017).

CCIX® Consortium Incorp., “An Introduction to CCIX® White Paper”, Https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf, Last accessed June 9, 2020 (2020).

Cen, Y., Y. Cen, K. Wang and J. Li, “Energy-efficient nonuniform content edge pre-caching to improve quality of service in fog radio access networks”, Sensors **19**, 6, URL `http://www.mdpi.com/1424-8220/19/6/1422` (2019).

Cepa, V., *Product-line development for mobile device applications with attribute supported containers*, Ph.D. thesis, TU Darmst. (2005).

Cerović, D., V. Del Piccolo, A. Amamou, K. Haddadou and G. Pujolle, “Fast packet processing: A survey”, IEEE Commun. Surv. & Tut. **20**, 4, 3645–3676 (2018).

Cerrato, I., M. Annarumma and F. Risso, “Supporting fine-grained network functions through Intel® DPDK”, in “Proc. IEEE EWSDN”, pp. 1–6 (2014).

Challa, N. R., “Hardware based i/o virtualization technologies for hypervisors, configurations and advantages-a study”, in “2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)”, pp. 1–5 (IEEE, 2012).

Chanclou, P., L. A. Neto, K. Grzybowski, Z. Tayq, F. Saliou and N. Genay, “Mobile fronthaul architecture and technologies: A RAN equipment assessment”, IEEE/OSA J. Opt. Commun. and Netw. **10**, 1, A1–A7 (2018).

Chandna, S., N. Naas and H. Mouftah, “Software defined survivable optical interconnect for data centers”, Optical Switching and Networking **31**, 86–99 (2019).

Chang, H.-C., B.-J. Qiu, J.-C. Chen, T.-J. Tan, P.-F. Ho, C.-H. Chiu and B.-S. P. Lin, “Empirical experience and experimental evaluation of open5gcore over hypervisor and container”, Wireless Communications and Mobile Computing **2018** (2018).

Chang, K. K., A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan and O. Mutlu, “Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization”, ACM SIGMETRICS Performance Evaluation Review **44**, 1, 323–336 (2016a).

Chang, K. K., P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi and O. Mutlu, "Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM", in "Proc. IEEE Int. Symp. on High Perf. Comp. Arch. (HPCA)", pp. 568–580 (2016b).

Charles, J., P. Jassi, N. S. Ananth, A. Sadat and A. Fedorova, "Evaluation of the intel® Core i7 Turbo Boost feature", in "Proc. IEEE Int. Symp. on Workload Charact. (IISWC)", pp. 188–197 (2009).

Chaudhary, J. K., J. Bartelt and G. Fettweis, "Statistical multiplexing in fronthaul-constrained massive MIMO", in "Proc. European Conf. on Networks and Commun. (EuCNC)", pp. 1–6 (2017).

Checko, A., H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger and L. Dittmann, "Cloud RAN for mobile networks—a technology overview", IEEE Communications Surveys & Tutorials **17**, 1, 405–426 (2015).

Chen, A., "A review of emerging non-volatile memory (NVM) technologies and applications", Solid-State Electronics **125**, 25–38 (2016).

Chen, T., Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning", ACM SIGARCH Computer Architecture News **42**, 1, 269–284 (2014).

Chen, X., W. Ni, T. Chen, I. Collings, X. Wang, R. P. Liu and G. B. Giannakis, "Multi-timescale online optimization of network function virtualization for service chaining", IEEE Trans. on Mobile Computing, in print (2019).

Chen, X., C. Wang, D. Xuan, Z. Li, Y. Min and W. Zhao, "Survey on QoS management of VoIP", in "Proc. IEEE Int. Conf. on Computer Netw. and Mobile Comp. (ICCNMC)", pp. 69–77 (2003).

Chen, Y., X. Wei, J. Shi, R. Chen and H. Chen, "Fast and general distributed transactions using RDMA and HTM", in "Proc. ACM Eu. Conf. on Computer Sys.", pp. 26.1–26.17 (2016).

Chhajer, S., A. S. Thyagaturu, A. Yatavelli, P. Lalwaney, M. Reisslein and K. G. Raja, "Hardware accelerations for container engine to assist container migration on client devices", in "Proc. IEEE Int. Symp. on Local and Metropolitan Area Networks (LANMAN", pp. 1–6 (2020).

Chiang, M., "Stochastic network utility maximization", Eur. T. on Telecommun. **22**, 1–22 (2008).

Chiang, M., S. H. Low, R. Calderbank and J. C. Doyle, "Layering as optimization decomposition", Proc. IEEE **95**, 255–312 (2007).

Chih-Lin, I., "Seven fundamental rethinking for next-generation wireless communications", APSIPA Trans. on Signal and Inform. Processing **6**, e10, 1–16 (2017).

Chih-Lin, I., H. Li, J. Korhonen, J. Huang and L. Han, “RAN revolution with NGFI (xHaul) for 5G”, IEEE/OSA Journal of Lightwave Technology **36**, 2, 541–550 (2018a).

Chih-Lin, I., H. Li, J. Korhonen, J. Huang and L. Han, “RAN revolution with NGFI (xhaul) for 5G”, Journal of Lightwave Technology **36**, 2, 541–550 (2018b).

Choi, H., D. Hong, J. Lee and S. Yoo, “Reducing DRAM refresh power consumption by runtime profiling of retention time and dual-row activation”, Microprocessors and Microsystems **72**, 102942.1–102942.1–11 (2020).

Choi, S., S. J. Park, M. Shahbaz, B. Prabhakar and M. Rosenblum, “Toward scalable replication systems with predictable tails using programmable data planes”, in “Proc. ACM Asia-Pacific Workshop on Networking”, pp. 78–84 (2019a).

Choi, S., M. Shahbaz, B. Prabhakar and M. Rosenblum, “λ-NIC: Interactive serverless compute on programmable smartnics”, arXiv preprint arXiv:1909.11958 (2019b).

Choi, Y.-K., J. Cong, Z. Fang, Y. Hao, G. Reinman and P. Wei, “A quantitative analysis on microarchitectures of modern CPU-FPGA platforms”, in “Proc. ACM Ann. Design Autom. Conf.”, pp. 1–6 (2016).

Chou, C.-H. and L. N. Bhuyan, “A multicore vacation scheme for thermal-aware packet processing”, in “Proc. IEEE Int. Conf. on Computer Design”, pp. 565–572 (2015).

Cilfone, A., L. Davoli, L. Belli and G. Ferrari, “Wireless mesh networking: An IoT-oriented perspective survey on relevant technologies”, Future Internet **11**, 4, URL `http://www.mdpi.com/1999-5903/11/4/99` (2019).

Clark, M., “A new $\times$86 core architecture for the next generation of computing.”, in “Proc. IEEEE Hot Chips Symp.”, pp. 1–19 (2016).

Clayman, S., E. Maini, A. Galis, A. Manzalini and N. Mazzocca, “The dynamic placement of virtual network functions”, in “Proc. IEEE Network Operations and Management Symp.”, pp. 1–9 (2014).

Cloutier, F., “CPU Identification”, Https://www.felixcloutier.com/x86/cpuid, last accessed June 2, 2020 (2019).

Cohen, A., F. Finkelstein, A. Mendelson, R. Ronen and D. Rudoy, “On estimating optimal performance of CPU dynamic thermal management”, IEEE Computer Architecture Letters **2**, 1, 6–6 (2003).

Coppola, M., M. D. Grammatikakis, R. Locatelli, G. Maruccia and L. Pieralisi, *Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC* (CRC Press, Boca Raton, FL, 2018).

Corporation, I. and D. Mulnix, “Intel® Xeon® Processor E5-2600 V4 Product Family Technical Overview”, Https://software.intel.com/en-us/articles/intel-xeon-processor-e5-2600-v4-product-family-technical-overview (2020).

Costa-Perez, X., A. Garcia-Saavedra, X. Li, T. Deiss, A. de la Oliva, A. di Giglio, P. Iovanna and A. Moored, “5G-Crosshaul: An SDN/NFV integrated fronthaul/backhaul transport network architecture”, IEEE Wireless Commun. **24**, 1, 38–45 (2017a).

Costa-Perez, X., A. Garcia-Saavedra, X. Li, T. Deiss, A. De La Oliva, A. Di Giglio, P. Iovanna and A. Moored, “5G-Crosshaul: An SDN/NFV integrated fronthaul/backhaul transport network architecture”, IEEE Wireless Communications **24**, 1, 38–45 (2017b).

Costanzo, S., I. Fajjari, N. Aitsaadi and R. Langar, “A network slicing prototype for a flexible cloud radio access network”, in “Proc. IEEE Consumer Commun. & Netw. Conf. (CCNC)”, pp. 1–4 (2018).

Coutinho, E. F., F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes and J. N. de Souza, “Elasticity in cloud computing: a survey”, Annals of Telecommunications **70**, 7-8, 289–309 (2015).

Cox, G., C. Dike and D. Johnston, “Intel’s digital random number generator (DRNG)”, in “Proc. IEEE Hot Chips Symp. (HCS)”, pp. 1–13 (2011).

Cox, J. H., J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley and H. L. Owen, “Advancing software-defined networks: A survey”, IEEE Access **5**, 25487–25526 (2017).

Cui, Y. and E. M. Yeh, “Delay optimal control and its connection to the dynamic backpressure algorithm”, in “IEEE Int. Symp. Info.”, pp. 451–455 (2014).

Cui, Y., E. M. Yeh and R. Liu, “Enhancing the delay performance of dynamic backpressure algorithms”, IEEE/ACM T. Netw. **24**, 2, 954–967 (2016).

CXL Consortium, “Compute Express Link™”, URL `https://www.computeexpresslink.org/download-the-specification`, last accessed June 9, 2020 (2019).

Cziva, R., S. Jouet, K. J. White and D. P. Pezaros, “Container-based network function virtualization for software-defined networks”, in “2015 IEEE symposium on computers and communication (ISCC)”, pp. 415–420 (IEEE, 2015).

Cziva, R. and D. P. Pezaros, “Container network functions: bringing NFV to the network edge”, IEEE Communications Magazine **55**, 6, 24–31 (2017).

Daoud, F., A. Watad and M. Silberstein, “GPUrdma: GPU-side library for high performance networking from GPU kernels”, in “Proc. ACM Int. Workshop on Runtime and Oper. Sys. for Supercomp.”, pp. 6.1–6.8 (2016).

Datta, D., D. Mittal, N. P. Mathew and J. Sairabanu, “Comparison of performance of parallel computation of CPU cores on CNN model”, in “Proc. IEEE Int. Conf. on Emerging Trends in Information Technology and Engineering”, pp. 1–8 (2020).

De Domenico, A., V. Savin and D. Ktenas, “A backhaul-aware cell selection algorithm for heterogeneous cellular networks”, in “Proc. IEEE Personal Indoor and Mobile Radio Commun.”, pp. 1688–1693 (2013).

de Sousa, N. F. S., D. A. L. Perez, R. V. Rosa, M. A. Santos and C. E. Rothenberg, “Network service orchestration: A survey”, Computer Communications **142-143**, 69–94 (2019).

Dehkordi, J. S. and V. Tralli, “Interference analysis for optical wireless communications in Network-on-Chip (NoC) scenarios”, IEEE Transactions on Communications **68**, 3, 1662–1674 (2019).

Desai, A., R. Oza, P. Sharma and B. Patel, “Hypervisor: A survey on concepts and taxonomy”, International Journal of Innovative Technology and Exploring Engineering **2**, 3, 222–225 (2013).

Deutsch, P., “RFC1952: GZIP file format specification version 4.3”, (1996).

Devera, M. and D. Cohen, “HTB linux queuing discipline manual-user guide”, M. Devera web site, Tech. Rep (2002).

Dinakar, K. R., “A survey on virtualization and attacks on virtual machine monitor (VMM)”, Int. Research J. of Eng. and Techn. (IRJET) **6**, 3, 6558–6563 (2019).

Doan, T. V., G. T. Nguyen, M. Reisslein and F. H. P. Fitzek, “FAST: Flexible and low-latency state transfer in mobile edge computing”, IEEE Access **9**, 115315–115334 (2021).

Dong, Y., D. Xu, Y. Zhang and G. Liao, “Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling”, in “2011 IEEE International Conference on Cluster Computing”, pp. 26–34 (IEEE, 2011).

Dong, Y., Z. Yu and G. Rose, “SR-IOV networking in Xen: Architecture, design and implementation.”, in “Proc. USENIX Workshop on I/O Virtualization (WIOV)”, pp. 1–10 (2008).

Dosanjh, M. G., W. Schonbein, R. E. Grant, P. G. Bridges, S. M. Ghazimirsaeed and A. Afsahi, “Fuzzy matching: Hardware accelerated MPI communication middleware”, in “Proc. IEEE/ACM Int. Symp. in Cluster, Cloud, and Grid Computing (CCGrid 2019)”, pp. 210–220 (2019).

Droste, H., P. Rost, M. Doll, I. Berberana, C. Mannweiler, M. Breitbach, A. Banchs and M. A. Puente, “An adaptive 5G multiservice and multitenant radio access network architecture”, Trans. Emerging Telecommun. Techn. **27**, 9, 1262–1270 (2016).

Duan, Q., N. Ansari and M. Toy, “Software-defined network virtualization: An architectural framework for integrating SDN and NFV for service provisioning in future networks”, IEEE Network **30**, 5, 10–16 (2016).

Durai, S., S. Raj and A. Manivannan, “Impact of thermal boundary resistance on the performance and scaling of phase change memory device”, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, in print (2020).

Elgendi, I., K. S. Munasinghe, D. Sharma and A. Jamalipour, “Traffic offloading techniques for 5G cellular: a three-tiered SDN architecture”, Annals of Telecommunications **71**, 11–12, 583–593 (2016).

Emmerich, P., D. Raumer, A. Beifuß, L. Erlacher, F. Wohlfart, T. M. Runge, S. Gallenmüller and G. Carle, “Optimizing latency and CPU load in packet processing systems”, in “Proc. IEEE Int. Symp. on Perf. Eval. of Computer and Telecommun. Sys. (SPECTS)”, pp. 1–8 (2015).

Eran, H., L. Zeno, M. Tork, G. Malka and M. Silberstein, “NICA: An infrastructure for inline acceleration of network applications”, in “Proc. USENIX Annual Techn. Conf.”, pp. 345–362 (2019).

Fan, W., Y. Liu, B. Tang, F. Wu and Z. Wang, “Computation offloading based on cooperations of mobile edge computing-enabled base stations”, IEEE Access **6**, 22622–22633 (2018).

Farhady, H., H. Lee and A. Nakao, “Software-defined networking: A survey”, Computer Networks **81**, 79–95 (2015).

Farooq, U., Z. Marrakchi and H. Mehrez, “FPGA architectures: An overview”, in “Tree-based Heterogeneous FPGA Architectures”, pp. 7–48 (Springer, New York, NY, 2012).

Farris, I., T. Taleb, Y. Khettab and J. Song, “A survey on emerging SDN and NFV security mechanisms for IoT systems”, IEEE Commun. Surveys & Tutorials **21**, 1, 812–837 (2018).

Fernandes, S., B. C. Oliveira and I. S. Silva, “Using NoC routers as processing elements”, in “Proc. ACM Symposium on Integrated Circuits and System Design”, pp. 1–6 (2009a).

Fernandes, S. R., B. Oliveira, M. Costa and I. S. Silva, “Processing while routing: A network-on-chip-based parallel system”, IET Computers & Digital Techniques **3**, 5, 525–538 (2009b).

Ferrari, L., N. Karakoc, A. Scaglione, M. Reisslein and A. Thyagaturu, “Layered cooperative resource sharing at a wireless SDN backhaul”, in “Proc. IEEE ICC Workshops”, pp. 1–6 (2018a).

Ferrari, L., N. Karakoc, A. Scaglione, M. Reisslein and A. Thyagaturu, “Layered cooperative resource sharing at a wireless SDN backhaul”, in “Proc. of IEEE Int. Conf. on Commun. Workshops (ICC Workshops), Int. Workshop on 5G Architecture (5GARCH)”, pp. 1–6 (2018b).

Ferrer, A. J., J. M. Marquès and J. Jorba, “Towards the decentralised cloud: Survey on approaches and challenges for mobile, ad hoc, and edge computing”, ACM Comp. Surv. **51**, 6, 1–36 (2019).

Fiessler, A., C. Lorenz, S. Hager, B. Scheuermann and A. W. Moore, “HyPaFilter+: Enhanced hybrid packet filtering using hardware assisted classification and header space analysis”, IEEE/ACM Transactions on Networking **25**, 6, 3655–3669 (2017).

Fleming, M., “A thorough introduction to eBPF”, Available from https://lwn.net/Articles/740157, Last accesed Apr. 2, 2020 (2017).

Foukas, X., G. Patounas, A. Elmokashfi and M. K. Marina, “Network slicing in 5G: Survey and challenges”, IEEE Communications Magazine **55**, 5, 94–100 (2017).

Frascolla, V., C. K. Dominicini, M. H. M. Paiva, G. Caporossi, M. A. Marotta, M. R. N. Ribeiro, M. E. V. Segatto, M. Martinello, M. E. Monteiro and C. B. Both, “Optimizing C-RAN backhaul topologies: A resilience-oriented approach using graph invariants”, Applied Sciences **9**, 1, URL `http://www.mdpi.com/2076-3417/9/1/136` (2019).

Gabriel, F., S. Wunderlich, S. Pandi, F. H. Fitzek and M. Reisslein, “Caterpillar RLNC with feedback (CRLNC-FB): Reducing delay in selective repeat ARQ through coding”, IEEE Access **6**, 44787–44802 (2018).

Gade, S. H. and S. Deb, “HyWin: Hybrid wireless NoC with sandboxed sub-networks for CPU/GPU architectures”, IEEE Transactions on Computers **66**, 7, 1145–1158 (2016).

Gade, S. H., S. S. Ram and S. Deb, “Millimeter wave wireless interconnects in deep submicron chips: Challenges and opportunities”, Integration **64**, 127–136 (2019).

Gallenmüller, S., P. Emmerich, F. Wohlfart, D. Raumer and G. Carle, “Comparison of frameworks for high-performance packet IO”, in “Proc. ACM/IEEE Symp. on Architectures for Net. and Commun. Systems”, pp. 29–38 (2015).

Garay, J., J. Matias, J. Unzilla and E. Jacob, “Service description in the NFV revolution: Trends, challenges and a way forward”, IEEE Communications Magazine **54**, 3, 68–74 (2016).

Garcia-Saavedra, A., X. Costa-Perez, D. J. Leith and G. Iosifidis, “FluidRAN: Optimized vRAN/MEC orchestration”, in “Proc. IEEE Infocom”, pp. 1–9 (2018).

Gazit, L. and H. Messer, “Advancements in the statistical study, modeling, and simulation of microwave-links in cellular backhaul networks”, Environments **5**, 7, URL `http://www.mdpi.com/2076-3298/5/7/75` (2018).

Ge, X., Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao and X. Hu, “OpenANFV: Accelerating network function virtualization with a consolidated framework in openstack”, ACM SIGCOMM Computer Communication Review **44**, 4, 353–354 (2014).

Ge, X., S. Tu, G. Mao, V. Lau and L. Pan, “Cost efficiency optimization of 5G wireless backhaul networks”, IEEE Transactions on Mobile Computing, in print (2019).

Gen, “The Gen-Z Consortium”, Https://genzconsortium.org, Last accessed June 9, 2020 (2020).

Georgiadis, L., M. J. Neely and L. Tassiulas, “Resource allocation and cross-layer control in wireless networks”, Found. Trends Netw. **1**, 1, 1–144, URL `http://dx.doi.org/10.1561/1300000001` (2006).

Gerszberg, I., K. X. Huang, C. K. Kwabi, J. S. Martin, I. R. R. Miller and J. E. Russell, “Network server platform for internet, JAVA server and video application server”, US Patent 6,044,403 (2000).

Ghosh, A., A. Maeder, M. Baker and D. Chandramouli, “5G evolution: A view on 5G cellular technology beyond 3GPP release 15”, IEEE Access **7**, 127639–127651 (2019).

Gilles, K., S. Groesbrink, D. Baldin and T. Kerstan, “Proteus hypervisor: Full virtualization and paravirtualization for multi-core embedded systems”, in “International Embedded Systems Symposium”, pp. 293–305 (Springer, 2013).

Giunta, G., R. Montella, G. Agrillo and G. Coviello, “A GPGPU transparent virtualization component for high performance computing clouds”, in “Proc. Eur. Conf. on Parallel Proc.”, pp. 379–391 (2010).

Gobriel, S., *Energy Efficiency in Communications and Networks* (IntechOpen, London, UK, 2012).

Goehringer, D., L. Meder, M. Hubner and J. Becker, “Adaptive multi-client network-on-chip memory”, in “Proc. IEEE Int. Conf. on Reconfig. Comp. and FPGAs”, pp. 7–12 (2011).

Gonzalez, C., E. Fluhr, D. Dreps, D. Hogenmiller, R. Rao, J. Paredes, M. Floyd, M. Sperling, R. Kruse, V. Ramadurai *et al.*, “3.1 POWER9™: A processor family optimized for cognitive computing with 25Gb/s accelerator links and 16Gb/s PCIe Gen4”, in “Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)”, pp. 50–51 (2017).

González, S., A. Oliva, X. Costa-Pérez, A. Di Giglio, F. Cavaliere, T. Deiß, X. Li and A. Mourad, “5G-Crosshaul: An SDN/NFV control and data plane architecture for the 5G integrated fronthaul/backhaul”, Trans. Emerg. Telecom. Techn. **27**, 9, 1196–1205 (2016).

Gowdal, N. M., X. Si and A. Sabharwall, “Full-duplex DOCSIS: A modem architecture for wideband ($>$ 1GHZ) self-interference cancellation for cable modem termination systems (CMTS)”, in “Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers”, pp. 2202–2206 (2018).

Goyal, S., “Public vs private vs hybrid vs community-cloud computing: a critical review”, IJCNIS **6**, 3, 20–29 (2014).

Granizo Arrabe, R., C. A. Platero, F. Alvarez Gomez and E. Rebollo Lopez, “New differential protection method for multiterminal HVDC cable networks”, Energies **11**, 12 (2018).

Gray, J., “GRVI Phalanx: A massively parallel RISC-V FPGA accelerator accelerator”, in “Proc. IEEE Int. Symp. on Field-Progr. Custom Comp. Mach. (FCCM)”, pp. 17–20 (2016).

Gu, H., K. Chen, Y. Yang, Z. Chen and B. Zhang, “MRONoC: A low latency and energy efficient on chip optical interconnect architecture”, IEEE Photonics Journal **9**, 1, 1–12 (2017).

Gu, L., Q. Tang, S. Wu, H. Jin, Y. Zhang, G. Shi, T. Lin and J. Rao, “N-Docker: A NVM-HDD hybrid Docker storage framework to improve docker performance”, in “Proc. IFIP Int. Conf. on Net. and Parallel Comp.”, pp. 182–194 (2019).

Guck, J. W., M. Reisslein and W. Kellerer, “Function split between delay-constrained routing and resource allocation for centrally managed QoS in industrial networks”, IEEE Transactions on Industrial Informatics **12**, 6, 2050–2061 (2016).

Guck, J. W., A. Van Bemten, M. Reisslein and W. Kellerer, “Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation”, IEEE Commun. Sur. & Tut. **20**, 1, 388–415 (2018).

Guerrieri, A., S. Kashani-Akhavan, M. Asiatici and P. Ienne, “Snap-on user-space manager for dynamically reconfigurable system-on-chips”, Ieee Access **7**, 103938–103947 (2019).

Gui, C.-Y., L. Zheng, B. He, C. Liu, X.-Y. Chen, X.-F. Liao and H. Jin, “A survey on graph processing accelerators: Challenges and opportunities”, Journal of Computer Science and Technology **34**, 2, 339–371 (2019).

Guo, L., Y. Ge, W. Hou, P. Guo, Q. Cai and J. Wu, “A novel IP-core mapping algorithm in reliable 3D optical network-on-chips”, Optical Switching and Networking **27**, 50–57 (2018).

Gupta, A. and R. K. Jha, “A survey of 5G network: Architecture and emerging technologies”, IEEE Access **3**, 1206–1232 (2015).

Gupta, A., X. Lin and R. Srikant, “Low-complexity distributed scheduling algorithms for wireless networks”, IEEE/ACM T. Netw. **17**, 6, 1846–1859 (2009).

Gupta, V., P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan and G. Srinivasa, “The forgotten 'uncore': On the energy-efficiency of heterogeneous cores”, in “Proc. USENIX Annual Techn. Conf. (ATC)”, pp. 367–372 (2012).

Gutiérrez, J., N. Maletic, D. Camps-Mur, E. García, I. Berberana, M. Anastasopoulos, A. Tzanakaki, V. Kalokidou, P. Flegkas, D. Syrivelis, T. Korakis, P. Legg, D. Markovic, G. Lyberopoulos, J. Bartelt, J. K. Chaudhary, M. Grieger, N. Vucic, J. Zou and E. Grass, “5G-XHaul: a converged optical and wireless solution for 5G transport networks”, Trans. Emer. Telecom. Techn. **27**, 9, 1187–1195 (2016).

Hamidouche, K. and M. LeBeane, “GPU initiated OpenSHMEM: Correct and efficient intra-kernel networking for dGPUs”, in “Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming”, pp. 336–347 (2020).

Han, B., V. Gopalakrishnan, L. Ji and S. Lee, “Network function virtualization: Challenges and opportunities for innovations”, IEEE Communications Magazine **53**, 2, 90–97 (2015).

Han, D., R.-H. Shiao and L. Xu, “Graceful shutdown with asynchronous DRAM refresh of non-volatile dual in-line memory module”, US Patent App. 15/261,397 (2018).

Haque, I. T. and N. Abu-Ghazaleh, “Wireless software defined networking: A survey and taxonomy”, IEEE Commun. Surv. & Tut. **18**, 4, 2713–2737 (2016).

Har’El, N., A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger and R. Ladelsky, “High performance I/O interposition in virtual systems”, in “USENIX Annual Technical Conference”, (2013).

Hassan, H., G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin and O. Mutlu, “ChargeCache: Reducing DRAM latency by exploiting row access locality”, in “Proc. IEEE Int. Symp. on High Perf. Comp. Arch. (HPCA)”, pp. 581–593 (2016).

Hassan, T. U. and F. Gao, “An active power control technique for downlink interference management in a two-tier macro–femto network”, Sensors **19**, 9 (2019).

Hennessy, J. L. and D. A. Patterson, “A new golden age for computer architecture.”, Commun. ACM **62**, 2, 48–60 (2019).

Herdt, V., D. Große, H. M. Le and R. Drechsler, “Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study”, in “Proc. ACM/IEEE Design Automation Conf.”, pp. 1–6 (2019).

Herrera, J. G. and J. F. Botero, “Resource allocation in NFV: A comprehensive survey”, IEEE Trans. on Network and Service Management **13**, 3, 518–532 (2016).

Heuchert, S., B. P. Rimal, M. Reisslein and Y. Wang, “Design of a small-scale and failure-resistant IaaS cloud using OpenStack”, Applied Computing and Informatics, in print (2022).

Hoeschele, T., C. Dietzel, D. Kopp, F. H. Fitzek and M. Reisslein, “Importance of internet exchange point (IXP) infrastructure for 5G: Estimating the impact of 5G use cases”, Telecommunications Policy **45**, 3, 102091 (2021).

Hofemeier, G. and R. Chesebrough, “Introduction to Intel AES-NI and Intel Secure Key Instructions, Intel Corp., White Paper”, URL `https://software.intel.com/en-us/articles/introduction-to-intel-aes-ni-and-intel-secure-key-instructions`, last accessed Apr. 3, 2020 (2012).

Hohlfeld, O., J. Krude, J. H. Reelfs, J. Rüth and K. Wehrle, “Demystifying the performance of XDP BPF”, in “Proc. IEEE Conf. on Net. Softwarization (NetSoft)”, pp. 208–212 (2019).

Høiland-Jørgensen, T., J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern and D. Miller, “The express data path: Fast programmable packet processing in the operating system kernel”, in “Proc. Inte. Conf. on Emerging Net. EXperiments and Techn.”, pp. 54–66 (2018).

Hovemeyer, D., J. K. Hollingsworth and B. Bhattacharjee, “Running on the bare metal with GeekOS”, ACM SIGCSE Bulletin **36**, 1, 315–319 (2004).

Hsu, Y., C.-Y. Chuang, X. Wu, G.-H. Chen, C.-W. Hsu, Y.-C. Chang, C.-W. Chow, J. Chen, Y.-C. Lai, C.-H. Yeh *et al.*, “2.6 Tbit/s on-chip optical interconnect supporting mode-division-multiplexing and PAM-4 signal”, IEEE Photonics Technology Letters **30**, 11, 1052–1055 (2018).

Huang, C., J. Zhang and T. Huang, “Updating data-center network with ultra-low latency data plane”, IEEE Access **8**, 2134–2144 (2020).

Huang, L., H. Gu, Y. Tian and T. Zhao, “Universal method for constructing the on-chip optical router with wavelength routing technology”, IEEE/OSA Journal of Lightwave Technology, in print (2020).

Huang, W., L. Ding, D. Meng, J. N. Hwang, Y. Xu and W. Zhang, “QoE-based resource allocation for heterogeneous multi-radio communication in software-defined vehicle networks”, IEEE Access **6**, 3387–3399 (2018a).

Huang, Y., C. Lu, M. Berg and P. Ödling, “Functional split of zero-forcing based massive MIMO for fronthaul load reduction”, IEEE Access **6**, 6350–6359 (2018b).

Huang, Y.-J., H.-H. Wu, Y.-C. Chung and W.-C. Hsu, “Building a KVM-based hypervisor for a heterogeneous system architecture compliant system”, in “Proc. ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments”, pp. 3–15 (2016).

Huang, Z., S. Wu, S. Jiang and H. Jin, “FastBuild: Accelerating Docker image building for efficient development and deployment of container”, in “Proc. Symp. on Mass Storage Sys. and Techn.”, pp. 28–37 (2019).

Huerfano, D., I. Demirkol and P. Legg, “Joint optimization of path selection and link scheduling for Millimeter Wave transport networks”, in “Proc. IEEE Int. Conf. on Commun. Workshops (ICC Workshops)”, pp. 115–120 (2017).

Intel, “Intel® Xeon® Processor Scalable Family Technical Overview”, URL `https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview`, last accessed June 2, 2020 (2019a).

Intel, “Overview – Claims and Benchmark Library”, URL `https://edc.intel.com/content/www/us/en/products/performance/benchmarks/overview.html` (2019b).

Intel Corp., "Intel® Data Direct I/O Technology (Intel DDIO) A Primer v1.0", Available from https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, Last accessed Apr. 2, 2020 (2012).

Intel Corp., "Data Plane Development Kit (DPDK)", (2014).

Intel Corp., "Intel® scalable I/O virtualization technical specification, reference number: 337679-001, revision: 1.0", URL `https://software.intel.com/sites/default/files/managed/cc/0e/intel-scalable-io-virtualization-technical-specification.pdf`, last accessed June 9, 2020 (2018).

Intel Corp., "Intel® Data Streaming Accelerator Preliminary Architecture Specification", URL `https://software.intel.com/sites/default/files/341204-intel-data-streaming- accelerator-spec.pdf`, last accessed June 9, 2020 (2019a).

Intel Corp., "Intel® Optane DC Persistent Memory - Content Delivery Networks Use Case", Https://builders.intel.com/docs/networkbuilders/intel-optane-dc-persistent-memory-content-delivery-networks-use-case.pdf, Last accessed June 9, 2020 (2019b).

Intel Corp., "Intel® QuickAssist Adapter 8960 and 8970 Product Brief", URL `https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/10-25-40-gigabit-adapters/quickassist-adapter-8960-8970-brief.html`, last accessed June 9, 2020 (2019c).

Intel Corp., "Intel® Resource Director Technology (Intel® RDT), Unlock System Performance in Dynamic Environments", URL `https://www.intel.com/content/www/us/en/architecture-and-technology /resource-director-technology.html`, last accessed June 9, 2020 (2019d).

Intel Corp., "White Paper: Accelerating High-Speed Networking with Intel® I/OAT", URL `https://www.intel.com/content/www/us/en/io/i-o-acceleration-technology -paper.html`, last accessed June 2, 2020 (2019e).

Intel Corp., "Intel oneAPI Product Brief", URL `https://software.intel.com/content/www/us/en/develop/download/oneapi-product-brief.html`, last accessed June 9, 2020 (2020a).

Intel Corp., "Intel Quickassist Technology Accelerator Abstraction Layer (AAL), White Paper, Platform-level Services for Accelerators", URL `https://blog-assets.oss-cn-shanghai.aliyuncs.com/18951/6103fadf4dd3a0dfbd0d637308a94b8e99e799d2.pdf`, last accessed June 9, 2020 (2020b).

Intel Corp., "Introduction to virtual machine extensions", Https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1043.html, Last accessed June 9, 2020 (2020c).

Intel Corp., "Virtual machine extension (VMX) operation", Https://software.intel.com/content/www/us/en/develop/documentation/debug-extensions-windbg-hyper-v-user-guide/top/virtual-machine-extension-vmx-operation.html, Last accessed June 9, 2020 (2020d).

Intel Corporation, "Clear linux* project", URL `https://clearlinux.org/about` (2020a).

Intel Corporation, "Configurable Spatial Accelerator (CSA)", Https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator, last accessed June 2, 2020 (2020b).

Intel Corporation, "Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration, UG-20136, 2020.05.11", Https://www.intel.com/content/www/us/en/programmable/documentation/tnc1513987819990.html, last accessed June 2, 2020 (2020c).

Intel Corporation, "Intel® Architecture Instruction Set Extensions and Future Features Programming Reference", Https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, Last accessed June 9, 2020 (2020d).

Jaber, M., M. A. Imran, R. Tafazolli and A. Tukmanov, "5G backhaul challenges and emerging research directions: A survey", IEEE Access **4**, 1743–1766 (2016).

Jagadeesan, N. A. and B. Krishnamachari, "Software-defined networking paradigms in wireless networks: A survey.", ACM Comput. Surv. **47**, 2, 27–1 (2014).

Jain, A. K., X. Li, P. Singhai, D. L. Maskell and S. A. Fahmy, "DeCO: A DSP block based FPGA accelerator overlay with low overhead interconnect", in "Proc. IEEE Int. Symp. on Field-Progr. Custom Comp. Mach. (FCCM)", pp. 1–8 (2016a).

Jain, P., S. J. Desai, M.-W. Shih, T. Kim, S. M. Kim, J.-H. Lee, C. Choi, Y. Shin, B. B. Kang and D. Han, "OpenSGX: An open platform for SGX research", in "Proc. Netw. and Distr. Sys. Security Symp. (NDSS)", (2016b).

Jain, R. and S. Paul, "Network virtualization and software defined networking for cloud computing: A survey", IEEE Communications Magazine **51**, 11, 24–31 (2013).

Jang, H. B., J. Lee, J. Kong, T. Suh and S. W. Chung, "Leveraging process variation for performance and energy: In the perspective of overclocking", IEEE Transactions on Computers **63**, 5, 1316–1322 (2012).

Jani, N., M. Deval, A. Singhai, P. Sarangam, M. Aggarwal, N. Parikh, A. H. Duyck, K. Patil, R. M. Sankaran, S. K. Kumar *et al.*, "Virtual device composition in a scalable input/output (I/O) virtualization (S-IOV) architecture", US Patent App. 16/211,941 (2019).

Jeddeloh, J. and B. Keeth, “Hybrid memory cube new DRAM architecture increases density and performance”, in “Proc. IEEE Symp. on VLSI Techn. (VLSIT)”, pp. 87–88 (2012).

Ji, B., C. Joo and N. Shroff, “Throughput-optimal scheduling in multihop wireless networks without per-flow information”, IEEE/ACM T. Netw. **21**, 2, 634–647 (2013).

Johansson, B., P. Soldati and M. Johansson, “Mathematical decomposition techniques for distributed cross-layer optimization of data networks”, IEEE J. Sel. Area. Comm. **24**, 1535–1547 (2006).

Junping, Z., K. Xu, M. Sohail and W. Cui, “Layer-aware data movement control for containers”, US Patent 10,659,533 (2020).

Kaljic, E., A. Maric, P. Njemcevic and M. Hadzialic, “A survey on data plane flexibility and programmability in Software-Defined Networking”, IEEE Access **7**, 47804–47840 (2019).

Kang, D., W. Jeong, C. Kim, D.-H. Kim, Y. S. Cho, K.-T. Kang, J. Ryu, K.-M. Kang, S. Lee, W. Kim *et al.*, “256 Gb 3 b/cell V-NAND flash memory with 48 stacked WL layers”, IEEE Journal of Solid-State Circuits **52**, 1, 210–217 (2016a).

Kang, K., S. Park, J.-B. Lee, L. Benini and G. De Micheli, “A power-efficient 3-D on-chip interconnect for multi-core accelerators with stacked L2 cache”, in “Proc. EDA Conf. on Design, Autom. & Test in Europe”, pp. 1465–1468 (2016b).

Kapre, N. and J. Gray, “Hoplite: Building austere overlay NoCs for FPGAs”, in “Proc. IEEE Int. Conf. on Field Progr. Logic and Appl.”, pp. 1–8 (2015).

Kar, K., X. Luo and S. Sarkar, “Throughput-optimal scheduling in multichannel access point networks under infrequent channel measurements”, IEEE T. Wirel. Commun. **7**, 7 (2008).

Kar, K., S. Sarkar, A. Ghavami and X. Luo, “Delay guarantees for throughput-optimal wireless link scheduling”, IEEE T. Automat. Contr. **57**, 11, 2906–2911 (2012).

Karakoc, N., A. Scaglione and A. Nedic, “Multi-layer decomposition of optimal resource sharing problems”, in “2018 IEEE Conf. on Dec. and Control (CDC)”, pp. 178–183 (2018).

Karakoç, N., A. Scaglione, A. Nedić and M. Reisslein, “Multi-layer decomposition of network utility maximization problems”, IEEE/ACM Transactions on Networking **28**, 5, 2077–2091 (2020).

Karakoç, N., A. Scaglione, M. Reisslein and R. Wu, “Federated edge network utility maximization for a multi- server system: Algorithm and convergence”, IEEE/ACM Transactions on Networking, in print (2022).

Karandikar, S., A. Ou, A. Amid, H. Mao, R. Katz, B. Nikolić and K. Asanović, "FirePerf: FPGA-accelerated full-system hardware/software performance profiling and co-design", in "Proc. ACM Int. Conf. on Arch. Support for Progr. Lang. and Operat. Sys.", pp. 715–731 (2020).

Karunaratne, M., A. K. Mohite, T. Mitra and L.-S. Peh, "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect", in "Proc. ACM/EDAC/IEEE Design Automation Conf. (DAC)", pp. 1–6 (2017).

Kaur, K., T. Dhand, N. Kumar and S. Zeadally, "Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers", IEEE Wireless Commun. **24**, 3, 48–56 (2017).

Kavanagh, A., "OpenStack as the API framework for NFV: The benefits, and the extensions needed", Ericsson Review **2015-3**, 1–8 (2015).

Kekely, M., L. Kekely and J. Kořenek, "General memory efficient packet matching FPGA architecture for future high-speed networks", Microproc. and Microsys. **73**, 102950.1–102950.12 (2020).

Kellerer, W., P. Kalmbach, A. Blenk, A. Basta, M. Reisslein and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges", Proc. IEEE **107**, 4, 711–731 (2019).

Kelly, F. P., A. K. Maulloo and D. K. H. Tan, "Rate control for communication networks: Shadow prices, proportional fairness and stability", The Journal of the Operational Research Society **49**, 3, 237–252 (1998).

Khan, S., D. Lee and O. Mutlu, "PARBOR: An efficient system-level technique to detect data-dependent failures in DRAM", in "Proc. IEEE/IFIP Int. Conf. on Dep. Systems and Netw. (DSN)", pp. 239–250 (2016).

Khan, S., C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee and O. Mutlu, "Detecting and mitigating data-dependent DRAM failures by exploiting current memory content", in "Proc. IEEE/ACM Int. Symp. on Microarchitecture", pp. 27–40 (2017).

Kim, H. J., H. Hirayama, S. Kim, K. J. Han, R. Zhang and J. W. Choi, "Review of near-field wireless power and communication for biomedical applications", IEEE Access **5**, 21264–21285 (2017).

Kim, M. and Y. S. Shao, "Hardware acceleration", IEEE Micro **38**, 6, 6–7 (2018).

King, D., A. Farrel, E. N. King, R. Casellas, L. Velasco, R. Nejabati and A. Lord, "The dichotomy of distributed and centralized control: METRO-HAUL, when control planes collide for 5G networks", Optical Switching and Networking **33**, 49–55 (2019).

Koehler, S., J. Curreri and A. D. George, "Performance analysis challenges and framework for high-performance reconfigurable computing", Parallel Computing **34**, 4-5, 217–230 (2008).

Kostal, K., R. Bencel, M. Ries, P. Truchly and I. Kotuliak, "High performance SDN WLAN architecture", Sensors **19**, 8, URL `http://www.mdpi.com/1424-8220/19/8/1880` (2019).

Kouchaksaraei, H. R. and H. Karl, "Service function chaining across OpenStack and Kubernetes domains", in "Proc. ACM Int. Conf, on Distr, and Event-based Sys.", pp. 240–243 (2019).

Kourtis, M.-A., G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis and F. Liberal, "Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration", in "Proc. IEEE Conf. on Network Function Virt. and Software Defined Netw. (NFV-SDN)", pp. 74–78 (2015).

Kozat, U. C., A. Xiang, T. Saboorian and J. Kaippallimalil, "The requirements and architectural advances to support URLLC verticals", in "5G Verticals: Customizing Applications, Technologies and Deployment Techniques", pp. 137–167 (Wiley, Hoboken, NJ, 2020).

Kratzke, N. and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study", Journal of Systems and Software **126**, 1–16 (2017).

Krzywda, J., A. Ali-Eldin, T. E. Carlson, P.-O. Östberg and E. Elmroth, "Power-performance tradeoffs in data center servers: DVFS, CPU pinning, horizontal, and vertical scaling", Future Generation Computer Systems **81**, 114–128 (2018).

Kumar, S., A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja and A. Hemani, "A network on chip architecture and design methodology", in "Proc. IEEE Comp. Soc. Ann. Symp. on VLSI (ISVLSI)", pp. 117–124 (2002).

Kutch, P., "PCI-SIG SR-IOV primer: An introduction to SR-IOV technology", (2011).

Lachaize, R., B. Lepers and V. Quéma, "Memprof: A memory profiler for {NUMA} multicore systems", in "Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)", pp. 53–64 (2012).

Lakshminarayana, S., M. Assaad and M. Debbah, "H-infinity control based scheduler for the deployment of small cell networks", Performance Eval. **70**, 7, 513–527 (2013).

Lal, S., T. Taleb and A. Dutta, "NFV: Security threats and best practices", IEEE Commun. Magazine **55**, 8, 211–217 (2017).

Lallet, J., A. Enrici and A. Saffar, "FPGA-based system for the acceleration of cloud microservices", in "Proc. IEEE Int. Symp. on Broadband Multi. Sys. and Broadcasting", pp. 1–5 (2018).

Landis, J. A., T. V. Powderly, R. Subrahmanian, A. Puthiyaparambil and J. R. Hunter Jr, "Computer system para-virtualization using a hypervisor that is implemented in a partition of the host system", US Patent 7,984,108 (2011).

Langley, A., A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC transport protocol: Design and Internet-scale deployment", in "Proc. Conf. ACM Special Interest Group on Data Communication", pp. 183–196 (2017).

Lauer, H. and N. Kuntze, "Hypervisor-based attestation of virtual environments", in "2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)", pp. 333–340 (IEEE, 2016).

Le, Y., H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift and T. Lakshman, "UNO: Uniflying host and smart NIC offload for flexible packet processing", in "Proc. ACM Symp. on Cloud Comp.", pp. 506–519 (2017).

Lee, D. U., K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29 nm process and TSV", in "Proc. IEEE Int. Solid-State Circuits Conf. Digest of Techn. Papers (ISSCC)", pp. 432–433 (2014).

Lee, V. W., C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU", ACM SIGARCH Comp. Arch. News **38**, 3, 451–460 (2010a).

Lee, Y., W. Kang and H. Son, "An Internet traffic analysis method with mapreduce", in "Proc. IEEE/IFIP Network Operations and Management Symposium Workshops", pp. 357–361 (2010b).

Leivadeas, A., G. Kesidis, M. Falkner and I. Lambadaris, "A graph partitioning game theoretical approach for the VNF service chaining problem", IEEE Trans. on Network and Service Management **14**, 4, 890–903 (2017).

Lepak, K., G. Talbot, S. White, N. Beck, S. Naffziger *et al.*, "The next generation AMD® enterprise server product architecture", in "Proc. IEEE Hot Chips", vol. 29, pp. 1–26 (2017).

Lettieri, G., V. Maffione and L. Rizzo, "A survey of fast packet I/O technologies for network function virtualization", in "Proc. Int. Conf. on High Performance Computing", pp. 579–590 (2017).

Li, A., S. L. Song, J. Chen, J. Li, X. Liu, N. Tallent and K. Barker, "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect", arXiv preprint arXiv:1903.04611 (2019a).

Li, B., K. Tan, L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng and E. Chen, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware", in "Proc. ACM SIGCOMM Conf.", pp. 1–14 (2016a).

Li, B., B. Yan and H. Li, "An overview of in-memory processing with emerging non-volatile memory for data-intensive applications", in "Proc. ACM Great Lakes Symp. on VLSI", pp. 381–386 (2019b).

Li, J., Z. Sun, J. Yan, X. Yang, Y. Jiang and W. Quan, "DrawerPipe: A reconfigurable pipeline for network processing on FPGA-Based SmartNIC", Electronics **9**, 1, 59.1–59.24 (2020).

Li, S., C. Xu, Q. Zou, J. Zhao, Y. Lu and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories", in "Proc. ACM Ann. Des. Autom. Conf.", pp. 1–6 (2016b).

Li, W., B. Guo, X. Li, Y. Zhou, S. Huang and G. N. Rouskas, "A large-scale nesting ring multi-chip architecture for manycore processor systems", Optical Switching and Networking **31**, 183–192 (2019c).

Li, W., Y. Zi, L. Feng, F. Zhou, P. Yu and X. Qiu, "Latency-optimal virtual network functions resource allocation for 5G backhaul transport network slicing", Applied Sciences **9**, 4 (2019d).

Li, X., R. Casellas, G. Landi, A. de la Oliva, X. Costa-Perez, A. Garcia-Saavedra, T. Deiss, L. Cominardi and R. Vilalta, "5G-Crosshaul network slicing: Enabling multi-tenancy in mobile transport networks", IEEE Commun. Mag. **55**, 8, 128–137 (2017).

Li, X. and C. Qian, "A survey of network function placement", in "Proc. IEEE Consumer Commun. & Netw. Conf. (CCNC)", pp. 948–953 (2016).

Li, Y. and M. Chen, "Software-defined network function virtualization: A survey", IEEE Access **3**, 2542–2553 (2015).

Liao, G., D. Guo, L. Bhuyan and S. R. King, "Software techniques to improve virtualized I/O performance on multi-core systems", in "Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems", pp. 161–170 (2008).

Lin, F., "Accelerate the Exploration of the Value of Genomic Data with Intel® QuickAssist Technology", URL `https://01.org/sites/default/files/downloads/intelr-quickassist-technology/336873-001crpesscasestudybgiquickassist.pdf` (2018).

Lin, X., N. B. Shroff and R. Srikant, "A tutorial on cross-layer optimization in wireless networks", IEEE J. Sel. Area. Comm. **24**, 8, 1452–1463 (2006).

Lin, X., N. B. Shroff and R. Srikant, "On the connection-level stability of congestion-controlled communication networks", IEEE T. Inform. Theory **54**, 5, 2317–2338 (2008).

Linguaglossa, L., S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel and G. Bianchi, "Survey of performance acceleration techniques for network function virtualization", Proc. of the IEEE **107**, 4, 746–764 (2019).

Linux Assembly, “Virtual machine extensions (VMX) instructions set”, Http://linasm.sourceforge.net/docs/instructions/vmx.php, Last accessed June 9, 2020 (2018).

Linux Foundation, “IO Visor Project”, URL `https://www.iovisor.org/`, last accessed Apr. 2, 2020 (2020).

Littley, M., A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng *et al.*, “Bolt: Towards a scalable Docker registry via hyperconvergence”, in “Proc. IEEE Cloud Comp.”, pp. 358–366 (2019).

Liu, J. and Q. Zhang, “Offloading schemes in mobile edge computing for ultra-reliable low latency communications”, IEEE Access **6**, 12825–12837 (2018).

Liu, J., S. Zhou, J. Gong, Z. Niu and S. Xu, “Statistical multiplexing gain analysis of heterogeneous virtual base station pools in cloud radio access networks”, IEEE Trans. Wireless Communications **15**, 8, 5681–5694 (2016a).

Liu, T., K. Wang, C. Ku and Y. Hsu, “QoS-aware resource management for multimedia traffic report systems over LTE-A”, Computer Networks **94**, 375–389 (2016b).

Lopez Rodriguez, F., U. Silva Dias, D. R. Campelo, R. d. Oliveira Albuquerque, S.-J. Lim and L. J. Garcia Villalba, “QoS management and flexible traffic detection architecture for 5G mobile networks”, Sensors **19**, 6 (2019).

Lotfollahi, M., M. J. Siavoshani, R. S. H. Zade and M. Saberian, “Deep packet: A novel approach for encrypted traffic classification using deep learning”, Soft Computing **24**, 3, 1999–2012 (2020).

Low, S. H. and D. E. Lapsley, “Optimization flow control. I. basic algorithm and convergence”, IEEE/ACM Trans. on Netw. **7**, 6, 861–874 (1999).

Lu, J., Y. Wan, Y. Li, C. Zhang, H. Dai, Y. Wang, G. Zhang and B. Liu, “Ultra-fast bloom filters using SIMD techniques”, IEEE Trans. on Parallel and Distr. Sys. **30**, 4, 953–964 (2018).

Lu, Z., Y. Wu, J. Xu and T. Wang, “An acceleration method for Docker image update”, in “Proc. IEEE Int. Conf. on Fog Comp.”, pp. 15–23 (2019).

Lucani, D. E., M. V. Pedersen, D. Ruano, C. W. Sørensen, F. H. Fitzek, J. Heide, O. Geil, V. Nguyen and M. Reisslein, “Fulcrum: Flexible network coding for heterogeneous devices”, IEEE Access **6**, 77890–77910 (2018).

Luong, N. C., P. Wang, D. Niyato, Y.-C. Liang, Z. Han and F. Hou, “Applications of economic and pricing models for resource management in 5G wireless networks: A survey”, IEEE Communications Surveys & Tutorials, in print (2019).

Luong, P., F. Gagnon, C. Despins and L.-N. Tran, “Joint virtual computing and radio resource allocation in limited fronthaul green C-RANs”, IEEE Trans. on Wireless Communications **17**, 4, 2602–2617 (2018).

Lyu, X., C. Ren, W. Ni, H. Tian, R. P. Liu and Y. J. Guo, "Multi-timescale decentralized online orchestration of software-defined networks", IEEE Journal on Selected Areas in Communications **36**, 12, 2716–2730 (2018).

Ma, L., S. Yi and Q. Li, "Efficient service handoff across edge servers via docker container migration", in "Proc. IEEE Symp. Edge Comp.", pp. 1–13 (2017).

Ma, Z., S. Shao, S. Guo, Z. Wang, F. Qi and A. Xiong, "Container migration mechanism for load balancing in edge network under power internet of things", IEEE Access **8**, 118405–118416 (2020).

Macri, J., "AMD's next generation GPU and high bandwidth memory architecture: FURY", in "Proc. IEEE Hot Chips Symp. (HCS)", pp. 1–26 (2015).

Mahajan, R., Z. Qian, R. S. Viswanath, S. Srinivasan, K. Aygün, W.-L. Jen, S. Sharan and A. Dhall, "Embedded multidie interconnect bridge–a localized, high-density multichip packaging interconnect", IEEE Trans. on Components, Packaging and Manufacturing Techn. **9**, 10, 1952–1962 (2019a).

Mahajan, R., R. Sankman, K. Aygun, Z. Qian, A. Dhall, J. Rosch, D. Mallik and I. Salama, "Embedded multi-die interconnect bridge (EMIB): A localized, high density, high bandwidth packaging interconnect", in "Advances in Embedded and Fan-Out Wafer-Level Packaging Technologies", pp. 487–499 (Wiley – IEEE Press, Hoboken, NJ, 2019b).

Majo, Z. and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead", in "Proceedings of the international symposium on Memory management", pp. 11–20 (2011).

Makineni, S. and R. Iyer, "Performance characterization of TCP/IP packet processing in commercial server workloads", in "Proc. IEEE Int. Conf. on Commun.", pp. 33–41 (2003).

Mao, Y., J. Zhang, S. Song and K. B. Letaief, "Stochastic joint radio and computational resource management for multi-user mobile-edge computing systems", IEEE Transactions on Wireless Communications **16**, 9, 5994–6009 (2017).

Marabissi, D., R. Fantacci and L. Simoncini, "SDN-based routing for backhauling in ultra-dense networks", Journal of Sensor and Actuator Networks **8**, 2, 1–23 (2019).

Mardani, M., G. Mateos and G. B. Giannakis, "Recovery of low-rank plus compressed sparse matrices with application to unveiling traffic anomalies", IEEE Transactions on Information Theory **59**, 8, 5186–5205 (2013).

Martí Luque, A., *Developing and deploying NFV solutions with OpenStack, Kubernetes and Docker*, Master's thesis, Universitat Politècnica de Catalunya (2019).

Martin, A., S. Raponi, T. Combe and R. Di Pietro, "Docker ecosystem–vulnerability analysis", Computer Commun. **122**, 30–43 (2018).

Marvell, "NITROX® III Security Processor Family", Https://www.marvell.com/products/security-solutions/nitrox-security-processors/nitrox-iii.html, Last accessed June 9, 2020 (2020).

Matthews, E. and L. Shannon, "Taiga: A new RISC-V soft-processor framework enabling high performance CPU architectural features", in "Proc. IEEE Int. Conf. on Field Progr. Logic and Appl. (FPL)", pp. 1–4 (2017).

Mayoral, A., R. Munoz, R. Vilalta, R. Casellas, R. Martinez and V. Lopez, "Need for a transport API in 5G for global orchestration of cloud and networks through a virtualized infrastructure manager and planner [invited]", IEEE/OSA J. Opt. Commun. and Netw. **9**, 1, A55–A62 (2017).

McDonnell, N. D., Z. Zhu and J. Mangan, "Power aware load balancing using a hardware queue manager", US Patent App. 16/131,728 (2019).

McGinnis, C., "PCI-SIG® Fast Tracks Evolution to 32 GT/s with PCI Express 5.0 Architecture", News Release, June **7** (2017).

Mechtri, M., C. Ghribi, O. Soualah and D. Zeghlache, "NFV orchestration framework addressing SFC challenges", IEEE Communications Magazine **55**, 6, 16–23 (2017).

Medhat, A. M., T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci and T. Magedanz, "Service function chaining in next generation networks: State of the art and research challenges", IEEE Communications Magazine **55**, 2, 216–223 (2017).

Mehrabi, M., S. Shen, Y. Hai, V. Latzko, G. P. Koudouridis, X. Gelabert, M. Reisslein and F. H. Fitzek, "Mobility-and energy-aware cooperative edge offloading for dependent computation tasks", Network **1**, 2, 191–214 (2021).

Mehrabi, M., D. You, V. Latzko, H. Salah, M. Reisslein and F. H. P. Fitzek, "Device-enhanced MEC: Multi-access edge computing (MEC) aided by end device computation and caching: A survey", IEEE Access **7**, 166079–166108 (2019).

Meng, N., M. Strassmaier and J. Erwin, "LS-DYNA® Performance on Intel® Scalable Solutions", in "Proc. Int. LS-DYNA® Users Conf.", pp. 1–8 (2018).

Merrifield, T. and H. R. Taheri, "Performance implications of extended page tables on virtualized x86 processors", ACM SIGPLAN Notices **51**, 7, 25–35 (2016).

Mharsi, N., M. Hadji, D. Niyato, W. Diego and R. Krishnaswamy, "Scalable and cost-efficient algorithms for baseband unit (BBU) function split placement", in "Proc. IEEE Wireless Commun. and Netw. Conf. (WCNC)", pp. 1–6 (2018).

Mijumbi, R., J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges", IEEE Commun. Surveys & Tutorials **18**, 1, 236–262 (2015).

Mikaeil, A. M., W. Hu, S. B. Hussain and A. Sultan, “Traffic-estimation-based low-latency XGS-PON mobile front-haul for small-cell C-RAN based on an adaptive learning neural network”, Applied Sciences **8**, 7, URL `http://www.mdpi.com/2076-3417/8/7/1097` (2018).

Miotto, G., M. C. Luizelli, W. L. da Costa Cordeiro and L. P. Gaspary, “Adaptive placement & chaining of virtual network functions with NFV-PEAR”, J. Internet Services and Appl. **10**, 1, 3.1–3.19 (2019).

Mnejja, S., Y. Aydi, M. Abid, S. Monteleone, V. Catania, M. Palesi and D. Patti, “Delta multi-stage interconnection networks for scalable wireless on-chip communication”, Electronics **9**, 6, 913.1–913.19 (2020).

Moby, “Moby project: An open framework to assemble specialized container systems without reinventing the wheel”, URL `https://mobyproject.org/` (2020).

Moon, Y., S. Lee, M. A. Jamshed and K. Park, “AccelTCP: Accelerating network applications with stateful TCP offloading”, in “Proc. USENIX Symp. on Netw. Systems Design and Impl. (NSDI)”, pp. 77–92 (2020).

Morabito, R., “A performance evaluation of container technologies on internet of things devices”, in “Proc. IEEE Infoc. Wkshp.”, pp. 999–1000 (2016).

Morais, L., V. Silva, A. Goldman, C. Alvarez, J. Bosch, M. Frank and G. Araujo, “Adding tightly-integrated task scheduling acceleration to a RISC-V multi-core processor”, in “Proc. IEEE/ACM Int. Symp. on Microarch.”, pp. 861–872 (2019).

Nadgowda, S., S. Suneja, N. Bila and C. Isci, “Voyager: Complete container state migration”, in “Proc. IEEE Int. Conf. Distr. Comp. Sys.”, pp. 2137–2142 (2017).

Nagaraj, D. and C. Gianos, “Intel® Xeon® processor D: The first Xeon processor optimized for dense solutions”, in “Proc. IEEE Hot Chips Symp. (HCS)”, pp. 1–22 (2015).

Nakajima, J., Q. Lin, S. Yang, M. Zhu, S. Gao, M. Xia, P. Yu, Y. Dong, Z. Qi, K. Chen *et al.*, “Optimizing virtual machines using hybrid virtualization”, in “Proceedings of the 2011 ACM Symposium on Applied Computing”, pp. 573–578 (2011).

Nakajima, Y., H. Masutani and H. Takahashi, “High-performance vNIC framework for hypervisor-based NFV with userspace vSwitch”, in “Proc. IEEE Eu. Workshop on Software Defined Netw.”, pp. 43–48 (2015).

Narmanlioglu, O., E. Zeydan and S. S. Arslan, “Service-aware multi-resource allocation in software-defined next generation cellular networks”, IEEE Access **6**, 20348–20363 (2018).

Nasrallah, A., A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein and H. ElBakoury, “Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research”, IEEE Communications Surveys & Tutorials **21**, 1, 88–145 (2018).

Nasrallah, A., A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein and H. Elbakoury, "Performance comparison of ieee 802.1 tsn time aware shaper (tas) and asynchronous traffic shaper (ats)", IEEE Access **7**, 44165–44181 (2019).

Neely, M. J., "Energy optimal control for time-varying wireless networks", IEEE Trans. Inf. Theor. **52**, 7, 2915–2934, URL `http://dx.doi.org/10.1109/TIT.2006.876219` (2006).

Neely, M. J., "Delay-based network utility maximization", IEEE/ACM T. Netw. **21**, 1, 41–54 (2013).

Nehama, D., R. Shiveley, J. Gasparakis and R. Love, "Developing High-Performance, Flexible SDN & NFV Solutions with Intel® Open Network Platform Server Reference Architecture", URL `http://docplayer.net/4394470-Developing-high-performance-flexible-sdn-nfv-solutions-with-intel-open-network-platform-server-reference-architecture.html`, last accessed June 9, 2020 (2014).

Neiger, G., A. Santoni, F. Leung, D. Rodgers and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization.", Intel Technology Journal **10**, 3, 167–178 (2006).

Neshatpour, K., M. Malik, A. Sasan, S. Rafatirad and H. Homayoun, "Hardware accelerated mappers for Hadoop MapReduce streaming", IEEE Trans. on Multi-Scale Computing Sys. **4**, 4, 734–748 (2018).

Nguyen, V., E. Tasdemir, G. T. Nguyen, D. E. Lucani, F. H. P. Fitzek and M. Reisslein, "DSEP Fulcrum: Dynamic sparsity and expansion packets for Fulcrum network coding", IEEE Access **8**, 78293–78314 (2020).

Nguyen, V.-G., A. Brunstrom, K.-J. Grinnemo and J. Taheri, "SDN/NFV-based mobile packet core network architectures: A survey", IEEE Commun. Surveys & Tutorials **19**, 3, 1567–1602 (2017).

Ni, Z., G. Liu, D. Afanasev, T. Wood and J. Hwang, "Advancing network function virtualization platforms with programmable NICs", in "Proc. IEEE Int. Symp. on Local and Metropolitan Area Netw. (LANMAN)", pp. 1–6 (2019).

Nie, J., C. Zhang, D. Zou, F. Xia, L. Lu, X. Wang and F. Zhao, "Adaptive sparse matrix-vector multiplication on CPU-GPU heterogeneous architecture", in "Proc. ACM High Perf. Comp. and Cluster Techn. Conf.", pp. 6–10 (2019).

Niemiec, G. S., L. M. S. Batista, A. E. Schaeffer-Filho and G. L. Nazar, "A survey on FPGA support for the feasible execution of virtualized network functions", IEEE Commun. Surveys & Tutorials **22**, 1, 504–525 (2020).

Niephaus, C., O. G. Aliu, M. Kretschmer, S. Hadzic and G. Ghinea, "Wireless Backhaul: a software defined network enabled wireless back-haul network architecture for future 5G networks", IET Networks **4**, 6, 287–295 (2015).

Niu, B., Y. Zhou, H. Shah-Mansouri and V. W. S. Wong, "A dynamic resource sharing mechanism for cloud radio access networks", IEEE Trans. Wirel. Comm. **15**, 12, 8325–8338 (2016).

Nobach, L. and D. Hausheer, "Open, elastic provisioning of hardware acceleration in NFV environments", in "Proc. IEEE Int. Conf. and Workshops on Networked Sys. (NetSys)", pp. 1–5 (2015).

Nunes, F. D. and M. E. Kreutz, "Using SDN strategies to improve resource management on a NoC", in "Proc. IFIP/IEEE Int. Conf. on Very Large Scale Integration (VLSI-SoC)", pp. 224–225 (2019).

Nurvitadhi, E., J. Sim, D. Sheffield, A. Mishra, S. Krishnan and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC", in "Proc. IEEE Int. Conf. on Field Progr. Logic and Appl. (FPL)", pp. 1–4 (2016).

NVidia Fermi, "Nvidia's next generation CUDA compute architecture", NVidia, Santa Clara, CA (2009).

Ogawa, H., G. K. Tran, K. Sakaguchi and T. Haustein, "Traffic adaptive formation of mmWave meshed backhaul networks", in "Proc. IEEE ICC Workshops", pp. 185–191 (2017).

Ogudo, K. A., D. Muwawa Jean Nestor, O. Ibrahim Khalaf and H. Daei Kasmaei, "A device performance and data analytics concept for smartphones' iot services and machine-type communication in cellular networks", Symmetry **11**, 4, URL `http://www.mdpi.com/2073-8994/11/4/593` (2019).

Oliva, L., A. De, X. C. Perez, A. Azcorra, A. D. Giglio, F. Cavaliere, D. Tiegelbekkers, J. Lessmann, T. Haustein, A. Mourad and P. Iovanna, "Xhaul: toward an integrated fronthaul/backhaul architecture in 5G networks", IEEE Wireless Commun. **22**, 5, 32–40 (2015).

Ordonez-Lucena, J., P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca and J. Folgueira, "Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges", IEEE Communications Magazine **55**, 5, 80–87 (2017).

OS^v, "NFV-optimized OS", URL `http://osv.io/nfv` (2020).

Ouyang, Y., Q. Wang, M. Ru, H. Liang and J. Li, "A novel low-latency regional fault-aware fault-tolerant routing algorithm for wireless NoC", IEEE Access **8**, 22650–22663 (2020).

Owaida, M., G. Alonso, L. Fogliarini, A. Hock-Koon and P.-E. Melet, "Lowering the latency of data processing pipelines through FPGA based hardware acceleration", Proc. of the VLDB Endowment **13**, 1, 71–85 (2019).

Ozdal, M. M., S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns and O. Ozturk, "Energy efficient architecture for graph analytics accelerators", in "Proc. ACM/IEEE Int. Symp. on Comp. Arch. (ISCA)", pp. 166–177 (2016).

Palomar, D. P. and M. Chiang, “A tutorial on decomposition methods for network utility maximization”, IEEE Journal on Selected Areas in Communications **24**, 8, 1439–1451 (2006).

Pan, C. and A. Naeemi, “A fast system-level design methodology for heterogeneous multi-core processors using emerging technologies”, IEEE Journal on Emerging and Selected Topics in Circuits and Systems **5**, 1, 75–87 (2015).

Papadimitriou, G., M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers and S. Das, “Harnessing voltage margins for energy efficiency in multicore CPUs”, in “Proc. IEEE/ACM Int. Symp. on Microarch.”, pp. 503–516 (2017).

Park, D., A. Vaidya, A. Kumar and M. Azimi, “MoDe-X: Microarchitecture of a layout-aware modular decoupled crossbar for on-chip interconnects”, IEEE Transactions on Computers **63**, 3, 622–636 (2012).

Parvez, I., A. Rahmati, I. Guvenc, A. I. Sarwat and H. Dai, “A survey on low latency towards 5G: RAN, core network and caching solutions”, IEEE Commun. Surv. & Tut. **20**, 4, 3098–3130 (2018).

Pateromichelakis, E., J. Gebert, T. Mach, J. Belschner, W. Guo, N. P. Kuruvatti, V. Venkatasubramanian and C. Kilinc, “Service-tailored user-plane design framework and architecture considerations in 5G radio access networks”, IEEE Access **5**, 17089–17105 (2017).

Pattaranantakul, M., R. He, Q. Song, Z. Zhang and A. Meddahi, “NFV security survey: From use case driven threat analysis to state-of-the-art countermeasures”, IEEE Commun. Surveys & Tutorials **20**, 4, 3330–3368 (2018).

Pattaranantakul, M., Q. Song, Y. Tian, L. Wang, Z. Zhang and A. Meddahi, “Footprints: Ensuring trusted service function chaining in the world of SDN and NFV”, in “Proc. Int. Conf. on Security and Privacy in Commun. Sys.”, pp. 287–301 (Springer, Cham, Switzerland, 2019).

Pauls, F., R. Wittig and G. Fettweis, “A latency-optimized hash-based digital signature accelerator for the tactile internet”, in “Proc. Int. Conf. on Embedded Computer Systems, Lecture Notes in Computer Science, vol 11733”, pp. 93–106 (Springer, Cham, Switzerland, 2019).

Pellegrini, A., N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala *et al.*, “The Arm Neoverse N1 platform: Building blocks for the next-gen cloud-to-edge infrastructure SoC”, IEEE Micro **40**, 2, 53–62 (2020).

Perez-Botero, D., J. Szefer and R. B. Lee, “Characterizing hypervisor vulnerabilities in cloud computing servers”, in “Proceedings of the 2013 international workshop on Security in cloud computing”, pp. 3–10 (2013).

Pham, Q.-V., H.-L. To and W.-J. Hwang, “A multi-timescale cross-layer approach for wireless ad hoc networks”, Comput. Netw. **91**, 471–482 (2015).

Pitaev, N., M. Falkner, A. Leivadeas and I. Lambadaris, "Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, FD.IO VPP and SR-IOV", in "Proc. ACM/SPEC Int. Conf. on Perform. Eng.", pp. 285–292 (2018).

Plouffe, J., S. H. Davis, A. D. Vasilevsky, B. J. Thomas III, S. S. Noyes and T. Hazel, "Distributed virtual machine monitor for managing multiple virtual resources across multiple physical nodes", US Patent 8,776,050 (2014).

Pongrácz, G., L. Molnár and Z. L. Kis, "Removing roadblocks from SDN: Openflow software switch performance on Intel DPDK", in "Proc. IEEE Eu. Workshop on Software Defined Netw.", pp. 62–67 (2013).

Pontarelli, S., M. Bonola and G. Bianchi, "Smashing OpenFlow's "atomic" actions: Programmable data plane packet manipulation in hardware", Int. Journal of Netw. Management **29**, 1, e2043.1–e2043.20 (2019).

Power, N., S. Harte, N. D. McDonnell and A. Cunningham, "System, apparatus and method for real-time activated scheduling in a queue management device", US Patent App. 15/719,769 (2019).

Prasad, N., M. Arslan and S. Rangarajan, "A two time scale approach for coordinated multi-point transmission and reception over practical backhaul", in "Proc. Int. Conf. on Commun. Sys. and Netw. (COMSNETS)", pp. 1–8 (2014).

Qadir, J., N. Ahmed and N. Ahad, "Building programmable wireless networks: an architectural survey", EURASIP Journal on Wireless Communications and Networking **2014**, 1, 1–31 (2014).

qatzip, "Intel® QuickAssist Technology (QAT) QATzip Library", URL `https://github.com/intel/QATzip` (2020).

Raj, H. and K. Schwan, "High performance and scalable I/O virtualization via self-virtualized devices", in "Proceedings of the 16th international symposium on High performance distributed computing", pp. 179–188 (2007).

Ramirez-Perez, C. and V. Ramos, "SDN meets SDR in self-organizing networks: Fitting the pieces of network management", IEEE Commun. Mag. **54**, 1, 48–57 (2016).

Randazzo, A. and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way", in "2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)", pp. 209–214 (IEEE, 2019).

Rao, J., K. Wang, X. Zhou and C.-Z. Xu, "Optimizing virtual machine scheduling in numa multicore systems", in "2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)", pp. 306–317 (IEEE, 2013).

Raparti, V. Y., N. Kapadia and S. Pasricha, "ARTEMIS: An aging-aware runtime application mapping framework for 3D NoC-based chip multiprocessors", IEEE Transactions on Multi-Scale Computing Systems **3**, 2, 72–85 (2017).

Ray, J., V. George, I. M. Sodhi and J. R. Wilcox, "Common platform for one-level memory architecture and two-level memory architecture", US Patent 9,600,413 (2017).

Regula, J., "Using non-transparent bridging in PCI Express systems, White Paper", URL `https://docs.broadcom.com/doc/12353428`, last accessed Apr. 4, 2020 (2004).

Rehman, A. U., R. L. Aguiar and J. P. Barraca, "Network functions virtualization: The long road to commercial deployments", IEEE Access **7**, 60439–60464 (2019).

Rehmani, M. H., A. Davy, B. Jennings, Z. Kaleem, A. S. Thyagaturu, H. Moustafa and A.-S. K. Pathan, "Ieee access special section editorial: Software-defined networks for energy internet and smart grid communication", IEEE Access **9**, 69139–69142 (2021).

Reis, D., J. Takeshita, T. Jung, M. Niemier and X. S. Hu, "Computing-in-memory for performance and energy efficient homomorphic encryption", arXiv preprint arXiv:2005.03002 (2020).

Reza, A., "Online multi-application mapping in photonic Network-on-Chip with mesh topology", Optical Switching and Networking **25**, 100–108 (2017).

Ricart-Sanchez, R., P. Malagon, A. Matencio-Escolar, J. M. Alcaraz Calero and Q. Wang, "Toward hardware-accelerated QoS-aware 5G network slicing based on data plane programmability", Trans. on Emerging Telecommun. Techn. **31**, 1, e3726.1–e3726.19 (2020).

Richart, M., J. Baliosian, J. Serrat and J. L. Gorricho, "Resource slicing in virtual wireless networks: A survey", IEEE Trans. on Network and Service Management **13**, 3, 462–476 (2016a).

Richart, M., J. Baliosian, J. Serrat and J.-L. Gorricho, "Resource slicing in virtual wireless networks: A survey", IEEE Trans. on Network and Service Management **13**, 3, 462–476 (2016b).

Rischke, J., P. Sossalla, S. Itting, F. H. P. Fitzek and M. Reisslein, "5G campus networks: A first measurement study", IEEE Access **9**, 121786–121803 (2021).

Rodrigues, J. J. P. C., D. B. D. R. Segundo, H. A. Junqueira, M. H. Sabino, R. M. Prince, J. Al-Muhtadi and V. H. C. D. Albuquerque, "Enabling technologies for the Internet of Health Things", IEEE Access **6**, 13129–13141 (2018).

Rodriguez, M. A. and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions", Software: Practice and Experience **49**, 5, 698–719 (2019).

Rooney, R. and N. Koyle, “Introducing Micron® DDR5 SDRAM: More Than a Generational Update”, URL `https://www.micron.com/-/media/client/global/documents/products/white-paper/ddr5_new_features_white_paper.pdf`, last accessed June 9, 2020 (2019).

Roseboro, R., “Cloud-native NFV architecture for agile service creation & scaling”, Https://www.mellanox.com/related-docs/whitepapers/wp-heavyreading-nfv-architecture-for-agile-service.pdf, Last accessed May 19, 2020 (2016).

Ross, K. W., *Multiservice Loss Models for Broadband Telecommunication Networks* (Springer Science, London, UK, 1995).

Ruaro, M., N. Velloso, A. Jantsch and F. G. Moraes, “Distributed SDN architecture for NoC-based many-core SoCs”, in “Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip”, pp. 1–8 (2019).

Ruiz, M., D. Sidler, G. Sutter, G. Alonso and S. López-Buedo, “Limago: An FPGA-based open-source 100 GbE TCP/IP Stack”, in “Proc. IEEE Int. Conf. on Field Progr. Logic and Appl.”, pp. 286–292 (2019).

Russell, R., “virtio: towards a de-facto standard for virtual i/o devices”, ACM SIGOPS Operating Systems Review **42**, 5, 95–103 (2008).

Sachs, J., G. Wikstrom, T. Dudda, R. Baldemair and K. Kittichokechai, “5G radio network design for ultra-reliable low-latency communication”, IEEE Network **32**, 2, 24–31 (2018).

sagroups, “Next generation fronthaul interface (1914) working group”, URL `http://sites.ieee.org/sagroups-1914` (1914).

Sailer, R., E. Valdez, T. Jaeger, R. Perez, L. Van Doorn, J. L. Griffin, S. Berger, R. Sailer, E. Valdez, T. Jaeger *et al.*, “sHype: Secure hypervisor approach to trusted virtualized systems”, Techn. Rep. RC23511 **5** (2005).

Samdanis, K., R. Shrivastava, A. Prasad, D. Grace and X. Costa-Perez, “TD-LTE Virtual Cells: An SDN architecture for user-centric multi-eNB elastic resource management”, Computer Commun. **83**, 1–15 (2016).

Sandoval-Arechiga, R., R. Parra-Michel, J. Vazquez-Avila, J. Flores-Troncoso and S. Ibarra-Delgado, “Software defined networks-on-chip for multi/many-core systems: A performance evaluation”, in “Proc. ACM Symp. on Arch. for Netw. and Commun. Sys.”, pp. 129–130 (2016).

Santa, J., P. J. Fernandez, J. Ortiz, R. Sanchez-Iborra and A. F. Skarmeta, “SURROGATES: Virtual OBUs to foster 5G vehicular services”, Electronics **8**, 2, URL `http://www.mdpi.com/2079-9292/8/2/117` (2019).

Scarpiniti, M., E. Baccarelli and A. Momenzadeh, "VirtFogSim: A parallel toolbox for dynamic energy-delay performance testing and optimization of 5G mobile-fog-cloud virtualized platforms", Applied Sciences **9**, 6, URL `http://www.mdpi.com/2076-3417/9/6/1160` (2019).

Scheepers, M. J., "Virtualization and containerization of application infrastructure: A comparison", in "Proc. Stu. Conf. IT", pp. 1–7 (2014).

Scheitle, Q., T. Chung, J. Amann, O. Gasser, L. Brent, G. Carle, R. Holz, J. Hiller, J. Naab, R. van Rijswijk-Deij, O. Hohlfeld, D. Choffnes and A. Mislove, "Measuring adoption of security additions to the HTTPS ecosystem", in "Proc. of the Applied Net. Research Workshop", pp. 1–2 (2018).

Schone, R., D. Hackenberg and D. Molka, "Memory performance at reduced CPU clock speeds: An analysis of current x86_64 processors", in "Proc USENIX Workshop on Power-Aware Computing and Systems", pp. 1–5 (2012).

Scionti, A., S. Mazumdar and A. Portero, "Software defined network-on-chip for scalable CMPs", in "Proc. IEEE Int. Conf. on High Perf. Comp. & Simul. (HPCS)", pp. 112–115 (2016).

Scionti, A., S. Mazumdar and A. Portero, "Towards a scalable software defined network-on-chip for next generation cloud", Sensors **18**, 7, 2330.1–2330.24 (2018).

Sebastian, A., M. Le Gallo, R. Khaddam-Aljameh and E. Eleftheriou, "Memory devices and applications for in-memory computing", Nature Nanotechnology pp. 1–16 (2020).

Semiari, O., W. Saad, S. Valentin, M. Bennis and H. Vincent Poor, "Context-aware small cell networks: How social metrics improve wireless resource allocation", IEEE Transactions on Wireless Communications **14**, 11, 5927–5940 (2015).

Sengupta, S., S. Basak, P. Saikia, S. Paul, V. Tsalavoutis, F. Atiah, V. Ravi and A. Peters, "A review of deep learning with special emphasis on architectures, applications and recent trends", Knowledge-Based Systems **194**, 105596.1–105596.33 (2020).

Seshadri, A., M. Luk, N. Qu and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses", in "Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles", pp. 335–350 (2007).

Seshadri, V., D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology", in "Proc. IEEE/ACM Int. Symp. on Microarch.", pp. 273–287 (2017).

Shah, P. and R. Patil, "Hardware Assisted Virtualization, 15-612 Operating System Practicum, Carnegie Mellon University", URL `https://www.cs.cmu.edu/ 412/lectures/L04_VTx.pdf`, last accessed Apr. 2, 2020 (2013).

Shah, S. D. A., M. A. Gregory, S. Li and R. Fontes, “SDN enhanced multi-access edge computing (MEC) for E2E mobility and QoS management”, IEEE Access **8**, 77459–77469 (2020).

Shah, S. D. A., M. A. Gregory, S. Li and R. D. R. Fontes, “SDN enhanced multi-access edge computing (MEC) for E2E mobility and QoS management”, IEEE Access **8**, 77459–77469 (2020).

Shantharama, P., A. S. Thyagaturu, N. Karakoc, L. Ferrari, M. Reisslein and A. Scaglione, “LayBack: SDN management of multi-access edge computing (MEC) for network access services and radio resource sharing”, IEEE Access **6**, 57545–57561 (2018a).

Shantharama, P., A. S. Thyagaturu, N. Karakoc, L. Ferrari, M. Reisslein and A. Scaglione, “LayBack: SDN management of multi-access edge computing (MEC) for network access services and radio resource sharing”, IEEE Access **6**, 57545–57561 (2018b).

Shantharama, P., A. S. Thyagaturu and M. Reisslein, “Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies”, IEEE Access **8**, 132021–132085 (2020).

Sharkawi, S. S. and G. Chochia, “Communication protocol optimization for enhanced GPU performance”, IBM Journal of Research and Development **64**, 3/4, 9:1–9:9 (2020).

Sharma, P., L. Chaufournier, P. Shenoy and Y. Tay, “Containers and virtual machines at scale: A comparative study”, in “Proc. Int. Middleware Conf.”, pp. 1–13 (2016).

Sharma, S. K., I. Woungang, A. Anpalagan and S. Chatzinotas, “Toward tactile internet in beyond 5G era: Recent advances, current issues, and future directions”, IEEE Access **8**, 56948–56991 (2020).

Shen, Z., Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. van Renesse and H. Weatherspoon, “Follow the sun through the clouds: Application migration for geographically shifting workloads”, in “Proc. ACM Symp. on Cloud Computing”, pp. 141–154 (2016).

Shen, Z., Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse and H. Weatherspoon, “X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers”, in “Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems”, pp. 121–135 (2019).

Sieber, C., R. Durner, M. Ehm, W. Kellerer and P. Sharma, “Towards optimal adaptation of NFV packet processing to modern CPU memory architectures”, in “Proc. ACM Workshop on Cloud-Assisted Netw.”, pp. 7–12 (2017).

Silberstein, M., “OmniX: an accelerator-centric OS for omni-programmable systems”, in “Proc. ACM Workshop on Hot Topics in Operating Sys.”, pp. 69–75 (2017).

Silva, J. d. C., J. J. P. C. Rodrigues, J. Al-Muhtadi, R. A. L. Rabelo and V. Furtado, "Management platforms and protocols for internet of things: A survey", Sensors **19**, 3, URL `http://www.mdpi.com/1424-8220/19/3/676` (2019).

Singhal, R., "Inside intel next generation nehalem microarchitecture", in "Hot Chips", vol. 20, p. 15 (2008).

Smith, D. R. and W. Whitt, "Resource sharing for efficiency in traffic systems", Bell Labs Techn. J. **60**, 1, 39–55 (1981).

Solomon, R. L. and T. E. Hoglund, "Paravirtualization acceleration through single root i/o virtualization", US Patent 8,332,849 (2012).

Song, H., J. Gong and H. Chen, "Network map reduce", arXiv preprint arXiv:1609.02982 (2016).

Sotomayor, B., K. Keahey and I. Foster, "Overhead matters: A model for virtual resource management", in "Proc. IEEE Virt. Tech. in Distr. Comp.", pp. 1–8 (2006).

Srikant, R., "On the positive recurrence of a markov chain describing file arrivals and departures in a congestion-controlled network", in "IEEE Conf. Comput.", (2004).

Stoess, J., C. Lang and F. Bellosa, "Energy management for hypervisor-based virtual machines.", in "USENIX annual technical conference", pp. 1–14 (2007).

Storck, C. R. and F. Duarte-Figueiredo, "A 5G V2X ecosystem providing internet of vehicles", Sensors **19**, 3, URL `http://www.mdpi.com/1424-8220/19/3/550` (2019).

Strunk, A., "Costs of virtual machine live migration: A survey", in "Proc. IEEE Eighth World Congress on Services", pp. 323–329 (2012).

Stuecheli, J., W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink and B. Allison, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI", IBM Journal of Research and Development **62**, 4/5, 8–1 (2018).

Stunkel, C., R. Graham, G. Shainer, M. Kagan, S. S. Sharkawi, B. Rosenburg and G. Chochia, "The high-speed networks of the Summit and Sierra supercomputers", IBM Journal of Research and Development **64**, 3/4, 3:1–3:10 (2020).

Su, R., D. Zhang, R. Venkatesan, Z. Gong, C. Li, F. Ding, F. Jiang and Z. Zhu, "Resource allocation for network slicing in 5G telecommunication networks: A survey of principles and models", IEEE Network **33**, 6, 172–179 (2019).

Sugerman, J., G. Venkitachalam and B.-H. Lim, "Virtualizing I/O devices on VMware workstation's hosted Virtual Machine Monitor", in "Proc. USENIX Annual Techn. Conf., General Track", pp. 1–14 (2001).

Sundaresan, K., M. Y. Arslan, S. Singh, S. Rangarajan and S. V. Krishnamurthy, "FluidNet: a flexible cloud-based radio access network for small cells", IEEE/ACM Trans. on Netw. **24**, 2, 915–928 (2016).

Szabo, R., M. Kind, F.-J. Westphal, H. Woesner, D. Jocha and A. Csaszar, "Elastic network functions: opportunities and challenges", IEEE Network **29**, 3, 15–21 (2015).

Szefer, J. and R. B. Lee, "Architectural support for hypervisor-secure virtualization", ACM SIGPLAN Notices **47**, 4, 437–450 (2012).

Taherizadeh, S. and M. Grobelnik, "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications", Advances in Engineering Software **140**, 102734.1–102734.11 (2020).

Taleb, T., Y. Hadjadj-Aoul and K. Samdanis, "Efficient solutions for enhancing data traffic management in 3GPP networks", IEEE Systems Journal **9**, 2, 519–528 (2015).

Taleb, T., K. Samdanis, B. Mada, H. Flinck, S. Dutta and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration", IEEE Commun. Surveys & Tutorials **19**, 3, 1657–1681 (2017).

Tam, S. M., H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh *et al.*, "SkyLake-SP: A 14 nm 28-Core Xeon® processor", in "Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)", pp. 34–36 (2018).

Tanaka, H., M. Yoshida, K. Mori and N. Takahashi, "Multi-access edge computing: A survey", Journal of Information Processing **26**, 87–97 (2018).

Tang, J., B. Shim and T. Q. S. Quek, "Service multiplexing and revenue maximization in sliced C-RAN incorporated with URLLC and multicast eMBB", IEEE J. on Sel. Areas in Commun. **37**, 4, 881–895 (2019).

Tang, J., L. Teng, T. Q. S. Quek, T. H. Chang and B. Shim, "Exploring the interactions of communication, computing and caching in cloud RAN under two timescale", in "Proc. IEEE Int. Workshop on Signal Proc. Advances in Wireless Commun. (SPAWC)", pp. 1–6 (2017).

Tang, L. and H. Hu, "Computation offloading and resource allocation for the internet of things in energy-constrained MEC-enabled HetNets", IEEE Access **8**, 47509–47521 (2020).

Tasdemir, E., M. Tömösközi, J. A. Cabrera, F. Gabriel, D. You, F. H. P. Fitzek and M. Reisslein, "SpaRec: Sparse systematic RLNC recoding in multi-hop networks", IEEE Access **9**, 168567–168586 (2021).

Tassiulas, L. and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks", IEEE T. Automat. Contr. **37**, 12, 1936–1948 (1992).

Tassiulas, L. and A. Ephremides, "Dynamic server allocation to parallel queues with randomly varying connectivity", IEEE T. Inform. Theory **39**, 2, 466–478 (1993).

Tayyaba, S. K. and M. A. Shah, "Resource allocation in sdn based 5g cellular networks", Peer-to-Peer Networking and Applications **12**, 2, 514–538 (2019).

Teich, P., "The heart of AMD's EPYC comeback is Infinity Fabric", URL `https://www.nextplatform.com/2017/07/12/heart-amds-epyc-comeback-infinity-fabric/`, last accessed June 2, 2020 (2017).

Telco Systems, A Bath Company, "NFVTime-OS: The NFVi OS that turns any x86 into a carrier class uCPE", URL `https://www.telco.com/product-description/nfvtime-os`, last accessed Apr. 4, 2020 (2020).

Teng, Y. and M. Song, "Cross-layer optimization and protocol analysis for cognitive ad hoc communications", IEEE Access (2017).

Tensilica, "Tensilica Customizable Processor and DSP IP: Application Optimization with the Xtensa Processor Generator", Https://ip.cadence.com/ipportfolio/tensilica-ip, Last accessed June 8, 2020 (2020).

Thalheim, J., P. Bhatotia, P. Fonseca and B. Kasikci, "Cntr: Lightweight {OS} containers", in "2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)", pp. 199–212 (2018).

Thyagaturu, A., Y. Dashti and M. Reisslein, "SDN based smart gateways (Sm-GWs) for multi-operator small cell network management", IEEE Trans. Netw. Serv. Managm. **13**, 4, 740–753 (2016a).

Thyagaturu, A. S., Z. Alharbi and M. Reisslein, "R-FFT: Function split at IFFT/FFT in unified LTE CRAN and cable access network", IEEE Transactions on Broadcasting **64**, 3, 648–665 (2018a).

Thyagaturu, A. S., Z. Alharbi and M. Reisslein, "R-fft: Function split at ifft/fft in unified lte cran and cable access network", IEEE Transactions on Broadcasting **64**, 3, 648–665 (2018b).

Thyagaturu, A. S., A. Mercian, M. P. McGarry, M. Reisslein and W. Kellerer, "Software defined optical networks (sdons): A comprehensive survey", IEEE Communications Surveys & Tutorials **18**, 4, 2738–2786 (2016b).

Tonini, F., B. M. Khorsandi, S. Bjornstad, R. Veisllari and C. Raffaelli, "C-RAN traffic aggregation on latency-controlled ethernet links", Applied Sciences **8**, 11, URL `http://www.mdpi.com/2076-3417/8/11/2279` (2018).

Tork, M., L. Maudlej and M. Silberstein, "Lynx: A SmartNIC-driven accelerator-centric architecture for network servers", in "Proc. ACM Int. Conf. on Arch. Support for Progr. Lang. and Operat. Sys.", pp. 117–131 (2020).

Tran, G. K., R. Santos, H. Ogawa, M. Nakamura, K. Sakaguchi and A. Kassler, "Context-based dynamic meshed backhaul construction for 5G heterogeneous networks", Journal of Sensor and Actuator Networks **7**, 4, URL `http://www.mdpi.com/2224-2708/7/4/43` (2018).

Tsai, W.-C., Y.-C. Lan, Y.-H. Hu and S.-J. Chen, "Networks on chips: structure and design methodologies", Journal of Electrical and Computer Engineering **2012**, 509465, 1–15 (2012).

Tu, C.-C., M. Ferdman, C.-T. Lee and T.-C. Chiueh, "A comprehensive implementation and evaluation of direct interrupt delivery", ACM SIGPLAN Notices **50**, 7, 1–15 (2015).

Tzanakaki, A., M. Anastasopoulos, I. Berberana, D. Syrivelis, P. Flegkas, T. Korakis, D. C. Mur, I. Demirkol, J. Gutierrez, E. Grass, Q. Wei, E. Pateromichelakis, N. Vucic, A. Fehske, M. Grieger, M. Eiselt, J. Bartelt, G. Fettweis, G. Lyberopoulos, E. Theodoropoulou and D. Simeonidou, "Wireless-optical network convergence: Enabling the 5G architecture to support operational and end-user services", IEEE Communications Magazine **55**, 10, 184–192 (2017).

Van Doorn, L., "Hardware virtualization trends", in "Proc. ACM Int. Conf. on Virtual Execution Environm.", pp. 45–45 (2006).

Vaughan-Nichols, S. J., "New approach to virtualization is a lightweight", Computer **39**, 11, 12–14 (2006).

Velte, T. and A. Velte, *Cisco A Beginner's Guide* (McGraw-Hill Education Group, New York, 2013).

Venkat, A., H. Basavaraj and D. M. Tullsen, "Composite-ISA cores: Enabling multi-ISA heterogeneity using a single ISA", in "Proc. IEEE Int. Symp. on High Perf. Computer Arch. (HPCA)", pp. 42–55 (2019).

Venkat, A. and D. M. Tullsen, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor", in "Proc. ACM/IEEE Int. Symp. on Comp. Arch.", pp. 121–132 (2014).

Vikranth, B., R. Wankar and C. R. Rao, "Topology aware task stealing for on-chip numa multi-core processors", Procedia Computer Science **18**, 379–388 (2013).

Vipin, K. and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: a survey of architectures, methods, and applications", ACM Computing Surveys (CSUR) **51**, 4, 72.1–72.39 (2018).

Wang, C., F. R. Yu, C. Liang, Q. Chen and L. Tang, "Joint computation offloading and interference management in wireless cellular networks with mobile edge computing", IEEE Trans. on Vehicular Techn. **66**, 8, 7432 – 7445 (2017a).

Wang, D., L. Zhang, Y. Qi and A. Quddus, "Localized mobility management for SDN-integrated LTE backhaul networks", in "Proc. of IEEE VTC", pp. 1–6 (2015a).

Wang, H., Z. Peng and Y. Pei, "Offloading schemes in mobile edge computing with an assisted mechanism", IEEE Access **8**, 50721–50732 (2020).

Wang, J., D. Bonneau, M. Villa, J. W. Silverstone, R. Santagati, S. Miki, T. Yamashita, M. Fujiwara, M. Sasaki, H. Terai *et al.*, "Chip-to-chip quantum photonic interconnect by path-polarization interconversion", Optica **3**, 4, 407–413 (2016a).

Wang, K., S. Qi, Z. Chen, Y. Yang and H. Gu, "SMONoC: Optical network-on-chip using a statistical multiplexing strategy", Optical Switching and Networking **34**, 1–9 (2019a).

Wang, M., N. Karakoc, L. Ferrari, P. Shantharama, A. S. Thyagaturu, M. Reisslein and A. Scaglione, "A multi-layer multi-timescale network utility maximization framework for the SDN-based LayBack architecture enabling wireless backhaul resource sharing", Electronics **8**, 9, 937.1–937.28 (2019b).

Wang, N., E. Hossain and V. Bhargava, "Backhauling 5G small cells: A radio resource management perspective", IEEE Wireless Communications **22**, 5, 41–49 (2015b).

Wang, P.-H., C.-H. Li and C.-L. Yang, "Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture", in "Proc. ACM Ann. Design Autom. Conf.", pp. 1–6 (2016b).

Wang, R., Y. Wang, J.-S. Tsai, A. Herdrich, T.-Y. Tai, N. McDonnell, S. Van Doren, D. Sonnier, D. Bernstein, H. Wilkinson *et al.*, "Technologies for a distributed hardware queue manager", US Patent App. 15/087,154 (2017b).

Wang, S., J. Xu, N. Zhang and Y. Liu, "A survey on service migration in mobile edge computing", IEEE Access **6**, 23511–23528 (2018).

Wang, S., X. Zhang, Y. Zhang, L. Wang, J. Yang and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications", IEEE Access **5**, 6757–6779 (2017c).

Wang, X., X. Chen, T. Chen, L. Huang and G. B. Giannakis, "Two-scale stochastic control for integrated multipoint communication systems with renewables", IEEE Trans. on Smart Grid **9**, 3, 1822–1834 (2018a).

Wang, Y., "Numa-aware design and mapping for pipeline network functions", in "Int. Conf. on Systems and Informatics (ICSAI)", pp. 1049–1054 (IEEE, 2017).

Wang, Z. and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity", in "Proc. IEEE Symp. on Security and Privacy", pp. 380–395 (2010).

Wang, Z., L. Zhang, M. Wang, Z. Wang, D. Zhu, Y. Zhang and W. Zhao, "High-density NAND-like spin transfer torque memory with spin orbit torque erase operation", IEEE Electron Device Letters **39**, 3, 343–346 (2018b).

Weiland, M., A. Jackson, N. Johnson and M. Parsons, "Exploiting the performance benefits of storage class memory for HPC and HPDA workflows", Supercomputing Frontiers and Innovations **5**, 1, 79–94 (2018).

Woesner, H., P. Greto and T. Jungel, "Hardware acceleration of virtualized network functions: Offloading to SmartNICs and ASIC", in "Proc. IEEE Int. Scientific and Techn. Conf. Modern Computer Netw. Techn. (MoNeTeC)", pp. 1–6 (2018).

Wood, T., K. Ramakrishnan, J. Hwang, G. Liu and W. Zhang, "Toward a software-based network: Integrating software defined networking and network function virtualization", IEEE Network **29**, 3, 36–41 (2015).

Wunderlich, S., F. H. Fitzek and M. Reisslein, "Progressive multicore RLNC decoding with online DAG scheduling", IEEE Access **7**, 161184–161200 (2019).

Wunderlich, S., F. Gabriel, S. Pandi, F. H. Fitzek and M. Reisslein, "Caterpillar RLNC (CRLNC): A practical finite sliding window RLNC approach", IEEE Access **5**, 20183–20197 (2017).

Xia, W., T. Q. S. Quek, J. Zhang, S. Jin and H. Zhu, "Programmable hierarchical C-RAN: From task scheduling to resource allocation", IEEE Trans. on Wireless Commun. **18**, 3, 2003–2016 (2019).

Xiang, Z., F. Gabriel, E. Urbano, G. T. Nguyen, M. Reisslein and F. H. Fitzek, "Reducing latency in virtual machines: Enabling tactile internet for human-machine co-working", IEEE J. Sel. Areas in Commun. **37**, 5, 1098–1116 (2019).

Xiang, Z., M. Höweler, D. You, M. Reisslein and F. H. Fitzek, "X-MAN: A non-intrusive power manager for energy-adaptive cloud-native network functions", IEEE Transactions on Network and Service Management, in print (2022).

Xiao, J., P. Andelfinger, D. Eckhoff, W. Cai and A. Knoll, "A survey on agent-based simulation using hardware accelerators", ACM Computing Surveys (CSUR) **51**, 6, 131.1–131.35 (2019).

Xie, Z., X. Song and S. Xu, "Peer-to-peer enhanced task scheduling for D2D enabled MEC network", IEEE Access pp. 1–1 (2020).

Xu, C., S. Chen, J. Su, S.-M. Yiu and L. C. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms", IEEE Commun. Surv. & Tut. **18**, 4, 2991–3029 (2016).

Xu, Q., H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh and V. Balakrishnan, "Performance analysis of NVMe SSDs and their implication on real world databases", in "Proc. ACM Int. Systems and Storage Conf.", pp. 6.1–6.11 (2015).

Xu, X., A. Watts and M. Reed, "Does access to Internet promote innovation? A look at the US broadband industry", Growth and Change **50**, 4, 1423–1440 (2019).

Yahya, M. R., N. Wu, Z. A. Ali and Y. Khizar, "Optical versus electrical: Performance evaluation of Network On-Chip topologies for UWASN manycore processors", Wireless Personal Communications, in print pp. 1–29 (2020).

Yamato, Y., "Openstack hypervisor, container and baremetal servers performance comparison", IEICE Communications Express **4**, 7, 228–232 (2015).

Yan, J., L. Tang, J. Li, X. Yang, W. Quan, H. Chen and Z. Sun, "UniSec: a unified security framework with SmartNIC acceleration in public cloud", in "Proc. ACM Turing Celebration Conf.-China", pp. 1–6 (2019).

Yang, J., J. Luo, F. Lin and J. Wang, "Content-sensing based resource allocation for delay-sensitive VR video uploading in 5G H-CRAN", Sensors **19**, 3, URL `http://www.mdpi.com/1424-8220/19/3/697` (2019a).

Yang, M., Y. Li, D. Jin, L. Zeng, X. Wu and A. V. Vasilakos, "Software-defined and virtualized future mobile and wireless networks: A survey", Mobile Networks and Applications **20**, 1, 4–18 (2015).

Yang, S., "Extending KVM with new intel® virtualization technology", URL `https://www.linux-kvm.org/images/c/c7/KvmForum2008%24kdf2008_11.pdf`, kVM Forum (2008).

Yang, S., W.-H. Yeung, T.-I. Chao, K.-H. Lee and C.-I. Ho, "Hardware acceleration for batched sparse codes", US Patent 10,237,782 (2019b).

Yang, W., "Conceptual verification of integrated heterogeneous network based on 5G millimeter wave use in gymnasium", Symmetry **11**, 3 (2019).

Yang, W. and C. Fung, "A survey on security in network functions virtualization", in "Proc. IEEE NetSoft Conf. and Workshops (NetSoft)", pp. 15–19 (2016).

Yao, R. and Y. Ye, "Towards a high-performance and low-loss Clos-Benes based optical Network-on-Chip architecture", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, in print (2020).

Yao, Y., L. Huang, A. B. Sharma, L. Golubchik and M. J. Neely, "Power cost reduction in distributed data centers: A two-time-scale approach for delay tolerant workloads", IEEE Trans. on Parallel and Distributed Systems **25**, 1, 200–211 (2014).

Yasukata, K., M. Honda, D. Santry and L. Eggert, "StackMap: Low-latency networking with the OS stack and dedicated NICs", in "Proc. USENIX Ann. Techn. Conf.", pp. 43–56 (2016).

Yazdanshenas, S. and V. Betz, "Interconnect solutions for virtualized field-programmable gate arrays", IEEE Access **6**, 10497–10507 (2018).

Ye, Y., W. Zhang and W. Liu, "Thermal-aware design and simulation approach for optical NoCs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, in print pp. 1–1 (2020).

Yeoh, C. Y., M. H. Mokhtar, A. A. A. Rahman and A. K. Samingan, "Performance study of LTE experimental testbed using OpenAirInterface", in "Proc. IEEE Int. Conf. Adv. Comm. Tech.", pp. 617–622 (2016).

Yi, S., C. Li and Q. Li, “A survey of fog computing: Concepts, applications and issues”, in “Proc. ACM Workshop on Mobile Big Data”, pp. 37–42 (2015).

Yi, X., J. Duan and C. Wu, “GPUNFV: A GPU-accelerated NFV system”, in “Proc. ACM Asia-Pacific Workshop on Networking”, pp. 85–91 (2017).

Yokoyama, D., B. Schulze, F. Borges and G. Mc Evoy, “The survey on ARM processors for HPC”, The Journal of Supercomputing **75**, 7003–7036 (2019).

Yousaf, F. Z., V. Sciancalepore, M. Liebsch and X. Costa-Perez, “MANOaaS: A multi-tenant NFV MANO for 5G network slices”, IEEE Communications Magazine **57**, 5, 103–109 (2019).

Youseff, L., R. Wolski, B. Gorda and C. Krintz, “Paravirtualization for hpc systems”, in “International Symposium on Parallel and Distributed Processing and Applications”, pp. 474–486 (Springer, 2006).

Yu, B., Y. Liu, Y. Ye, X. Liu and Q. J. Gu, “Low-loss and broadband G-band dielectric interconnect for chip-to-chip communication”, IEEE Microwave and Wireless Components Letters **26**, 7, 478–480 (2016).

Yu, L., T. Jiang, Y. Cao and Q. Qi, “Joint workload and battery scheduling with heterogeneous service delay guaranteesfor data center energy cost minimization”, IEEE Trans. on Parallel and Distributed Systems **26**, 7, 1937–1947 (2015).

Zhang, F., J. Chen, H. Chen and B. Zang, “CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization”, in “Proc. ACM Symp. on Operating Sys. Principles”, pp. 203–216 (2011).

Zhang, H., “The end of the x86 dominance in databases?”, in “Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)”, p. 1 (2019).

Zhang, J., G. Park, D. Donofrio, J. Shalf and M. Jung, “DRAM-Less: Hardware acceleration of data processing with new memory”, in “Proc. IEEE Int. Symp. on High Perf. Computer Arch. (HPCA)”, pp. 287–302 (2020).

Zhang, L., Y. Zhang, A. Tsuchiya, M. Hashimoto, E. S. Kuh and C.-K. Cheng, “High performance on-chip differential signaling using passive compensation for global communication”, in “Proc. IEEE Asia and South Pacific Design Autom. Conf.”, pp. 385–390 (2009).

Zhang, Q., L. Liu, C. Pu, Q. Dou, L. Wu and W. Zhou, “A comparative study of containers and virtual machines in big data environment”, in “Proc. IEEE Int. Conf. on Cloud Comp. (CLOUD)”, pp. 178–185 (2018).

Zhang, S., Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen and Y. Chen, “Cambricon-X: An accelerator for sparse neural networks”, in “Proc. IEEE/ACM Int. Symp. on Microarch.”, pp. 1–12 (2016).

Zhang, S., C. Kai and L. Song, "SDN based uniform network architecture for future wireless networks", in "Proc. Int. Conf. on Computing, Commun. and Netw. Techn. (ICCCNT)", pp. 1–5 (2014).

Zhang, T., "Design of NFV platforms: A survey", arXiv preprint arXiv:2002.11059 (2020).

Zhang, T., L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone and J. Roberts, "Comparing the performance of state-of-the-art software switches for NFV", in "Proc. ACM Int. Conf. on Emerging Netw. Exp. and Techn.", pp. 68–81 (2019a).

Zhang, W., L. Li, N. Zhang, T. Han and S. Wang, "Air-ground integrated mobile edge networks: A survey", IEEE Access **8**, 125998–126018 (2020).

Zhang, Y., A. Rucker, M. Vilim, R. Prabhakar, W. Hwang and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures", in "Proc. Int. Symp. on Computer Arch.", pp. 615–628 (2019b).

Zhang, Y., X. Xiao, K. Zhang, S. Li, A. Samanta, Y. Zhang, K. Shang, R. Proietti, K. Okamoto and S. B. Yoo, "Foundry-enabled scalable all-to-all optical interconnects using silicon nitride arrayed waveguide router interposers and silicon photonic transceivers", IEEE Journal of Selected Topics in Quantum Electronics **25**, 5, 1–9 (2019c).

Zhao, N., X. Liu, F. R. Yu, M. Li and V. C. Leung, "Communications, caching, and computing oriented small cell networks with interference alignment", IEEE Commun. Mag. **54**, 9, 29–35 (2016).

Zhezlov, K. A., F. M. Putrya and A. A. Belyaev, "Analysis of performance bottlenecks in SoC interconnect subsystems", in "Proc. IEEE Conf. of Russian Young Researchers in Electrical and Electronic Eng.", pp. 1911–1914 (2020).

Zhong, Y., Z. Zhou, D. Li, M. Guo, Q. Liu, Y. Liu and L. Guo, "SAED: A self-adaptive encryption and decryption architecture", in "Proc. IEEE Int. Conf. on Parallel Distr. Proc. with Appl.", pp. 388–397 (2019).

Zhou, Y., F. He, N. Hou and Y. Qiu, "Parallel ant colony optimization on multi-core SIMD CPUs", Future Generation Computer Systems **79**, 473–487 (2018).

Zhu, Q., B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi and F. Franchetti, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing", in "Proc. IEEE Int. 3D Systems Integration Conf.", pp. 1–7 (2013).

Zilberman, N., P. M. Watts, C. Rotsos and A. W. Moore, "Reconfigurable network systems and software-defined networking", Proc. IEEE **103**, 7, 1102–1124 (2015).