IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

*Digital Object Identifier 10.1109/ACCESS.2017.DOI*

# Hardware-Accelerated Platforms and Infrastructures for Network Functions: A Survey of Enabling Technologies and Research Studies

**PRATEEK SHANTHARAMA[1], AKHILESH S. THYAGATURU[2], and MARTIN REISSLEIN[1]**
*Fellow, IEEE*
[1]School of Electrical, Computer, and Energy Engineering, Arizona State University (ASU), Tempe, AZ 85287, USA (e-mail: {pshanth1, reisslein}@asu.edu)
[2]Programmable Solutions Group (PSG), Intel Corporation, Chandler, AZ 85226, USA (e-mail: akhilesh.s.thyagaturu@intel.com)

Corresponding author: Martin Reisslein (e-mail: reisslein@asu.edu).

This work was supported by the National Science Foundation through Grant 1716121.

**ABSTRACT** 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. This article comprehensively surveys hardware-accelerated platforms and infrastructures for executing softwarized NFs. This survey covers both commercial products, which we consider to be enabling technologies, as well as relevant research studies. We have organized the survey into the main categories of enabling technologies and research studies on hardware accelerations for the CPU, the memory, and the interconnects (e.g., between CPU and memory), as well as custom and dedicated hardware accelerators (that are embedded on the platforms); furthermore, we survey hardware-accelerated infrastructures that connect GPC platforms to networks (e.g., smart network interface cards). We find that the CPU hardware accelerations have mainly focused on extended instruction sets and CPU clock adjustments, as well as cache coherency. Hardware accelerated interconnects have been developed for on-chip and chip-to-chip connections. Our comprehensive up-to-date survey identifies the main trade-offs and limitations of the existing hardware-accelerated platforms and infrastructures for NFs and outlines directions for future research.

**INDEX TERMS** Central Processing Unit (CPU), Hardware Accelerator, Interconnect, Memory, Software Defined Networking (SDN), Virtualized Network Function (VNF).

## I. INTRODUCTION

### A. 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 [1]–[3]. 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 I-B, followed by an overview of the contributions of this survey in Section I-C.

#### 1) 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) [4] or containers [5], 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. 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.

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

Software Defined Networking (SDN) [6]–[9] 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 [10], [11]. 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 ongoing operations based on run-time requirements, such as traffic loads, failures and their restoration, as well as resource utilization.

#### 3) 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 [12] 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 [13].

The wide-spread adoption of Multi-Access Edge Computing (MEC) [14] 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 [15]–[19]. 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 [20]. In a typical MEC node deployment, the GPC platform is virtualized by a hypervisor [21], 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 [22].

#### 4) 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 [11], [23]–[28]. OpenStack [29] and

Kubernetes [30], [31] 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.

## B. 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 [11]. 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 [32] and AMD® [33] CPU platforms, are not natively optimized to run NFs that include routine packet processing procedures across the I/O path [34]–[37]. 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) [38], Field Programmable Gate Array (FPGA), Graphics Processing Unit (GPU), and Application Specific Integrated Circuit (ASIC) [39], 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 [40].

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 [41]. Moreover, the CPU power and thermal characteristics can vary the base operating frequency, introducing variable processing time behaviors [42]–[44]. 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.

## C. 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 I-D and provide background on the processing of NFs in Section II. Section III comprehensively surveys the relevant enabling technologies for hardware-accelerated platforms and infrastructures for processing NFs, while Section IV 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 III 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 III surveys the relevant infrastructure technologies (SmartNICs and Non-Transparent Bridging). Section IV categorizes the research studies into studies addressing the computer architecture, interconnects, memory, and accelerators on platforms; moreover, Section IV surveys infrastructure research on SmartNICs. Section V summarizes the main open challenges for hardware-accelerated platforms and infrastructures for processing softwarized NFs and Section VI concludes this survey article.

## D. 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 I-D1 through I-D5 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 I-D6 focuses on prior survey articles that cover aspects of our topic area. Section I-D6 highlights our original survey coverage of hardware-accelerated platforms and infrastructures for NFs with respect to prior related survey articles

### 1) 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 [45]–[47]. The traditional challenges of NFV deployment are associated with the virtualization process of NFs, such as overhead, isolation, resource allocation, and function management [48]. Herrera et al. [49] have discussed the resource allocation and placement of applications, VMs, and containers on GPC platforms. More specifically, Herrera et al. [49] 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. [50] and Farris et al. [51] 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. [52]. Furthermore, Lal et al. [53] have presented best practices for NFV designs against security threats.

### 2) 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) [54]–[56]. 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. [11]. 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 [57]. In contrast to surveys of independent designs of SDN and NFV, Bonfim et al. [58] have presented an overview of integrated NFV/SDN architectures, focusing on SDN interfaces and Application Programming Interfaces (APIs) specific to NFV management.

### 3) Network Function Virtualization (NFV) and Network Slicing

5th Generation (5G) [59]–[61] 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 [62]–[64]. 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. [65] have presented a survey of SDN management of VNFs for 5G networks, while Nguyen et al. [66] have discussed the relative benefits of different SDN/NFV-based mobile packet core network architectures. Bouras et al. [67] have discussed the challenges that are associated with SDN and NFV based 5G networks, such as scalability and reliability. Costa et al. [68] 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. [69] have surveyed the softwarization principles and enabling technologies for network slicing. As discussed in the survey by Foukas et al. [70] 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. [71]. The surveys [72]–[75] have discussed network slicing and the management of resources in the context of 5G based on both SDN and NFV.

### 4) 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. [76] have presented a survey of NFV techniques as applied to edge networks. Some of the NFV aspects that are highlighted by Yi et al. [76] 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. [77], while the role of NFV in MEC has been surveyed by Taleb et al. [78]. The use of both SDN and NFV provides strategies for effective management of MEC resources in edge networks as described by Baktir et al. [79] and Blanco et al. [80].

### 5) 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. [81] 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. [82]. Duan et al. [83] have presented a survey on SDN-based orchestration studies for NFV management.

### 6) 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 [84]. Since GPC platforms are not fundamentally designed for packet processing, GPC platforms require additional acceleration techniques for effective high-speed packet processing [85]. Linguaglossa et al. [86] 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 [86] 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 [87]. However, a comprehensive detailed survey of specific hardware offloading technologies and research studies is not provided in [86]. Zhang [88] 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 [88] mainly covers the VNF software and management aspects, i.e., Management and Orchestration (MANO) components [89], that are involved in NFV deployments. Zhang [88] covers hardware acceleration only very briefly, with only about ten references in one paragraph. In contrast to [86] and [88], 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 [90]. 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. [91] 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 [92]. The Niemiec et al. survey [91] 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).

## II. 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 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. II-A). Recently, with the softwarization of NFs, the trend is towards implementing NFs as software

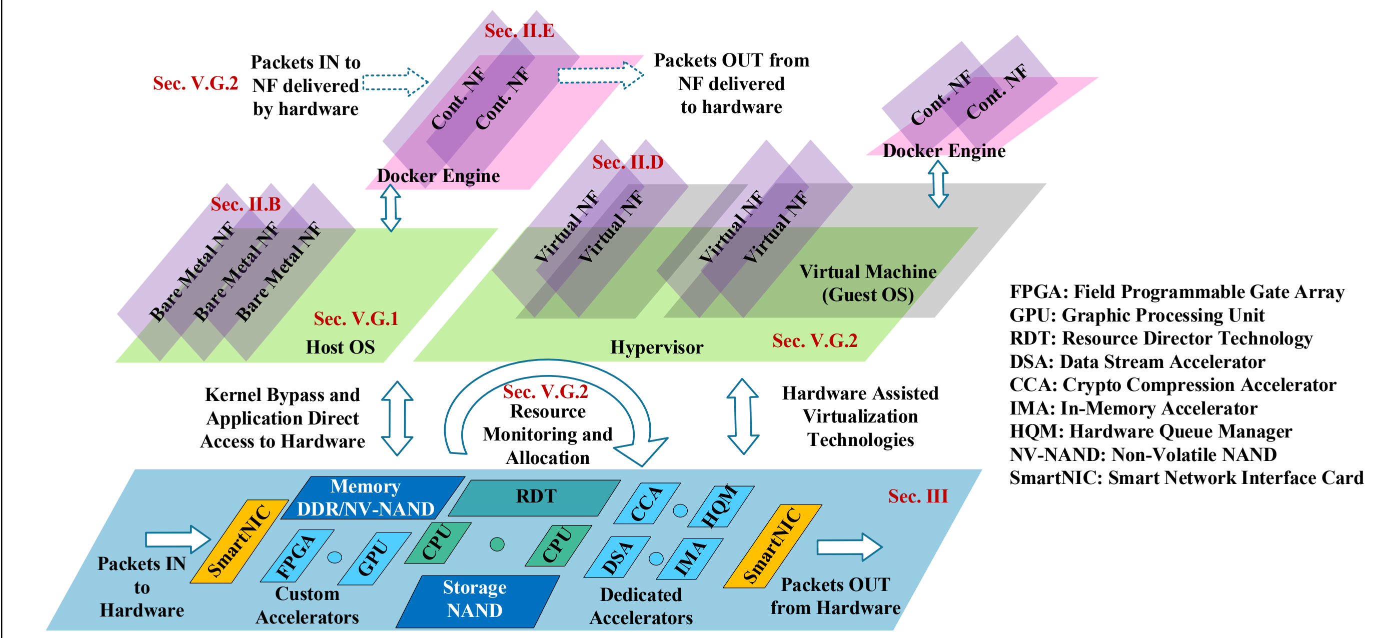
Sec. II.E
Sec. V.G.2
Packets IN to NF delivered by hardware
Cont. NF
Cont. NF
Packets OUT from NF delivered to hardware
Cont. NF
Cont. NF
Docker Engine
Docker Engine
Sec. II.D
Sec. II.B
Bare Metal NF
Bare Metal NF
Bare Metal NF
Virtual NF
Virtual NF
Virtual NF
Virtual NF
Virtual Machine (Guest OS)
Sec. V.G.1
Host OS
Hypervisor
Sec. V.G.2
Kernel Bypass and Application Direct Access to Hardware
Sec. V.G.2
Resource Monitoring and Allocation
Hardware Assisted Virtualization Technologies
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
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

FIGURE1: 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).

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. II-B), as application running on an OS, i.e., as user application, or as kernel module as part of the OS (see Sec. II-C), as Virtual Machine (VM) on a hypervisor (see Sec. II-D), or as container running on a container engine (see Sec. II-E). Brief background on general acceleration strategies for NFs running on GPC platforms is given in Sec. II-F.

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.

## A. DEDICATED HARDWARE BASED NF IMPLEMENTATION

### 1) 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 [93]–[95]. 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.

### 2) 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.

#### 3) 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.

### B. BARE-METAL (BM) NF IMPLEMENTATION

#### 1) 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 [96]. 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 [97], [98]. 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. [99].

#### 2) Benefits

The BM implementation of NFs can achieve relatively higher performance as compared to NFs running on virtualized and abstracted environments [99]. 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.

#### 3) 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.

### C. APPLICATION AND KERNEL BASED NF IMPLEMENTATION

#### 1) 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.

#### 2) 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.

#### 3) 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.

### D. VIRTUAL MACHINE (VM) BASED NF IMPLEMENTATION

#### 1) 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.

#### 2) 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.

#### 3) 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 [99]. 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 [100].

### E. CONTAINER BASED NF IMPLEMENTATION

#### 1) 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 [101]. 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 [102]. 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.

#### 2) 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) [103].

#### 3) 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.

### F. 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 II-B– II-E), 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.

### 1) Software Acceleration Methods

#### a: 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 [104].

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.

#### b: 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.

#### c: 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 [53]. Also, additional layers of software abstractions for acceleration add more latency for the overall task processing as compared to hardware acceleration.

### 2) Hardware Acceleration Methods

#### a: 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.

#### b: 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 [105].

#### c: 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.

## III. 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 structured according to the classification structure of the enabling technologies in Fig. 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.

### A. 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.

#### 1) 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) [106]. 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 [107] is an example of low-power DSP processor based on the RISC architecture which is optimized for floating point operations [108]. Tensilica processors are typically used in the design of I/O devices (e.g., NIC) 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 [109], [110] and dynamic task scheduling [111]. Similar IS extensions have accelerated the complex network coding function [112], [113], [193] in a hardware design [114].

ISAcc [115], [116] 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:

**TABLE1:** 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 |

#### a: Advanced Encryption Standard-New Instructions (AES-NI)

Advanced Encryption Standard-New Instructions (AES-NI) [117], [118] 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

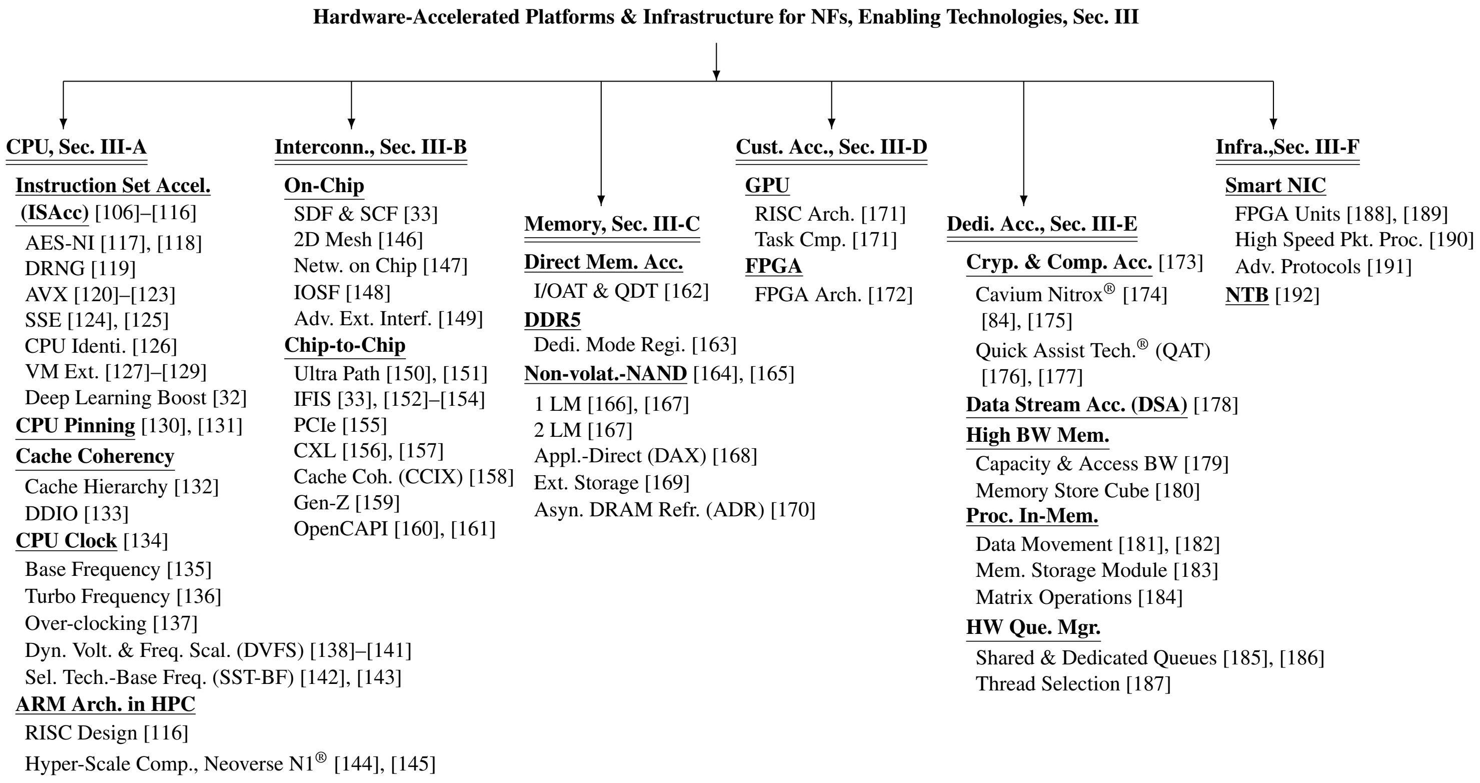

FIGURE2: Classification taxonomy of enabling technologies for hardware-accelerated platforms and infrastructure for processing softwarized NFs: The main platform related categories are hardware accelerations for the CPU, interconnects, and memory, as well as custom and dedicated hardware accelerators that are embedded on the platform; the infrastructure hardware accelerations focus on network interface cards and bridging.

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.

##### b: Digital Random Number Generator (DRNG)

The Digital Random Number Generator (DRNG) [119] 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.

##### c: The Advanced Vector Extensions (AVX)

The Advanced Vector Extensions (AVX) [120], [121] implement an advanced data processing IS for machine learning, encryption, and signal processing [122]. The vectorization of the CPU processing significantly improves the data computations for large vector data sets [123].

##### d: Streaming SIMD Extensions (SSE)

The Streaming SIMD Extensions (SSE) [124], [125] implement accelerations aimed at string and text character processing, which is essential for searches and comparisons. NFs rely on JavaScript Object Notation (JSON), extensible markup language (XML), and text parsing protocols to perform management functions. SSE instructions play an important role in achieving near-real-time decisions based on text look-up and comparisons. SSE instructions also implement compute functions for 32 bit Cyclic Redundancy Checks (CRC32) which are commonly used in data transfer and external storage NFs.

##### e: CPU IDentification (`CPUID`)

The CPU IDentification (`CPUID`) [126] 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.

#### f: Virtual Machine Extensions (VMX)

The Virtual Machine Extensions (VMX) [127]–[129] 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) [194], [195]. 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.

#### g: Deep Learning (DL) Boost

The Deep Learning (DL) Boost IS acceleration on Intel® CPUs [32] 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.

### 2) CPU Pinning

CPU pinning is a resource allocation strategy that allocates and pins a specific workload to a specific physical CPU core. Traditionally, in the OS, application threads and processes are

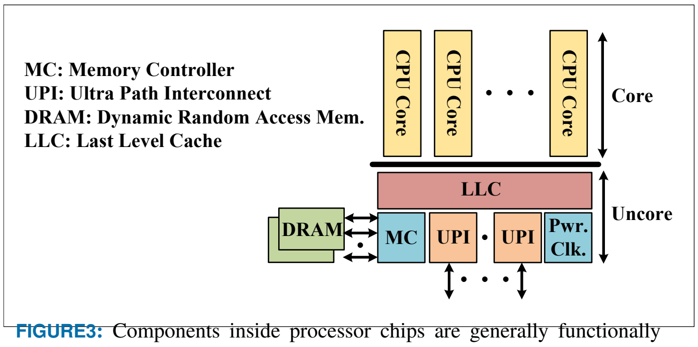

FIGURE3: Components inside processor chips are generally functionally separated into core (i.e., CPUs) and uncore elements. Uncore elements are non-core components, such as clock, memory controllers, integrated accelerators, interrupt controllers, and interconnects.

dynamically scheduled on the CPU cores based on their relative priorities and processing states, such as wait and ready-to-run. As opposed to the OS management of CPU resources, the dedicated and static allocation of CPU core resources for the execution of application threads and processes improves the performance of the pinned application [130], [131]. In addition to no-contention of resources, the performance benefits of CPU pinning are attributed to the data cache coherency, especially at the L1 and L2 cache levels (which reside within the CPU core), when only one application accesses a memory location from a given CPU core.

In virtualization, the VMM scheduler allocates the CPU resources to VMs, i.e., the conversion of instructions to a virtual CPU (vCPU) to an actual physical CPU (pCPU) is achieved dynamically at run time. However, the VMM scheduler may impact the overall performance when there is resource contention by other VMs running on a VMM; in addition, VM based cache coherency issues may arise. Therefore, CPU pinning is an important aspect to consider for the CPU resource allocation (vCPU or pCPU) to a virtualized NF (VNF) via CPU pinning.

### 3) Cache Coherency

Caches play an important role in the overall software execution performance by directly impacting the latency of memory accesses by the CPU. The memory access flow from a CPU first performs an address translation from a virtual address to a physical address. If the address translation fails, then a page fault is registered and a page walk process is invoked. If the address translation succeeds, then cache levels are checked. If there is a cache hit, then the data is read from or written to the cache; whereas, if there is a cache miss, then an external memory read or write is performed. A cache miss or an address translation failure page walk significantly increase the latency and severely impede the NF performance. Therefore, NF designs have to carefully consider the cache coherency of data accesses.

#### a: 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 [132] (see Fig. 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.

#### b: Data-Direct IO (DDIO)

The Data-Direct IO (DDIO) [133] 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.

### 4) 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 [134]. The CPU clock frequency directly impacts the total power dissipated as heat on the CPU die.

#### a: Base Frequency

The base frequency [135] 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.

#### b: Turbo Frequency

The turbo frequency technique [136] *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.

#### c: Over-clocking

Over-clocking [137] 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.

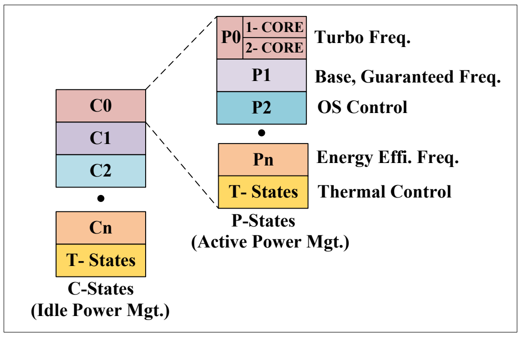

**FIGURE4:** Processor states are broadly classified as CPU states ($C$-States) which indicate the overall CPU state; additionally, when the CPU is active (i.e., in $C0$), then core-specific Power states ($P$-States) indicate the operational frequencies of the cores that are actively executing instructions.

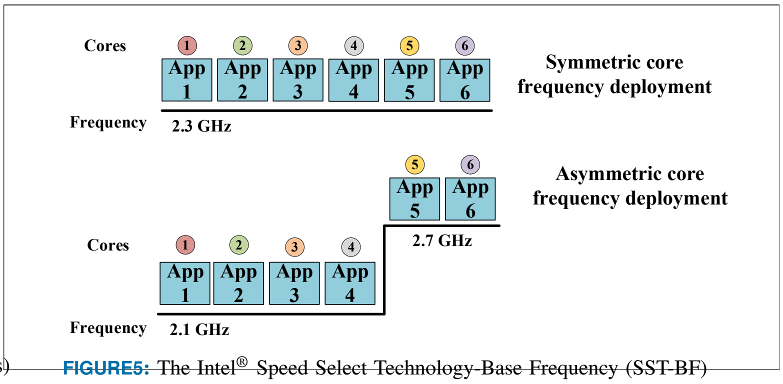

**FIGURE5:** The Intel® Speed Select Technology-Base Frequency (SST-BF) technology [142] deterministically modifies the operating base frequency on specific cores to increase or decrease the base frequencies of the cores at run-time based on the need to provide adequate performance to NF applications running on the cores. In the depicted example scenario, the operating base frequency of two cores is increased to 2.7 GHz while four cores slowed down to 2.1 GHz such that the average operating base frequency across all six cores remains around 2.3 GHz.

#### d: Dynamic Voltage and Frequency Scaling (DVFS)

Dynamic Voltage and Frequency Scaling (DVFS) [138]–[140] defines a system and a set of procedures that control the operational frequency and voltage in a CPU subsystem. Typically, CPU manufactures provide several operational states (see Fig. 4) including: $C0$: CPU is actively executing instructions; $C1$: CPU in halt state with no instruction execution, capable of instantaneously transitioning into $C0$ state; $C2$: Stop-Clock, transition to $C1$ and $C0$ takes longer; $C3$: $C3$ and higher CPU states can be defined as sleep states without power.

In addition to the $C$ states, which define the CPU power characteristics, $P$ states define the performance characteristics of the individual CPU cores, typically when the CPU is in $C0$. The $P$ states include: $P0$: CPU core is operating in turbo frequency mode, the highest performance can be achieved by a specific core; $P1$: CPU core is operating at a guaranteed (base) frequency, a sustained performance can be achieved by all cores; $P2$: CPU core is operating in OS managed lower frequency and voltage, i.e., in low performance modes for $P2$ and subsequent $P$ states; $T$: Thermal control is applied to the CPU cores, as the CPU die has reached the safe operating temperature.

The transitions between different $C$ states and $P$ states are managed by the DVFS subsystem. The DVFS, in conjunction with the OS and BIOS through the Advanced Configuration and Power Interface (ACPI) [141], tries to attain the best performance for the lowest possible power consumption.

#### e: Speed Select Technology-Base Frequency (SST-BF)

The Intel® Speed Select Technology-Base Frequency (SST-BF) [142] enhances the application performance by increasing the base frequency of the CPU. SST-BF increases the base frequency on demand so as to adaptively boost the application performance. SST-BF is thus particularly well suited for NF acceleration, e.g., for quickly handling bursty network traffic through increasing the base frequency when a traffic burst occurs. In contrast to the turbo frequency technique (see Section III-A4b), which increases the CPU frequency opportunistically, SST-BF increases the CPU frequency deterministically when there is a need. An NF running on a GPC platform is susceptible to variations of the CPU clock frequency; thus, running an NF application with the opportunistic turbo frequency technique cannot guarantee the Quality-of-Service (QoS) for the NF. Most NFV deployments require prescribed worst case performance guarantees in order to deliver the services to the users [143]. A high deterministic CPU clock frequency as achieved by SST-BF is an important factor to guarantee the QoS performance.

SST-BF segregates the CPU cores into SST-BF supported and non-supported cores based on their relative distance in terms of their thermal characteristics. A system configuration during the start-up enables SST-BF on the supported cores. When the application requests an increased base frequency, the OS sends a configuration command to the supported cores to increase their base frequency (could be maximum supported value as suggested by the manufacturer). At the same time, the operating frequencies of the SST-BF non-supported CPU cores are reduced so as to maintain the overall average frequency of the cores and to keep the thermal budget of the CPU die within the safe operational range. For instance, if there are 6 cores in a CPU die operating with a normal base frequency of 2.3 GHz (see Fig. 5), and 2 of the SST-BF supported cores request an increased base frequency, the operational frequencies for these two cores would be changed to the maximum base frequency, e.g., 2.7 GHz, while reducing the operational frequencies of the other 4 cores to 2.1 GHz. The OS and the orchestrator can decide which applications to run on the SST-BF supported cores and when to switch the operational frequencies to the supported maximum base frequency on the supported cores.

### 5) 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

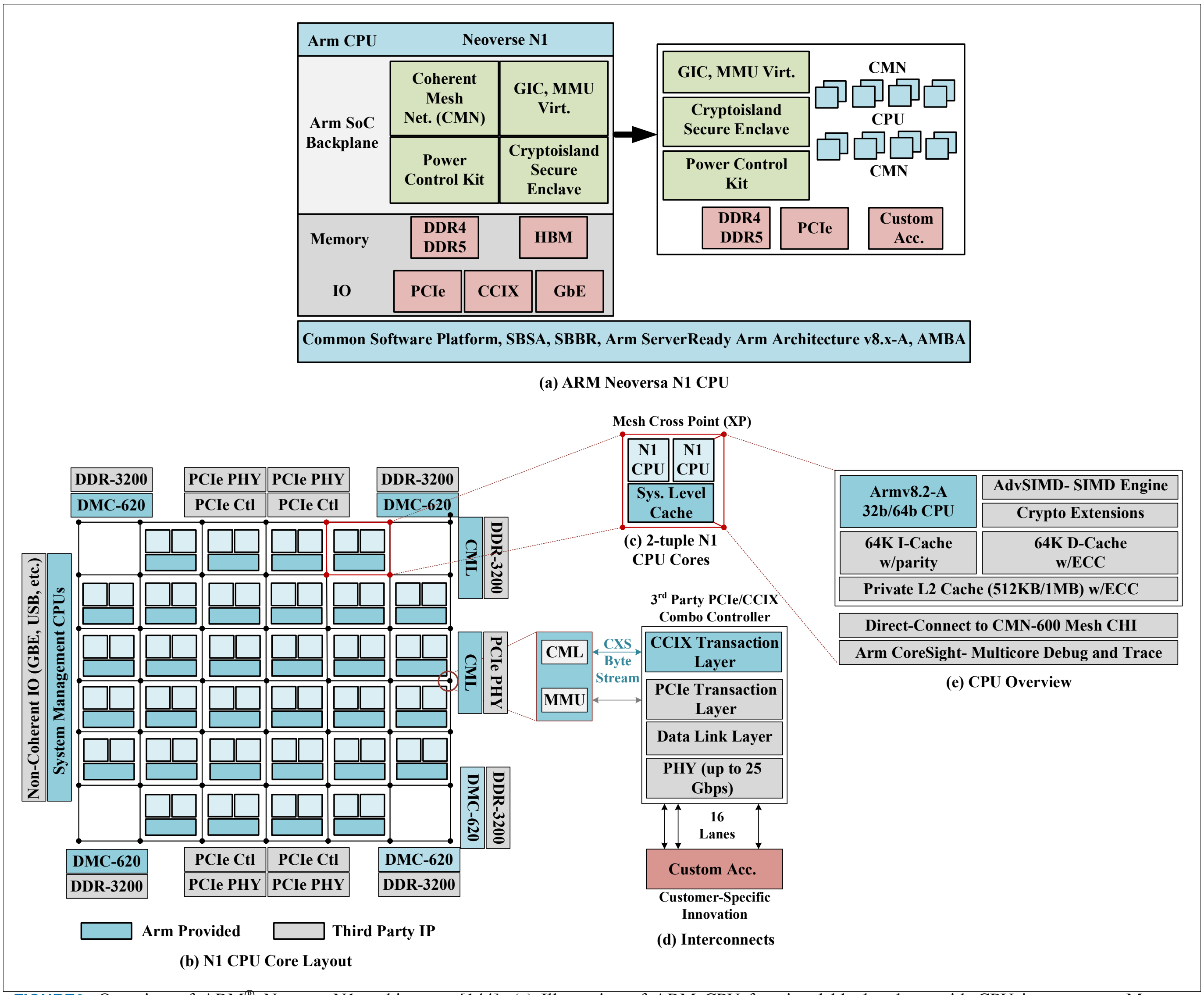

FIGURE6: Overview of ARM® Nervosa N1 architecture [144]: (a) Illustration of ARM CPU functional blocks along with CPU interconnect, Memory Management Unit (MMU), power management, and security components in relation to third-party memory and I/O components. Nervosa N1 can be extended to server-scale deployments with specifications of Server Base System Architecture (SBSA), Server Base Boot Requirements (SBBR), and Advanced Microcontroller Bus Architecture (AMBA) [149]. 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). The left part of (a) shows the general template arranged as layers of components, such as backplane, ARM CPUs, memory, and I/O devices; while the right part shows the actual scalable view, with a flexibly scalable number of CPUs on top of the CMN, supported by common functional blocks, such as virtualization, security, and power control. (b) Layout overview of N1 CPU cores supported by CMN, with extensions to memory controller and PCIe controllers for external memory and I/O device interfaces. (c) Two-tuple N1 CPU cores and system level cache surrounded by Mesh Cross Points (XPs). (d) Hardware interconnect and interface extensions along with PCIe protocol operations to connect external hardware accelerators with N1 CPUs. (e) Overview of N1 CPU with 64 KB L1 data and instruction cache, 512 KB/1 MB private L2 cache, mesh connect, and debug-and-trace components.

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. [116] 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 6 presents an overview of the Neoverse N1 [144] 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 $8 \times 8$ mesh fabric. The chip-to-chip connectivity (e.g., between CPU and accelerator) is enabled by the CCIX® (see Sec. III-B2e) 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 [145] 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.

#### 6) 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. III-A along with the general CPU technology trends in support of diverse application demands [196], [197] 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.

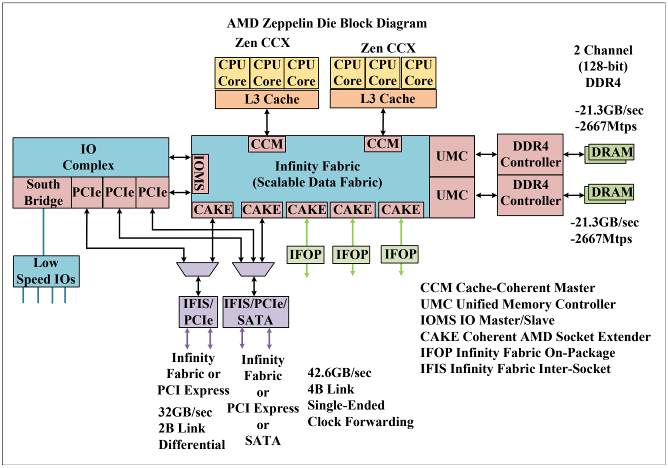
AMD Zeppelin Die Block Diagram
Zen CCX
CPU Core
L3 Cache
CCM
Infinity Fabric (Scalable Data Fabric)
IOMS
IO Complex
South Bridge
PCIe
CAKE
UMC
DDR4 Controller
DRAM
2 Channel (128-bit) DDR4
-21.3GB/sec -2667Mtps
IFOP
Low Speed IOs
IFIS/ PCIe
IFIS/PCIe/ SATA
Infinity Fabric or PCI Express
32GB/sec 2B Link Differential
Infinity Fabric or PCI Express or SATA
42.6GB/sec 4B Link Single-Ended Clock Forwarding
CCM Cache-Coherent Master
UMC Unified Memory Controller
IOMS IO Master/Slave
CAKE Coherent AMD Socket Extender
IFOP Infinity Fabric On-Package
IFIS Infinity Fabric Inter-Socket

FIGURE7: Overview of AMD® Zen core and Infinity core-to-core Fabric [198]. 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). In addition to inter-socket, CAKE enables intra-socket connections for die-to-die (on-package) with Infinity Fabric On-Package (IFOP) extensions. 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. CAKE interfaces can also be extended to I/O with support for PCIe.

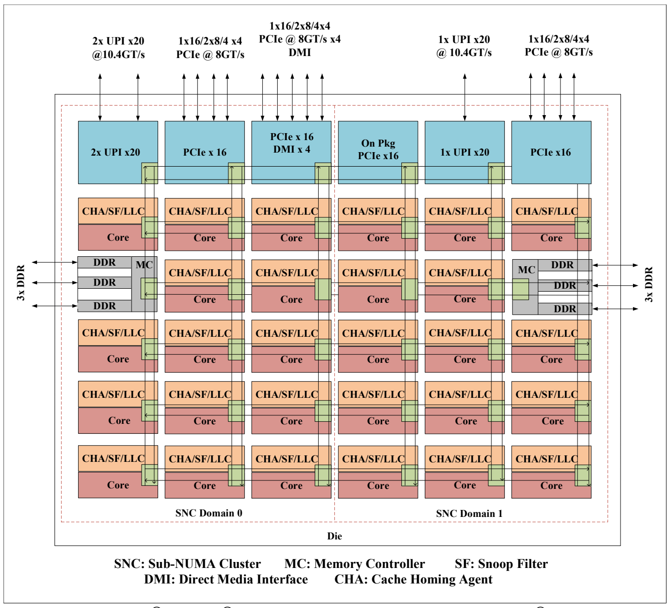

FIGURE8: Intel® Xeon® CPU overview [199]: The Intel® Xeon CPU in a single-socket package consisting of single die with 22 cores and 2 Memory Controllers (MCs) on either side of the die extending to DDR interfaces. The cores are arranged in a rectangular grid supported by a 2D mesh interconnect that connects 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. Interconnects, such as the Ultra-Path Interconnects (UPI), enable communication between multiple sockets, the DDR enables communication between DRAM and CPU, and the Peripheral Component Interconnect express (PCIe) enables communications between external components and the CPU.

## B. 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 [200].

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.

### 1) 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.

#### a: Scalable Data Fabric (SDF) & Scalable Control Fabric (SCF)

The Infinity Scalable Data Fabric (SDF) and Scalable Control Fabric (SCF) [33] (see Fig. 7) 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.

#### b: 2D Mesh

The Intel® 2D mesh [146] (see Fig. 8) 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. 8. 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. 7), and the 2D mesh (Fig. 8) 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.

#### c: Network on Chip (NoC)

A Network on Chip (NoC) [147] (see Fig. 9) 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

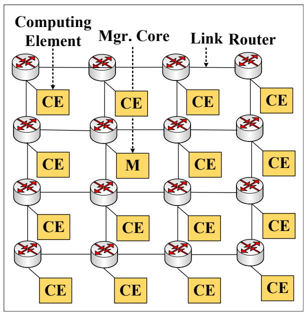

**FIGURE9:** Overview of Network on Chip (NoC) [147] where each Compute Element (CE) connects to a router: The NoC comprises a fabric of interconnects that provides on-chip communication to compute and memory elements which are connected to routers. The NoC provides homogeneous connection services as 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.

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.

### d: Intel® On-Chip System Fabric (IOSF)

The Intel® On-Chip System Fabric (IOSF) [148] provides a proprietary hierarchical switching fabric that connects multiple hardware components for on-chip communications. The IOSF characteristics include: *i*) Modular design: The IOSF can be applied and extended to multiple devices and applications by reusing and extending the IOSF design in the hardware components of the devices and applications; *ii*) Compatibility with the PCIe: The IOSF can convert PCIe transaction packets to IOSF packets by using a PCIe compatible switch; and *iii*) IOSF provides a sideband interface for error reporting and Design for Text/Debug (DFX) procedures.

### e: 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 [149] provides a generic interface for on-chip communication that flexibly connects various on-die components (see Fig. 10). 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

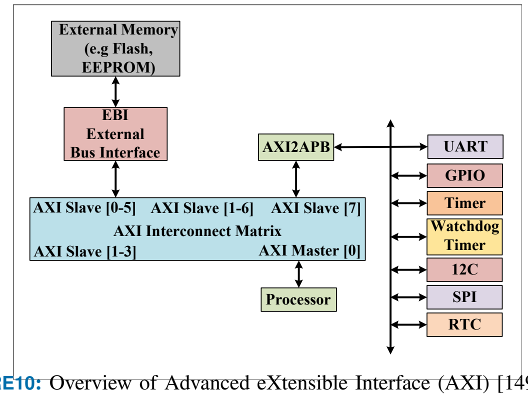

**FIGURE10:** Overview of Advanced eXtensible Interface (AXI) [149]: The AXI provides an on-chip fabric for communication between components. The AXI operates in a master and slave model, the slave nodes read and write data between components as directed by master nodes. The AXI also provide cache coherency with the AXI-Coherency Extension (ACE) specification [149] to keep the device cache coherent with CPU cores.

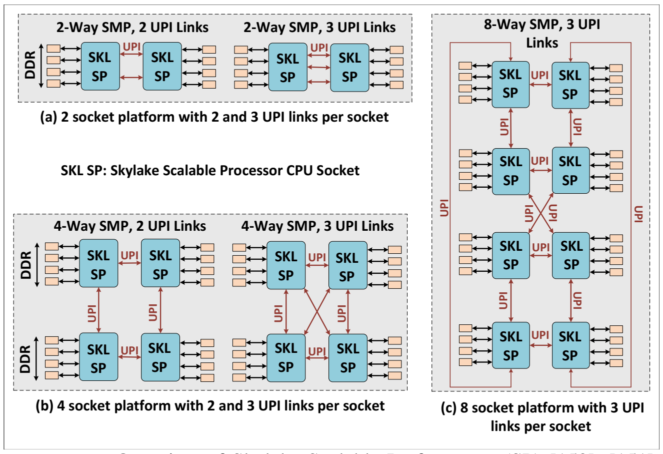
2-Way SMP, 2 UPI Links
2-Way SMP, 3 UPI Links
DDR
SKL SP
UPI
(a) 2 socket platform with 2 and 3 UPI links per socket
SKL SP: Skylake Scalable Processor CPU Socket
4-Way SMP, 2 UPI Links
4-Way SMP, 3 UPI Links
(b) 4 socket platform with 2 and 3 UPI links per socket
8-Way SMP, 3 UPI Links
(c) 8 socket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

**FIGURE11:** Overview of Skylake Scalable Performance (SP) [150], [151] 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.

*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.

### 2) 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.

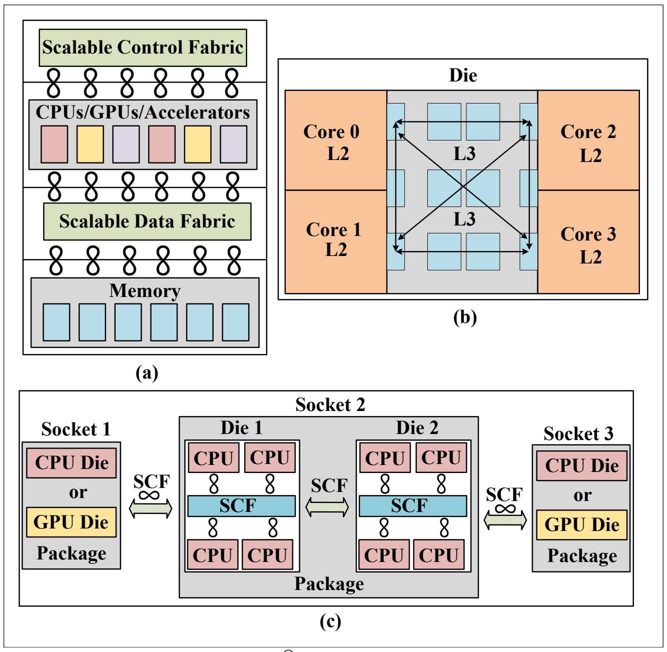

FIGURE12: Overview of AMD® Infinity Fabric [33], [152]–[154] for on-chip and chip-to-chip interconnects an accelerator chip: (a) shows the overview of interconnects between CPU and GPU through the Scalable Control Fabric (SCF), (b) shows the interconnects from core-to-core within a die for relative comparison, and (c) shows the overall fabric extensions at the socket, package, and die levels.

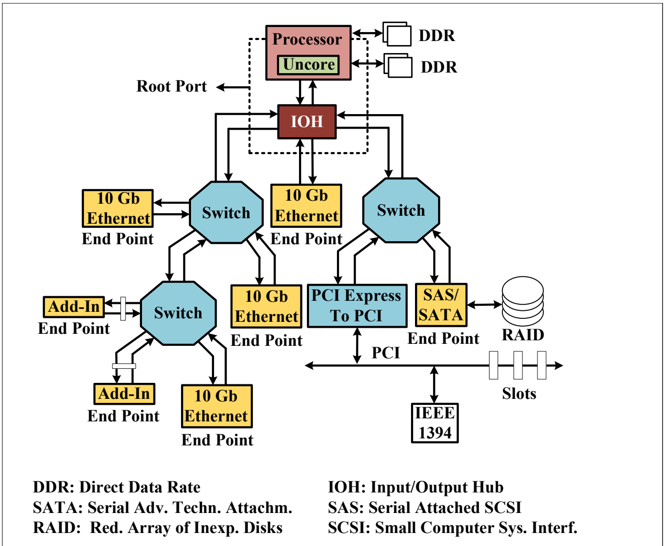

FIGURE13: Overview of Peripheral Component Interconnect express (PCIe) [155] interface which is an extension to PCI technology: PCI operated as a parallel bus with limited throughput due to signal synchronization among the parallel buses. The PCIe implements a serial communication per bus without any 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). 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.

#### a: Ultra Path Interconnect (UPI)

The Intel® Ultra Path Interconnect (UPI) [150], [151] 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. 11 for Intel® Skylake processors.

#### b: Infinity Fabric InterSocket (IFIS)

The Infinity Fabric InterSocket (IFIS) [33], [152]–[154] 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.

#### c: Peripheral Component Interconnect express (PCIe)

The Peripheral Component Interconnect express (PCIe) [155] (see Fig. 13) is a chip-to-chip interconnect and interface protocol that enables an external system-on-chip component, 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,

**TABLE2:** 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 |

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 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.

### d: Compute eXpress Link (CXL)

The Compute eXpress Link (CXL) [156] (see Fig. 14) presents a PCIe compliant interconnect focusing primarily on providing cache coherency across either side of the CXL link. The CXL link is targeted for accelerators on the platform as current chip-to-chip interconnects that connect accelerators do not support cache-to-cache coherency between the CPU LLC and the local cache of the accelerator. As the computing demands increase, there will be accelerators with large processing units (i.e., local CPU), large local memory (i.e., local to accelerator), and an associated local cache. As the PCIe does not provide any support to manage the coordination between the main CPU, memory, and cache, on the one hand, and the accelerator-local CPU, memory, and cache, on the other hand, the CXL extends the PCIe function to coordinate resources on either side of the CXL link. As a result, a diverse range from compute-intensive to memory-intensive accelerators can be flexibly supported over the CXL without compromising performance. The CXL provides speeds up to 3.938 GB/s for 1 lane and 63.01 GB/s for 16 lanes.

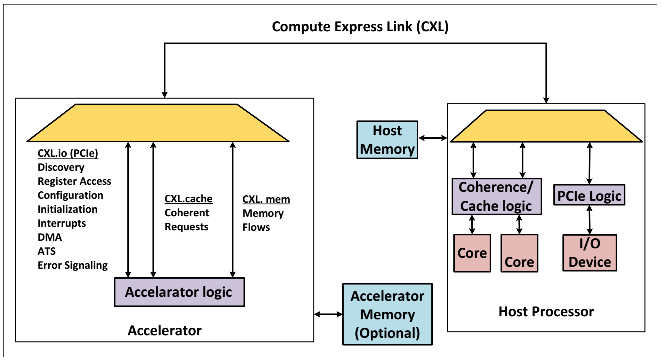

**FIGURE14:** Overview of Compute eXpress Link (CXL) [156] interconnect (which uses the PCIe as its interface): The CXL provides a protocol specification over the PCIe physical layer to support memory extensions, caching, and data transactions from I/O devices, while concurrently supporting the PCIe 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.

The CXL provides different protocol sets to coordinate data I/O, memory, and cache while being fully compatible with the PCIe. The `CXL.io` protocol defines the PCIe transaction procedures which are also used for discovery, enumeration, and error reporting, `CXL.mem` for memory access, and `CXL.cache` for caching mechanisms. Low latency and high bandwidth access to system resources are the key CXL advantages over the traditional PCIe. The CXL specifications also define a Flex Bus which supports a shared bus to commonly transport the CXL and PCIe protocols. The Flex Bus [157] mode of operation is negotiated during boot up as requested by the device/accelerator (an external component to the CPU). The Flex Bus operates on the PCIe electrical signal characteristics as well as the PCIe form factors of an add-in card.

### e: 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. III-A3), 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

**TABLE3:** Comparison of Cache Coherent Interconnects [201].

| Standard | PHY Layer | Topology | Unidirectional BW | Coherence |
|---|---|---|---|---|
| Compute eXpress Link (CXL) [156] | PCIe PHY | p2p Switched | 32–50 Gb/s (×16) | `CXL.cache` based coherency between processor and accelerator |
| Cache Coherent Interconnect for Accelerators (CCIX) [158] | PCIe PHY | p2p Switched | 32–50 Gb/s (×16) | Full cache coherency between processor and accelerator |
| Generation-Z (Gen Z) [159] | IEEE 802.3 Short & Long Haul PHY | p2p Switched | Signaling Rates: 16, 25, 28, and 56 GT/s; Mult. link widths: 1 to 256 lanes | Does not specify cache coherent agent operations, but does specify protocols that support cache coherent agents. |
| Open Coherent Accelerator Processor Interface (OpenCAPI) [161] | Bluelink 25 Gb/s PHY used for OpenCAPI & NVLINK | p2p | 25 Gb/s (×8) | Coherent access to memory cache coherence not supported until v4.0 |

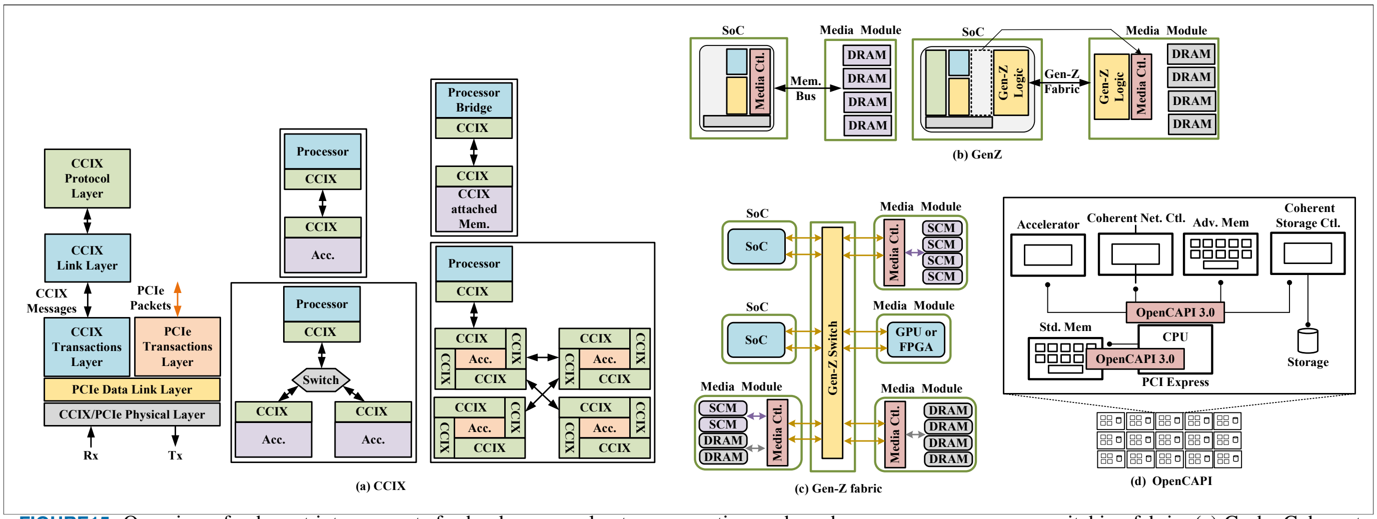

**FIGURE15:** Overview of coherent interconnects for hardware accelerators supporting cache coherency across common switching fabric: (a) Cache Coherent Interconnect for Accelerators (CCIX)® [158] defines a protocol to automatically synchronizes caches between CPU and I/O devices. (b) and (c) Gen-Z [159] 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) [160] homogeneously connects devices to a host platform with a common protocol to support coherency with memory, host interrupts, and exchange messages across devices.

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®) [158] (pronounced "See 6") is a new interconnect design and protocol definition to seamlessly connect computing nodes supporting cache coherency (see Fig. 15(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 15(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.

#### f: Generation-Z (Gen-Z)

The Gen-Z Consortium [159] (see Fig 15(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.

##### g: Open Coherent Accelerator Processor Interface (OpenCAPI)

The Open Coherent Accelerator Processor Interface (OpenCAPI) [160] (see Fig. 15(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 [161] 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 3.

#### 3) 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.

### C. 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 [202] 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.

#### 1) 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

**TABLE4:** 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. | DDR1 | DDR2 | DDR3 | DDR4 | DDR5 |
|---|---|---|---|---|---|
| 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 |

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.

#### a: I/O Acceleration Technology (I/OAT)

The Intel® I/O Acceleration Technology (I/OAT), as part of the Intel® QuickData Technology (QDT) [162], 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) [162].

### 2) 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 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 4. 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 [163]. The DDR5 also increases the total number of memory bank groups to twice of the DDR4, see Table 4. Overall, the DDR5 maximum data rate is twice the DDR4 maximum data rate, see Table 4. 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.

### 3) 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 [164] 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 [165], 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 Intelli-

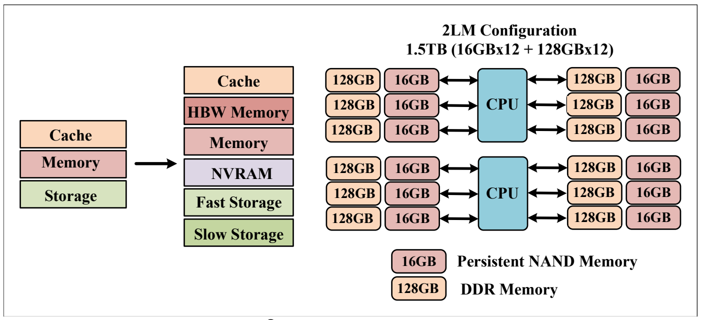

**FIGURE16:** Overview of Intel® Optane DC persistent Memory configured as 2 Level Memory (2LM) where the DRAM is used as cache to store only the most frequently accessed data and NVDIMM is used as an alternative to the DRAM with the Byte-Addressable Persistent Memory (B-APM) technique.

gence (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.

#### a: 1 Level Memory (1LM)

In the 1LM memory [166], [167] 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.

#### b: 2 Level Memory (2LM)

In the 2LM [167] mode (see Fig. 16), 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.

#### c: Application-Direct (DAX)

In the Application-Direct (DAX) [167], [168] mode, the NVDIMMs are seen as an independent PM memory type that can be used by the OS. The Non-Volatile RAM (NV-RAM) memory regions can be directly assigned to applications for direct access of memory through block level memory access by the memory controller to support the OS file system. In DAX mode, the applications and OS have to be PM memory aware such that dedicated CPU load and store instructions specific to PM memory access are used for the transactions between CPU and NVDIMMs. Essentially, applications on the OS see PM NVDIMMs in DAX mode as a storage memory space in the platform for OS file-system store usage. Traditional disk access by the application involves a kernel mode transition and disk I/O request and interrupt on completion of the disk read process which adds up as a significant overhead for storage-intensive applications. Therefore, PM offers an alternative to storage on NVDIMMs with block memory read capabilities close to the DDR DRAM access speeds.

#### d: 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 [169]. 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.

#### e: Asynchronous DRAM Refresh (ADR)

Asynchronous DRAM Refresh (ADR) [170] 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.

### 4) 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

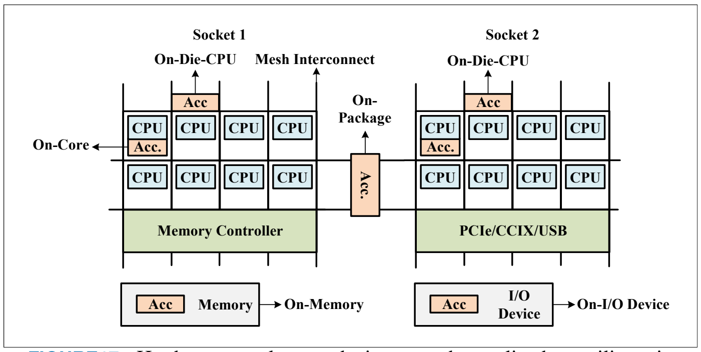

FIGURE17: Hardware accelerator devices can be realized on silicon in different placements: *i*) on-core, whereby the accelerator device is placed right next to a CPU core; *ii*) on-CPU-die, whereby the accelerator device is placed around the CPU mesh interconnects; *iii*) on-package, whereby the accelerator 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.

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.

## D. 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.

### 1) 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 17. 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.

### 2) 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. 18. 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

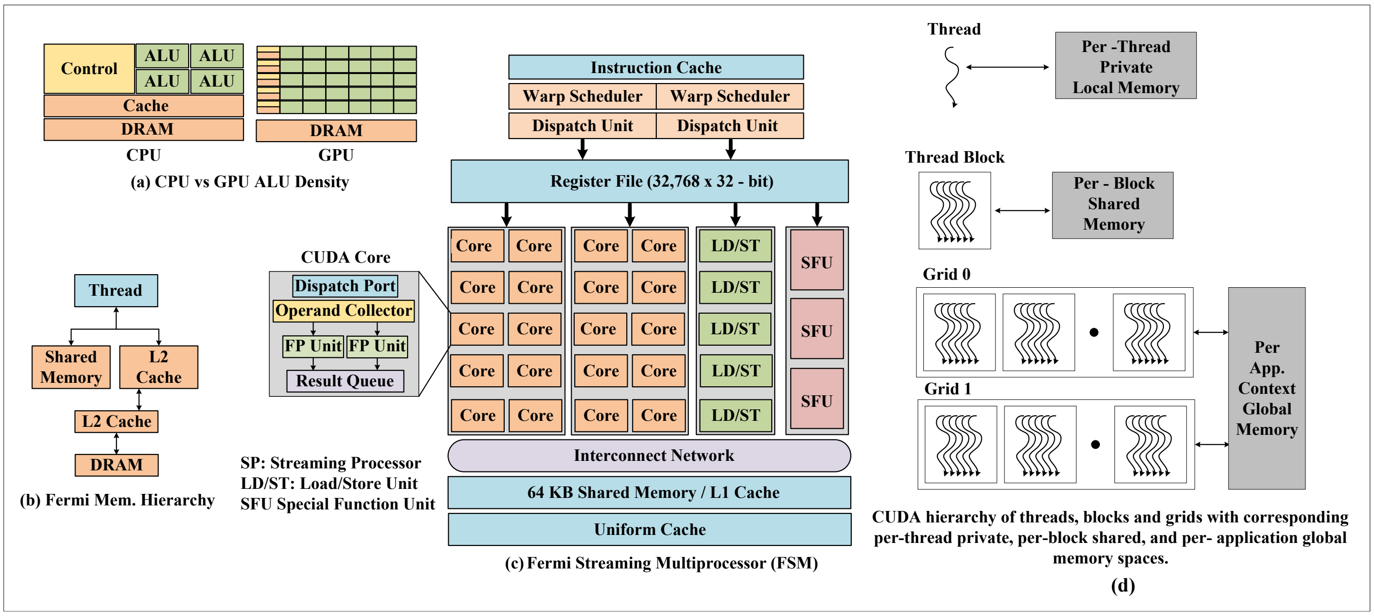
Control
ALU
ALU
ALU
ALU
Cache
DRAM
DRAM
CPU
GPU
(a) CPU vs GPU ALU Density
Thread
Shared Memory
L2 Cache
L2 Cache
DRAM
(b) Fermi Mem. Hierarchy
CUDA Core
Dispatch Port
Operand Collector
FP Unit
FP Unit
Result Queue
SP: Streaming Processor
LD/ST: Load/Store Unit
SFU Special Function Unit
Instruction Cache
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)

FIGURE18: 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 [171].

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 [171] (see Fig. 18) 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 [203]. 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.

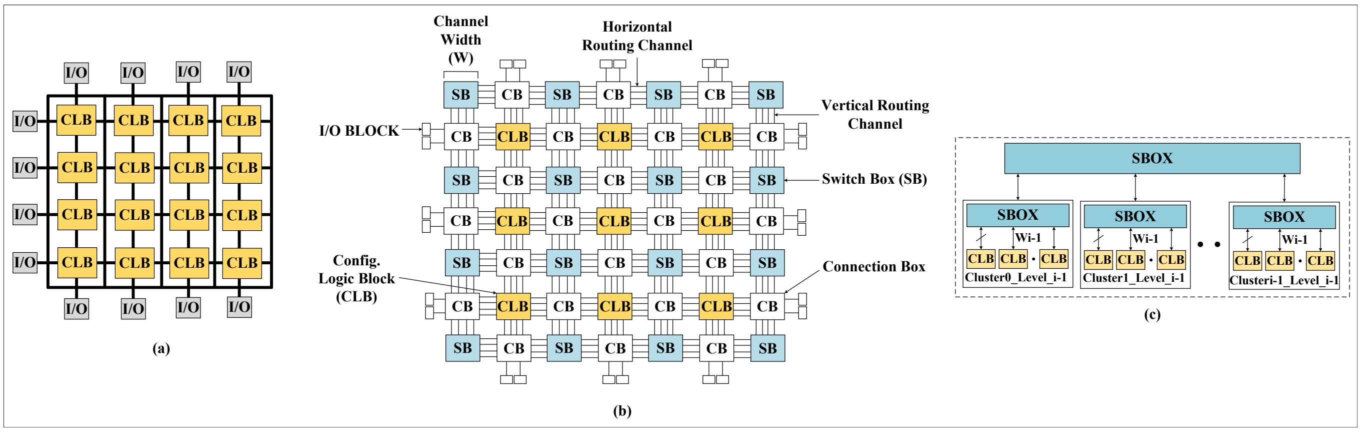

**FIGURE19:** (a) Overview of FPGA architecture: Configurable logic blocks (CLBs) are interconnected in a two-dimensional programmable routing grid, with I/O blocks at the grid periphery. (b) Illustration of a traditional island-style (mesh based) FPGA architecture with CLBs; the CLBs are "islands in 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 [172].

### 3) 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 [172]. As illustrated in Fig. 19, 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. 19, the FPGA switching network can be categorized into two basic forms: *i*) island-style routing (Fig. 19(b)), and *ii*) hierarchical routing (Fig. 19(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.

#### 4) 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 [204] 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 [205]. 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* [206] for run-time reconfiguration processes, and *personas* [207] 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.

### E. 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 [208] and graph processing [209]. 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.

#### 1) 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 [173] 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.

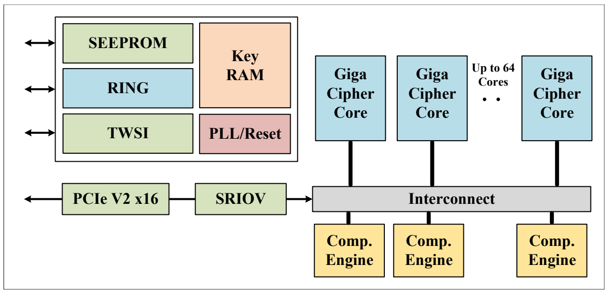

**FIGURE20:** Block diagram of Nitrox cryptography and compression accelerator [174]: 64 Giga cipher cores offer high throughput due to parallel processing, coupled with dedicated compression engines. The Nitrox hardware accelerator is external to the CPU and interfaces with the CPU via a PCIe interconnect.

#### a: Cavium Nitrox®

Nitrox [174] 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. III-B) 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) [84], [175] virtualization.

#### b: Intel® Quick Assist Technology®

Similarly, to address the cryptography and compression computing needs, the Intel® Quick Assist Technology® (QAT) [176] 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) [177]. 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.

**TABLE5:** 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<br>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 |

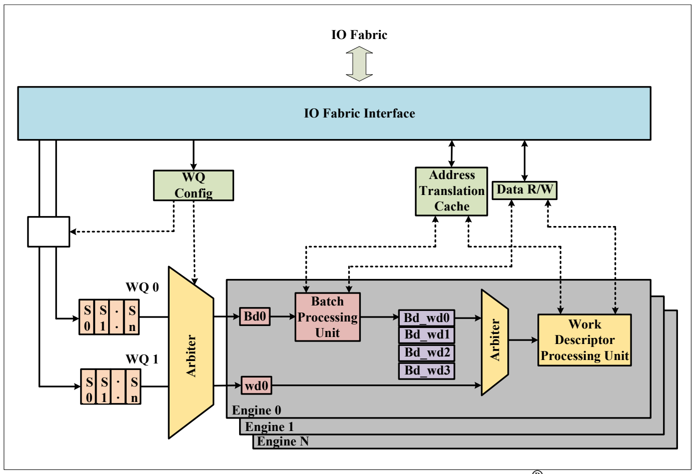

**FIGURE21:** Illustration of high-level blocks within the Intel® DSA device at a conceptual level. In DSA, the receiving of downstream work requests from clients and upstream work requests, such as read, write and address translation operations, are accessed with the help of I/O fabric interfaces. The 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 output [178].

### 2) 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 (DSA) [178] has been introduced. The DSA functions are summarized in Table 5, and the internal DSA blocks have been illustrated in Fig. 21.

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.

### 3) 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) [179] 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) [180] technology.

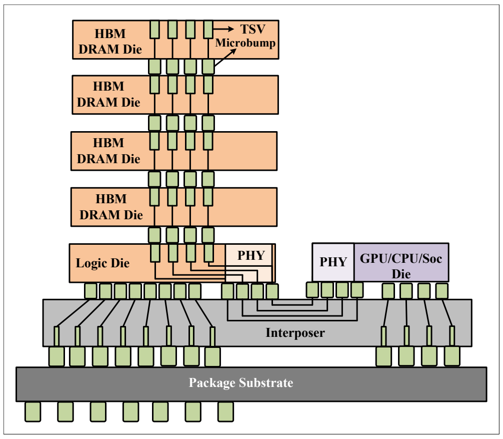

FIGURE22: Illustration of high-bandwidth memory (HBM) with low power consumption and ultra-wide bus width. Several HBM DRAM dies are vertically stacked (to shorten the propagation distance) and interconnected by "through-silicon vias (TSV)", while "microbumps" connect multiple DRAM chips [179] The vertically stacked HBMs are plugged into an interposer, i.e., an ultra-fast interconnect, which connects to a CPU or GPU [179].

### 4) Processing In-Memory (PIM) Accelerator

Likewise, to memory, bandwidths and latencies of external interconnects (e.g., PCIe) define the overall benefit of an accelerator (especially considering that the computing capacity of an accelerator can be relatively easily scaled up). The system data processing pipeline also involves the memory transactions between external components and the system memory unit (i.e., DRAM), which involves two interconnects, namely between the DDR and CPU, and between CPU and external component (e.g., PCIe). If the application that is being accelerated by an external hardware is data intensive and involves large memory transactions between the DRAM and the external hardware, then significant amounts of CPU/DMA cycles are needed for the data movement.

Processing-In-Memory (PIM) [181], [182] envisions to overcome the data movement problem between accelerator and memory by implementing an acceleration function directly on the memory. This PIM may seem to be a simple solution that solves many problems, including memory move and interconnect speeds. However, the current state-of-art of PIM is limited to basic acceleration functions that can be implemented on memory units under consideration of the packaging and silicon design challenges which require the 3D integration of acceleration function units onto the memory storage modules [183]. The current applications of PIM accelerations are large-scale graph processing with repeated memory updates as part of machine learning computations as well as neural network coefficient updates with simple operations involving multiplications and additions of data in static memory locations (for the duration of application run-time), such as matrix operations [184]. A PIM architecture proposed by Ahn et al. [181] achieved ten-fold throughput increases and saved 87% of the energy. NF application can potentially greatly benefit from in-memory computation which avoids packet movements between memory and accelerators.

#### 5) 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 [185], [186] 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 [187], 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.

#### 6) 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).

### F. INFRASTRUCTURE

#### 1) 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.

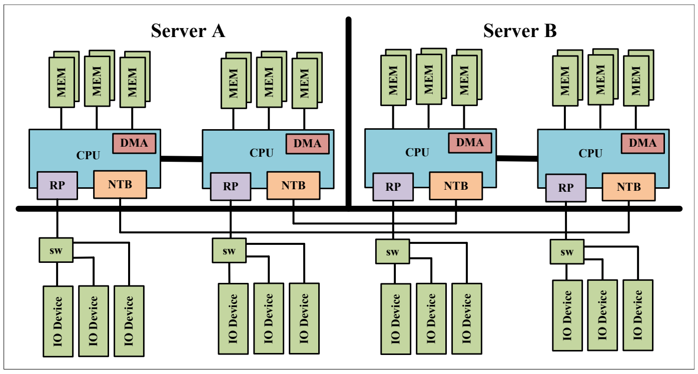

**FIGURE23:** Overview of Linux server with Non-Transparent Bridges (NTBs) [192]: The memory regions of servers A and B can be inter-mapped across platforms to appear as their own physically addressed memory regions. An NTB physically interconnects platforms in 1 : 1 fashion through a PCIe physical interface. 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.

A Smart-Network Interface Controller (SmartNIC) [188], [189] 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 [190] while supporting advanced protocols, such as Infiniband and Remote-DMA (RDMA) [191].

### 2) 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) [192] 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).

### 3) 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 III-A1), or system memory capabilities, such as NV-NAND persistent memory (Section III-C3). 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.

### G. SUMMARY AND DISCUSSION

In Sec. III 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.

## IV. 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. 24.

### A. 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.

#### 1) 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® [210], [211] 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. 25.

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 6 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

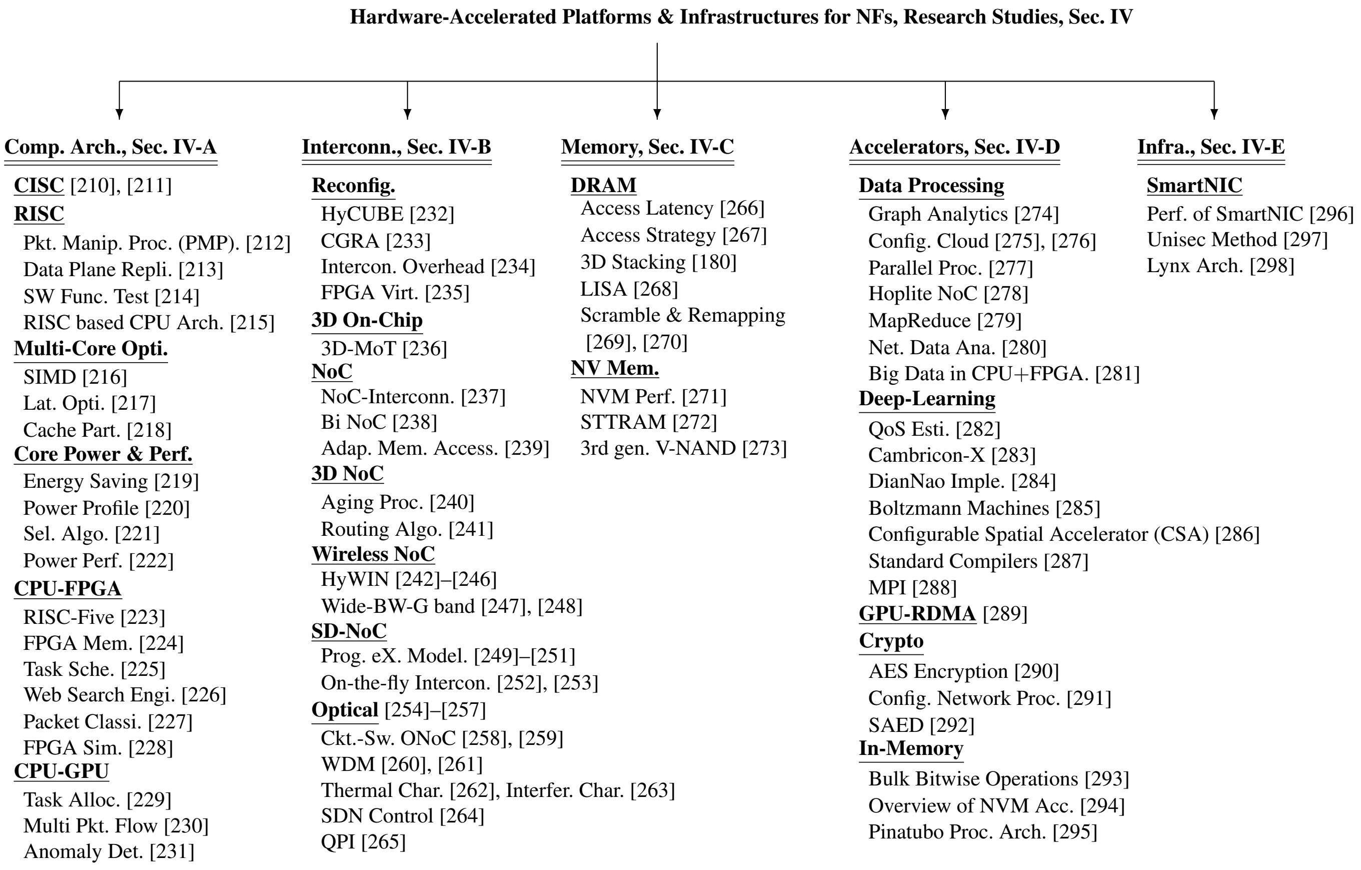

**FIGURE24:** Classification taxonomy of research studies on hardware-accelerated platforms and infrastructures for processing softwarized NFs.

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.

**TABLE6:** 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 |
|---|---|---|---|---|
| L1I | 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 |

### 2) 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. III-E), due to the power and space efficient RISC architectural characteristics.

Typically, network applications involve direct packet pro-

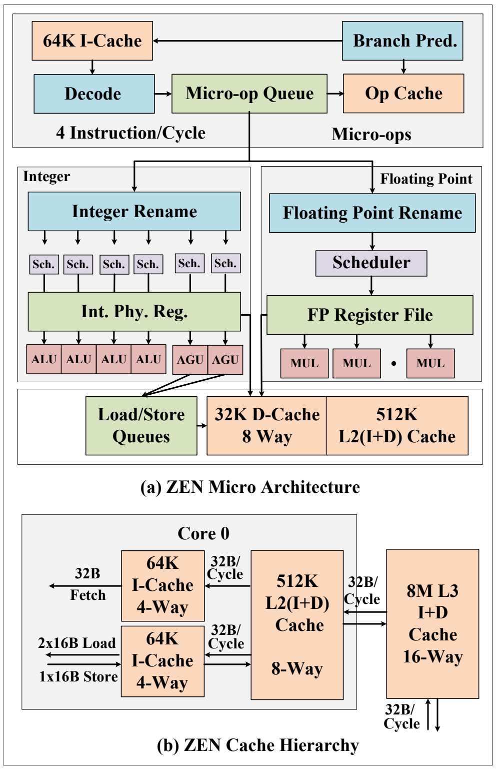

FIGURE25: (a) Overview of ZEN Micro architecture [210], [211]: The Zen micro architecture has 3 modules: *i*) Front End Module, *ii*) Integer and Floating-Point Modules, and *iii*) Memory Subsystem Module. Each core performs instruction fetching, decoding (decodes 4 instructions/cycle into the micro-op 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.

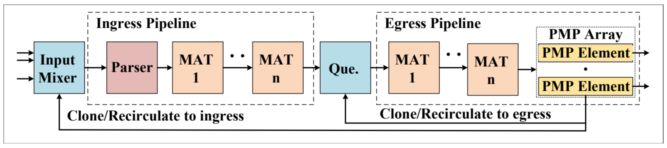

FIGURE26: An overview of the RISC based Packet Manipulation Processor (PMP) [212] which implements a programmable packet header matching table based on atomic operations. The table can be dynamically updated by multiple processes running on the CPU without impacting the matching operations.

cessing 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 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. [212] 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. 26 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. [213] 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. [214] 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 [214] 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. [215]. Morais et al. [215] 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.

### 3) 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 [216]. 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. [217] have proposed a latency optimization for SIMD operations in multi-core systems based on Ant-Colony Optimization (ACO). The Zhou et al. [217] 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 [217] 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. [218] 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.

### 4) 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. [219] 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 [219] 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. [220] 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. [220] 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. [221]. Begum et al. [221] 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. [222] 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

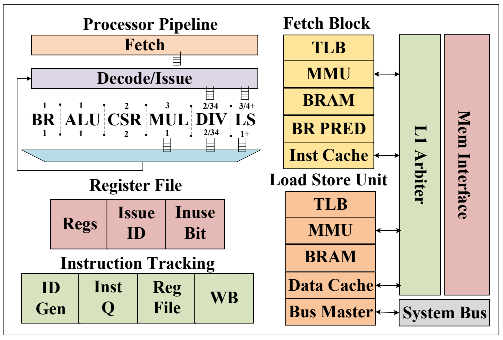

FIGURE27: Taiga computing architecture with reconfigurable design using FPGA [223]. The compute logic units, such as ALU, BRanch unit (BR), Multiply (MUL), and Division (DIV), are implemented with independent circuitry, i.e, with Instruction Level Parallelism (ILP). Block RAM (BRAM) and BRanch PREDiction (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.

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.

#### 5) 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. [223] (see Fig. 27) 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$\times$ 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 [223] successfully validated the processor configurations and identified the critical paths in the design: The write-back

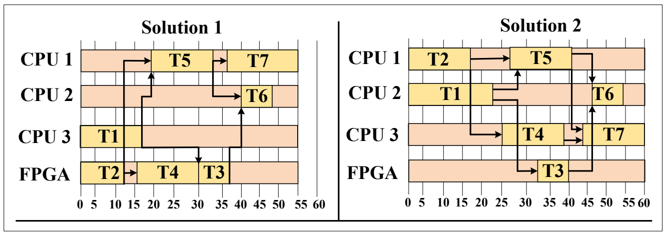

**FIGURE28:** Illustration of heterogeneous scheduling between CPU and FPGA where different tasks are commonly scheduled relative to number of clock cycles [225]: Solutions 1 and 2 are possible scheduling paths for tasks (T1–T7) among CPUs and FPGA homogeneously. The optimization algorithm evaluates all possible paths and estimates the best path in terms of lowest overall processing latency and power.

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. [224] 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 $\mu$s. 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 $\mu$s 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. [225] (see Fig. 28) 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. [226] 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. [227] 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. [227] 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. [228] 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.

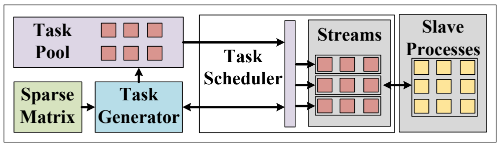

FIGURE29: Overview of dynamic task scheduling framework [229] for CPU-GPU heterogeneous architecture: Tasks are partitioned into parallel (GPU) execution tasks and single-threaded (CPU) execution tasks and are then scheduled as GPU and CPU slave processes. The streams organize the tasks such that the inter-scheduling intervals of tasks are minimized.

#### 6) CPU-GPU

Similar to the study by Abdallah et al. [225], Nie et al. [229] (see Fig. 29) 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 [230]. However, the performance characterizations by Yi et al. [230] 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 [231]; 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 [231] 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 [231] which is applied in packet filtering and DoS attack mitigation.

#### 7) 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 [299]–[301] 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.

### *B. 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 4), 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 III-E3]; for example, the total bandwidth scales up to 512 Gbps for a 4 stack die with 128 Gbps bandwidth per die [302]. 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. [303], [304] 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. [305] have adapted PCIe links to flexibly interface the accelerators with the compute nodes (25 Gb/s) to support NF applications, such as cognitive computing.

### 1) 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. [232] have proposed HyCUBE, a reconfigurable multi-hop interconnect, see Fig. 30. 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 [233]. 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. [234] 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. [235] 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 [235]. 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.

### 2) 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. [236] 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. 31. 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 [236] 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.

### 3) 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

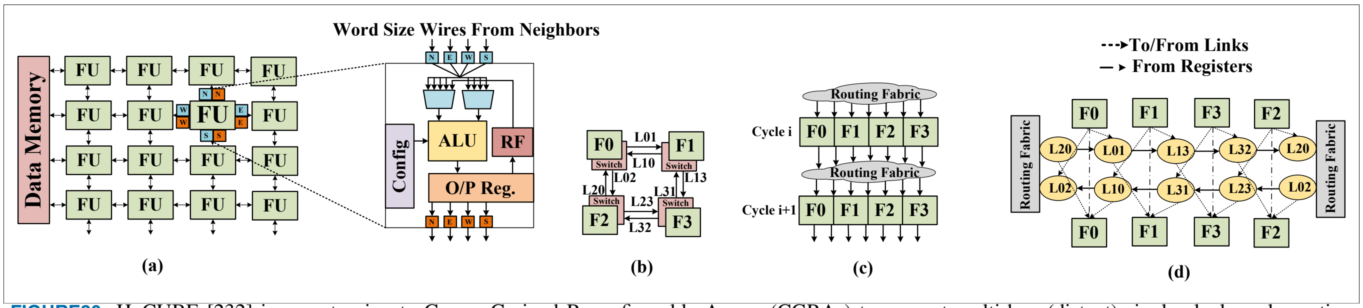

**FIGURE30:** HyCUBE [232] is an extension to Coarse-Grained Reconfigurable Arrays (CGRAs) to support multi-hop (distant) single clock cycle routing between Functional Units (FU): (a) Illustration of $4 \times 4$ CGRA interconnected by 2D mesh, (b) FU placement on routing fabric with bidirectional link 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.

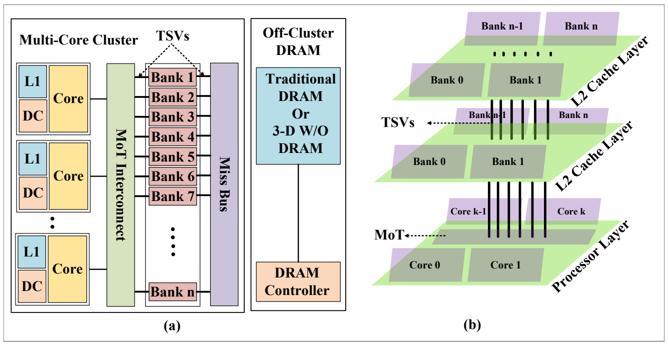

**FIGURE31:** (a) Overview of Mesh-of-Trees (MoT) based interconnect over 3D multi-core cluster with L2 cache stacking facilitated by TSVs. Each simple core in the multi-core cluster contains its own L1 instruction cache and data cache (DC). Multiple SRAM banks that are connected with the 3D MoT interconnect 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 [236].

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. [237], 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. [238] 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. [239] 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.

### 4) 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. [240] 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

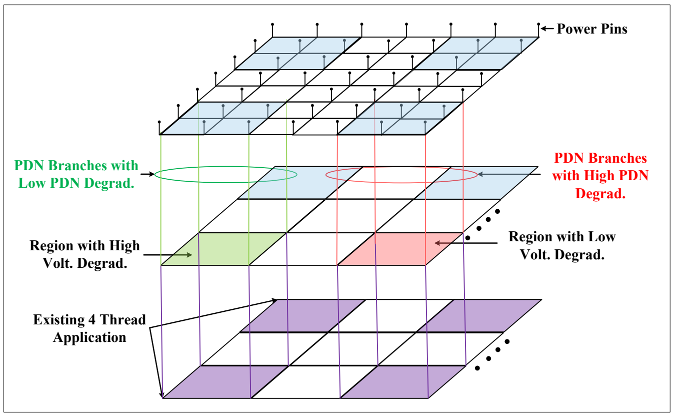
Power Pins
PDN Branches with Low PDN Degrad.
PDN Branches with High PDN Degrad.
Region with High Volt. Degrad.
Region with Low Volt. Degrad.
Existing 4 Thread Application

FIGURE32: 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 [240] considers both PDN and voltage degradations.

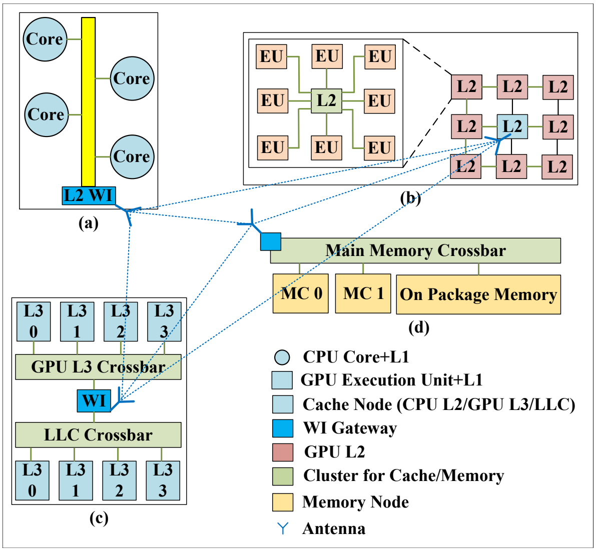

FIGURE33: Illustration of wireless NoC HyWin [242]: (a) The CPU subsystem with CPU cores (along with their respective L1 caches) is connected to a BUS interface; the L2 cache is shared between all CPU cores. (b) The GPU subsystem with shared L2 cache at the center connects multiple execution units in 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.

to uneven scheduling of computing tasks, leading to uneven aging of the 3D NoC, as illustrated in Fig. 32. 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. [241] 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. [241] 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.

#### 5) 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. [242] have proposed a Hybrid Wireless NoC (HyWin), as illustrated in Fig. 33, 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 [242] 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 [243], while other recent related studies have examined scalability [244], low-latency [245], and energy efficiency [246].

Similarly, for planar interconnected circuits (commonly used for chip-to-chip packaging), Yu et al. [247] 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 [248].

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.

#### 6) 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 [306]. 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. [249], [250] 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 [307], 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 [308], [309] has been examined in [251].

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. [252] 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 [253].

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.

#### 7) 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 [254]–[256]. SiPh offers solutions for both on-chip and chip-to-chip interconnects. For instance, Hsu et al. [257] 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. [258] 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. [258] have proposed a Multiple Ring-based Optical NoC (MRONoC) design which uses ring based routing, as well as redundant paths to reuse the wavelength resources without contentions. (A related circuit-switched ONoC with a hierarchical structure based on a Clos-Benes topology has been examined in [259].) 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 [258] 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 [260], while NoC wavelength routing has been studied in [261]. Moreover, recent studies have explored the thermal characteristics of ONoCs [262], the interference characteristics in optical wireless NoCs [263], and the SDN control for optical interconnects [264].

Further evolutions of integrated photonics and optical interconnects have been applied in quantum computing technologies. Wang et al. [265] 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.

#### 8) 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 [310]. 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 [311]. 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) [253] 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.

### C. MEMORY

#### 1) 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. [266] 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. [266] 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. [266] 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 [266] have shown nearly 20% reduction of the average latencies.

Utilizing similar techniques to reduce DRAM access latency, Hassan et al. [267] 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 [267] 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. [180] 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. [268] 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 [269], [270]. 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.

### 2) 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. [271]. 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) [272], Resistive Random Access Memory (RRAM), and Ferroelectric Field Effect Transistor (FeFET). The investigative study of Chen et al. [271] 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. [273] 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. [273] 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.

### 3) 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. III-C) 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) [312]. 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) [313]. Illustrating the benefits of integrated DRAM, Zhang et al. [314] have proposed a method to integrate memory cells into compute modules (i.e., CPUs and accelerators) based on phase change memory (PRAM) modules [315]. 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 [314].

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.

### D. ACCELERATORS

#### 1) Data Processing Accelerators

Specialized hardware accelerators can significantly improve the performance and power efficiency of NFs. Ozdal et al. [274] 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. [274] 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. [275], [276]. 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 [277] 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 [278]. 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 [279] and network data analytics (e.g., related to users, nodes, as well as cost and efficiency of end-to-end paths) [280], 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. [281] 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.

### 2) 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 [282]. 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. [283] 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× the throughput of the state-of-the-art DianNao implementation [284], while Cambricon-X is 6.4× more energy efficient.

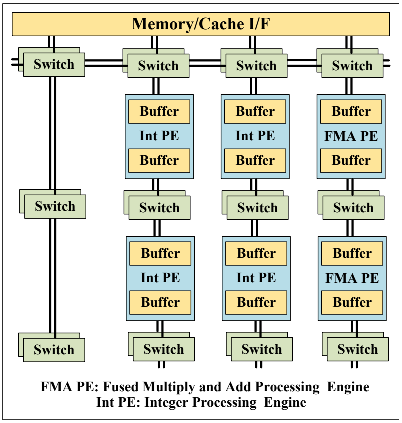
Memory/Cache I/F
Switch
Buffer
Int PE
FMA PE
FMA PE: Fused Multiply and Add Processing Engine
Int PE: Integer Processing Engine

FIGURE34: 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 [286]: 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.

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 [283] 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. [285] 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 [316] 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. III-E4), 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 runtime) 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) [286] architecture (see Fig. 34) 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 [287]. 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 34 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 intercommunicating 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. [288] 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.

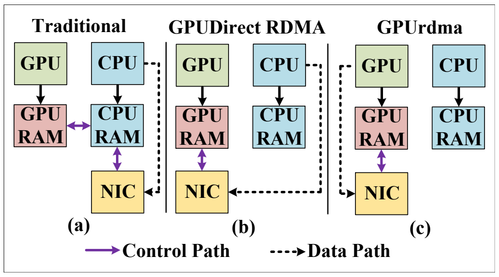

FIGURE35: Evolution of GPU-RDMA techniques [289]: (a) traditional method of GPU accessing RDMA with assistance from CPU, (b) GPU accesses RDMA directly from NIC, but CPU still performs the connection management for the GPU, and (c) GPU interacts with NIC independent of CPU, thereby reducing the CPU load for GPU RDMA purposes.

### 3) 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. III-F2) 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. [289] 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. 35(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 [289]. The evaluations showed 4.5-fold faster processing as compared to the CPU managing the remote data for the GPU.

### 4) 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. [290] 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 [291] 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. [292] 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. III-E1). 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.

### 5) 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. [293] 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 [293] 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. [294] 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. [295] 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 $1.12\times$ overall latency reductions and $1.11\times$ energy savings compared to a conventional CPU for data intensive graph processing and database applications.

### 6) 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 [317] 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 [318]–[320]. 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 [321], [322]. 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 [183] supporting HBM-in-memory acceleration.

## E. INFRASTRUCTURE

### 1) SmartNIC

SmartNICs enable programmability of the NICs to assist NFs by enabling hardware acceleration in the packet processing pipeline. Ni et al. [296] 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. [297] 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. 36. 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.

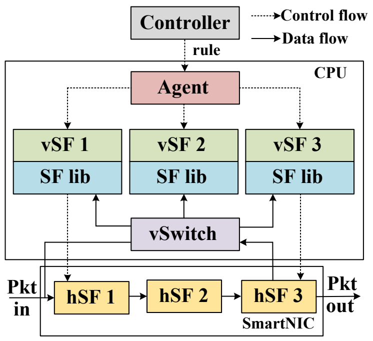

FIGURE36: UniSec implements a unified programming interface to configure and utilize the security functions implemented on SmartNICs [297]. SmartNICs implement hardware Security Functions (hSF) which are exposed to applications by virtual Security Functions (vSF) through a UniSec Security Function (SF) library.

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. [298] 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 37 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. [298] where GPUs communicate with an external (remote) database through SmartNICs show 25% system throughput

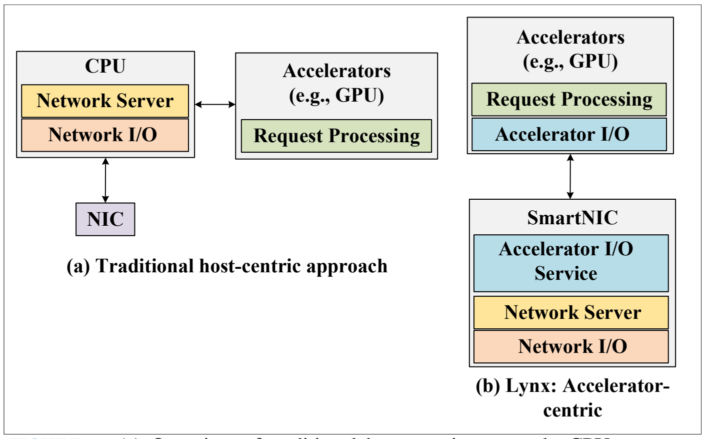

FIGURE37: (a) Overview of traditional host-centric approach: CPU runs the software that implements network-server function (for interacting with remote client nodes) to process the requests from a remote node (over the network). CPU also runs the Network I/O, which implements the network stack processing; (b) Overview of Lynx architecture [298]: 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.

increases for a neural network workload as compared to the traditional CPU service requests to the GPU accelerator.

#### 2) 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. [323] have implemented a programmable Intrusion Detection System (IDS) and packet firewall based on an FPGA embedded in the SmartNIC. Belocchi et al. [324] 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.

### F. 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.

## V. 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.

## A. OVERARCHING GRAND CHALLENGES

### 1) 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. IV-C1), and accelerator scheduling. In summary, when improving the platform and infrastructure capabilities, the complexity of the overall system should still be reasonable.

### 2) 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 [325]. 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.

### 3) 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.

### 4) Power and Performance

Zhang et al. [326] 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.

## B. CPU AND COMPUTING ARCHITECTURE

### 1) 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.

#### 2) 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 [327]), 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.

#### 3) 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. IV-A5) 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.

#### 4) API based Resource Control

Although there exists frequency control technologies and strategies (see Sec. III-A4), 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.

### C. INTERCONNECTS

#### 1) 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. III-B). 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.

### D. MEMORY

#### 1) Heterogeneous Non-Uniform Memory Access (NUMA)

Sieber et al. [328] 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.

#### 2) 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. IV-D5) 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.

### *E. ACCELERATORS*

#### 1) 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. III-B1) 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) [329].

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. III-D and III-E). 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.

#### 2) 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) [175], 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.

## F. INFRASTRUCTURE

### 1) SmartNIC Offline Processing Without CPU

Traditional systems process packets in two modes (see Sec. II-F1): *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 [330]. 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.

## G. NF ACCELERATION BEYOND PLATFORMS AND INFRASTRUCTURES

### 1) 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 [331]. Therefore, several software optimizations, such as zero copy and kernel bypass, as well as hardware acceleration strategies, such as in-line processing [189], 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) [84], [175], 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.

### 2) 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) [332], 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 [333] 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) [334], [335]. 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) [28], protocol operations (e.g., NF application protocols, such as HTTP and REST, as well as higher layer protocol operations, such as firewalls [336], 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) [337], and hardware accelerations [338], [339] that should be covered in a future survey focused specifically on the acceleration of orchestration and protocols.

## VI. 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 V.

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 V-G, 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` [327], enable applications to submit acceleration tasks directly to hardware accelerators, eliminating the software management (abstraction) overhead and latency.

## REFERENCES

[1] H. Ballani, 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.*, vol. 45, no. 4, pp. 493–507, 2015.

[2] U. C. Kozat, 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*. Wiley, Hoboken, NJ, 2020, pp. 137–167.

[3] R. Mijumbi, 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*, vol. 18, no. 1, pp. 236–262, 2015.

[4] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca, "The dynamic placement of virtual network functions," in *Proc. IEEE Network Operations and Management Symp.*, 2014, pp. 1–9.

[5] J. Anderson, 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.*, 2016, pp. 1–7.

[6] R. Amin, M. Reisslein, and N. Shah, "Hybrid SDN networks: A survey of existing approaches," *IEEE Commun. Surv. & Tut.*, vol. 20, no. 4, pp. 3259–3306, 2018.

[7] J. H. Cox, J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley, and H. L. Owen, "Advancing software-defined networks: A survey," *IEEE Access*, vol. 5, pp. 25 487–25 526, 2017.

[8] H. Farhady, H. Lee, and A. Nakao, "Software-defined networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[9] E. Kaljic, A. Maric, P. Njemcevic, and M. Hadzialic, "A survey on data plane flexibility and programmability in Software-Defined Networking," *IEEE Access*, vol. 7, pp. 47 804–47 840, 2019.

[10] A. M. Alberti, 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*, vol. 7, pp. 165 748–165 778, 2019.

[11] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[12] R. Roseboro, "Cloud-native NFV architecture for agile service creation & scaling," Jan. 2016, https://www.mellanox.com/related-docs/whitepapers/wp-heavyreading-nfv-architecture-for-agile-service.pdf, Last accessed May 19, 2020.

[13] S. D. A. Shah, M. A. Gregory, S. Li, and R. Fontes, "SDN enhanced multi-access edge computing (MEC) for E2E mobility and QoS management," *IEEE Access*, vol. 8, pp. 77 459–77 469, 2020.

[14] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2017.

[15] J. Liu and Q. Zhang, "Offloading schemes in mobile edge computing for ultra-reliable low latency communications," *IEEE Access*, vol. 6, pp. 12 825–12 837, 2018.

[16] M. Mehrabi, 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*, vol. 7, pp. 166 079–166 108, 2019.

[17] L. Tang and H. Hu, "Computation offloading and resource allocation for the internet of things in energy-constrained MEC-enabled HetNets," *IEEE Access*, vol. 8, pp. 47 509–47 521, 2020.

[18] Z. Xiang, 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.*, vol. 37, no. 5, pp. 1098–1116, 2019.

[19] H. Wang, Z. Peng, and Y. Pei, "Offloading schemes in mobile edge computing with an assisted mechanism," *IEEE Access*, vol. 8, pp. 50 721–50 732, 2020.

[20] Q. Zhang, 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)*, 2018, pp. 178–185.

[21] K. R. Dinakar, "A survey on virtualization and attacks on virtual machine monitor (VMM)," *Int. Research J. of Eng. and Techn. (IRJET)*, vol. 6, no. 3, pp. 6558–6563, Mar. 2019.

[22] S. Wang, J. Xu, N. Zhang, and Y. Liu, "A survey on service migration in mobile edge computing," *IEEE Access*, vol. 6, pp. 23 511–23 528, 2018.

[23] M. Alsaeedi, M. M. Mohamad, and A. A. Al-Roubaiey, "Toward adaptive and scalable OpenFlow-SDN flow control: A survey," *IEEE Access*, vol. 7, pp. 107 346–107 379, 2019.

[24] A. Binsahaq, T. R. Sheltami, and K. Salah, "A survey on autonomic provisioning and management of QoS in SDN networks," *IEEE Access*, vol. 7, pp. 73 384–73 435, 2019.

[25] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges," *Proc. IEEE*, vol. 107, no. 4, pp. 711–731, 2019.

[26] P. Shantharama, 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*, vol. 6, pp. 57 545–57 561, 2018.

[27] M. Wang, 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*, vol. 8, no. 9, pp. 937.1–937.28, 2019.

[28] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and software-defined networking," *Proc. IEEE*, vol. 103, no. 7, pp. 1102–1124, 2015.

[29] A. Kavanagh, "OpenStack as the API framework for NFV: The benefits, and the extensions needed," *Ericsson Review*, vol. 2015-3, pp. 1–8, Apr. 2015.

[30] A. Martí Luque, "Developing and deploying NFV solutions with OpenStack, Kubernetes and Docker," Master's thesis, Universitat Politècnica de Catalunya, 2019.

[31] H. R. Kouchaksaraei and H. Karl, "Service function chaining across OpenStack and Kubernetes domains," in *Proc. ACM Int. Conf, on Distr, and Event-based Sys.*, 2019, pp. 240–243.

[32] M. Arafa, 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*, vol. 39, no. 2, pp. 29–36, 2019.

[33] K. Lepak, 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, 2017, pp. 1–26.

[34] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Fast packet processing: A survey," *IEEE Commun. Surv. & Tut.*, vol. 20, no. 4, pp. 3645–3676, 2018.

[35] J. Garay, J. Matias, J. Unzilla, and E. Jacob, "Service description in the NFV revolution: Trends, challenges and a way forward," *IEEE Communications Magazine*, vol. 54, no. 3, pp. 68–74, 2016.

[36] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in NFV environments," in *Proc. IEEE Int. Conf. and Workshops on Networked Sys. (NetSys)*, 2015, pp. 1–5.

[37] J. Ordonez-Lucena, 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*, vol. 55, no. 5, pp. 80–87, 2017.

[38] Intel Corp., "Data Plane Development Kit (DPDK)," 2014.

[39] E. Nurvitadhi, 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)*, 2016, pp. 1–4.

[40] G. Pongrácz, 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.*, 2013, pp. 62–67.

[41] S. Gallenmüller, 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*, 2015, pp. 29–38.

[42] S. Makineni and R. Iyer, "Performance characterization of TCP/IP packet processing in commercial server workloads," in *Proc. IEEE Int. Conf. on Commun.*, 2003, pp. 33–41.

[43] P. Emmerich, 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)*, 2015, pp. 1–8.

[44] C.-H. Chou and L. N. Bhuyan, "A multicore vacation scheme for thermal-aware packet processing," in *Proc. IEEE Int. Conf. on Computer Design*, 2015, pp. 565–572.

[45] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[46] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: A survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[47] A. U. Rehman, R. L. Aguiar, and J. P. Barraca, "Network functions virtualization: The long road to commercial deployments," *IEEE Access*, vol. 7, pp. 60 439–60 464, 2019.

[48] T. Wood, K. Ramakrishnan, J. Hwang, G. Liu, and W. Zhang, "Toward a software-based network: Integrating software defined networking and network function virtualization," *IEEE Network*, vol. 29, no. 3, pp. 36–41, 2015.

[49] J. G. Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Trans. on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[50] W. Yang and C. Fung, "A survey on security in network functions virtualization," in *Proc. IEEE NetSoft Conf. and Workshops (NetSoft)*, 2016, pp. 15–19.

[51] I. Farris, T. Taleb, Y. Khettab, and J. Song, "A survey on emerging SDN and NFV security mechanisms for IoT systems," *IEEE Commun. Surveys & Tutorials*, vol. 21, no. 1, pp. 812–837, 2018.

[52] M. Pattaranantakul, 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*, vol. 20, no. 4, pp. 3330–3368, 2018.

[53] S. Lal, T. Taleb, and A. Dutta, "NFV: Security threats and best practices," *IEEE Commun. Magazine*, vol. 55, no. 8, pp. 211–217, 2017.

[54] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *J. Network and Computer Applications*, vol. 75, pp. 138–155, 2016.

[55] X. Li and C. Qian, "A survey of network function placement," in *Proc. IEEE Consumer Commun. & Netw. Conf. (CCNC)*, 2016, pp. 948–953.

[56] G. Miotto, 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.*, vol. 10, no. 1, pp. 3.1–3.19, 2019.

[57] M. Pattaranantakul, 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.* Springer, Cham, Switzerland, 2019, pp. 287–301.

[58] M. S. Bonfim, K. L. Dias, and S. F. Fernandes, "Integrated NFV/SDN architectures: A systematic literature review," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 114:1–114:39, 2019.

[59] A. Ghosh, A. Maeder, M. Baker, and D. Chandramouli, "5G evolution: A view on 5G cellular technology beyond 3GPP release 15," *IEEE Access*, vol. 7, pp. 127 639–127 651, 2019.

[60] A. Gupta and R. K. Jha, "A survey of 5G network: Architecture and emerging technologies," *IEEE Access*, vol. 3, pp. 1206–1232, 2015.

[61] S. K. Sharma, I. Woungang, A. Anpalagan, and S. Chatzinotas, "Toward tactile internet in beyond 5G era: Recent advances, current issues, and future directions," *IEEE Access*, vol. 8, pp. 56 948–56 991, 2020.

[62] A. Nasrallah, 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 Commun. Surv. & Tut.*, vol. 21, no. 1, pp. 88–145, 2019.

[63] I. Parvez, 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.*, vol. 20, no. 4, pp. 3098–3130, 2018.

[64] J. Sachs, G. Wikstrom, T. Dudda, R. Baldemair, and K. Kittichokechai, "5G radio network design for ultra-reliable low-latency communication," *IEEE Network*, vol. 32, no. 2, pp. 24–31, 2018.

[65] M. Yang, 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*, vol. 20, no. 1, pp. 4–18, 2015.

[66] V.-G. Nguyen, A. Brunstrom, K.-J. Grinnemo, and J. Taheri, "SDN/NFV-based mobile packet core network architectures: A survey," *IEEE Commun. Surveys & Tutorials*, vol. 19, no. 3, pp. 1567–1602, 2017.

[67] C. Bouras, A. Kollia, and A. Papazois, "SDN & NFV in 5G: Advancements and challenges," in *Proc. IEEE Conf. on Innov. in Clouds, Internet and Netw. (ICIN)*, 2017, pp. 107–111.

[68] X. Costa-Perez, 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*, vol. 24, no. 1, pp. 38–45, 2017.

[69] I. Afolabi, 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*, vol. 20, no. 3, pp. 2429–2453, 2018.

[70] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5G: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[71] M. Richart, J. Baliosian, J. Serrat, and J.-L. Gorricho, "Resource slicing in virtual wireless networks: A survey," *IEEE Trans. on Network and Service Management*, vol. 13, no. 3, pp. 462–476, 2016.

[72] A. A. Barakabitze, A. Ahmad, A. Hines, and R. Mijumbi, "5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, pp. 1–40, Feb. 2020.

[73] L. Ben Azzouz and I. Jamai, "SDN, slicing, and NFV paradigms for a smart home: A comprehensive survey," *Trans. on Emerging Telecommun. Techn.*, vol. 30, no. 10, pp. e3744.1–e3744.13, 2019.

[74] Z. Bojkovic, 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*. IGI Global, Hershey, PA, 2019, pp. 271–284.

[75] R. Su, 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*, vol. 33, no. 6, pp. 172–179, Nov.-Dec. 2019.

[76] S. Yi, C. Li, and Q. Li, "A survey of fog computing: Concepts, applications and issues," in *Proc. ACM Workshop on Mobile Big Data*, 2015, pp. 37–42.

[77] H. Tanaka, M. Yoshida, K. Mori, and N. Takahashi, "Multi-access edge computing: A survey," *Journal of Information Processing*, vol. 26, pp. 87–97, 2018.

[78] T. Taleb, 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*, vol. 19, no. 3, pp. 1657–1681, 2017.

[79] A. C. Baktir, 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*, vol. 19, no. 4, pp. 2359–2391, 2017.

[80] B. Blanco, 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*, vol. 54, pp. 216–228, Nov. 2017.

[81] N. F. S. de Sousa, D. A. L. Perez, R. V. Rosa, M. A. Santos, and C. E. Rothenberg, "Network service orchestration: A survey," *Computer Communications*, vol. 142-143, pp. 69–94, Jun. 2019.

[82] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache, "NFV orchestration framework addressing SFC challenges," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 16–23, 2017.

[83] Q. Duan, 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*, vol. 30, no. 5, pp. 10–16, 2016.

[84] N. Pitaev, 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.*, 2018, pp. 285–292.

[85] M. Kim and Y. S. Shao, "Hardware acceleration," *IEEE Micro*, vol. 38, no. 6, pp. 6–7, 2018.

[86] L. Linguaglossa, 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*, vol. 107, no. 4, pp. 746–764, 2019.

[87] H. Woesner, 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)*, 2018, pp. 1–6.

[88] T. Zhang, "Design of NFV platforms: A survey," *arXiv preprint arXiv:2002.11059*, 2020.

[89] F. Z. Yousaf, V. Sciancalepore, M. Liebsch, and X. Costa-Perez, "MANOaaS: A multi-tenant NFV MANO for 5G network slices," *IEEE Communications Magazine*, vol. 57, no. 5, pp. 103–109, 2019.

[90] B. Li, 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.*, 2016, pp. 1–14.

[91] G. S. Niemiec, 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*, vol. 22, no. 1, pp. 504–525, First Qu. 2020.

[92] C. Xu, 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.*, vol. 18, no. 4, pp. 2991–3029, 2016.

[93] T. Velte and A. Velte, *Cisco A Beginner's Guide*. McGraw-Hill Education Group, New York, 2013.

[94] M. O. Ball, T. Magnanti, C. Monma, and G. Nemhauser, *Network Routing*. Elsevier, Amsterdam, 1995.

[95] X. Chen, 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)*, 2003, pp. 69–77.

[96] D. Hovemeyer, J. K. Hollingsworth, and B. Bhattacharjee, "Running on the bare metal with GeekOS," *ACM SIGCSE Bulletin*, vol. 36, no. 1, pp. 315–319, 2004.

[97] M. Ben-Yehuda, 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, 2010, pp. 423–436.

[98] F. Zhang, 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*, 2011, pp. 203–216.

[99] Y. Yamato, "Openstack hypervisor, container and baremetal servers performance comparison," *IEICE Communications Express*, vol. 4, no. 7, pp. 228–232, 2015.

[100] A. Strunk, "Costs of virtual machine live migration: A survey," in *Proc. IEEE Eighth World Congress on Services*, 2012, pp. 323–329.

[101] E. F. Coutinho, 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*, vol. 70, no. 7-8, pp. 289–309, 2015.

[102] R. Cziva and D. P. Pezaros, "Container network functions: bringing NFV to the network edge," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 24–31, 2017.

[103] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, and A. Csaszar, "Elastic network functions: opportunities and challenges," *IEEE Network*, vol. 29, no. 3, pp. 15–21, 2015.

[104] D. Barach, 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*, 2018, pp. 1–2.

[105] L. Benini, 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)*, 2012, pp. 983–987.

[106] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, 2009.

[107] "Tensilica Customizable Processor and DSP IP: Application Optimization with the Xtensa Processor Generator," 2020, https://ip.cadence.com/ipportfolio/tensilica-ip, Last accessed June 8, 2020.

[108] O. Arnold, 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*, vol. 13, no. 3s, pp. 107.1–107.24, Mar. 2014.

[109] O. Arnold, 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*, 2014, pp. 25–33.

[110] F. Pauls, 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*. Springer, Cham, Switzerland, 2019, pp. 93–106.

[111] O. Arnold, B. Noethen, and G. Fettweis, "Instruction set architecture extensions for a dynamic task scheduling unit," in *Proc. IEEE Computer Society Annual Symp. on VLSI*, 2012, pp. 249–254.

[112] S. Wunderlich, F. H. Fitzek, and M. Reisslein, "Progressive multicore RLNC decoding with online DAG scheduling," *IEEE Access*, vol. 7, pp. 161 184–161 200, 2019.

[113] S. Yang, W.-H. Yeung, T.-I. Chao, K.-H. Lee, and C.-I. Ho, "Hardware acceleration for batched sparse codes," Mar. 19 2019, US Patent 10,237,782.

[114] J. Acevedo, 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*, vol. 7, no. 9, p. 180, 2018.

[115] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture." *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[116] D. Yokoyama, B. Schulze, F. Borges, and G. Mc Evoy, "The survey on ARM processors for HPC," *The Journal of Supercomputing*, vol. 75, pp. 7003–7036, 2019.

[117] K. Akdemir, 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," pp. 1–11, Jun. 2010, last accessed June 8, 2020. [Online]. Available: 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

[118] G. Hofemeier and R. Chesebrough, "Introduction to Intel AES-NI and Intel Secure Key Instructions, Intel Corp., White Paper," Jul. 2012, last accessed Apr. 3, 2020. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-aes-ni-and-intel-secure-key-instructions

[119] G. Cox, C. Dike, and D. Johnston, "Intel's digital random number generator (DRNG)," in *Proc. IEEE Hot Chips Symp. (HCS)*, 2011, pp. 1–13.

[120] S. A. Hassan, A. Hemeida, and M. M. Mahmoud, "Performance evaluation of matrix-matrix multiplications using Intel's Advanced Vector Extensions (AVX)," *Microprocessors and Microsystems*, vol. 47, pp. 369–374, 2016.

[121] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the Intel® Haswell processor," in *Proc. IEEE Int. Parallel and Distr. Proc. Symp. Workshop*, 2015, pp. 896–904.

[122] D. Takahashi, "An implementation of parallel 2-D FFT using Intel AVX instructions on multi-core processors," in *Proc. Int. Conf. on Algorithms and Arch. for Parallel Proc., Lecture Notes in Computer Science, vol. 7440*. Springer, Berlin, Heidelberg, 2012, pp. 197–205.

[123] H. Watanabe and K. M. Nakagawa, "SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions," *Computer Physics Communications*, vol. 237, pp. 1–7, 2019.
[124] H. Amiri and A. Shahbahrami, "SIMD programming using Intel vector extensions," *Journal of Parallel and Distributed Computing*, vol. 135, pp. 83–100, 2020.
[125] S. K. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming SIMD extensions on the Pentium III processor," *IEEE Micro*, vol. 20, no. 4, pp. 47–57, 2000.
[126] F. Cloutier, "CPU Identification," 2019, https://www.felixcloutier.com/x86/cpuid, last accessed June 2, 2020.
[127] Intel Corp., "Introduction to virtual machine extensions," 2020, https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1043.html, Last accessed June 9, 2020.
[128] ——, "Virtual machine extension (VMX) operation," 2020, 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.
[129] Linux Assembly, "Virtual machine extensions (VMX) instructions set," 2018, http://linasm.sourceforge.net/docs/instructions/vmx.php, Last accessed June 9, 2020.
[130] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of CPU pinning and partial CPU loads on performance and energy efficiency," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Comp.*, 2015, pp. 1–10.
[131] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "autopin – automated optimization of thread-to-core pinning on multicore systems," in *Trans. on High-Performance Embedded Architectures and Compilers III*. Springer, Berlin, Heidelberg, 2011, pp. 219–235.
[132] V. Gupta, 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)*, 2012, pp. 367–372.
[133] Intel Corp., "Intel® Data Direct I/O Technology (Intel DDIO) A Primer v1.0," Feb. 2012, 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.
[134] A. Cohen, F. Finkelstein, A. Mendelson, R. Ronen, and D. Rudoy, "On estimating optimal performance of CPU dynamic thermal management," *IEEE Computer Architecture Letters*, vol. 2, no. 1, pp. 6–6, 2003.
[135] R. Schone, 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*, 2012, pp. 1–5.
[136] J. Charles, 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)*, 2009, pp. 188–197.
[137] H. B. Jang, 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*, vol. 63, no. 5, pp. 1316–1322, 2012.
[138] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Proc. IEEE Int. Symp. on Low Power Electr. and Design (ISLPED)*, 2007, pp. 38–43.
[139] M. Wolf, *The Physics of Computing*. Morgan Kaufman - Elsevier, Burlington, MA, 2016.
[140] ——, *High-Performance Embedded Computing: Applications in Cyber-physical Systems and Mobile Computing*. Newnes, Boston, MA, 2014.
[141] F. P. Miller, A. F. Vandome, and J. McBrewster, *Advanced Configuration and Power Interface*. Alphascript Publishing, 2009.
[142] "Intel® Speed Select Technology (Intel® SST)," 2019, last accessed June 9, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/speed-select-technology-article.html
[143] T. Kim, T. Koo, and E. Paik, "SDN and NFV benchmarking for performance and reliability," in *Proc. IEEE Asia-Pacific Netw. Operations and Management Symp. (APNOMS)*, 2015, pp. 600–603.
[144] A. Pellegrini, 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*, vol. 40, no. 2, pp. 53–62, 2020.
[145] Amazon Web Services, Inc., "AWS Graviton Processor," 2020, https://aws.amazon.com/ec2/graviton/, last accessed June 2, 2020.
[146] D. Park, 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*, vol. 63, no. 3, pp. 622–636, 2012.
[147] S. Kumar, 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)*, 2002, pp. 117–124.
[148] S. Lakshmanamurthy, M. C. Hunsaker, M. T. Klinglesmith, B. Fanning, E. Tamari, J. Murray, and R. P. Adler, "Providing multiple decode options for a system-on-chip (SoC) fabric," Apr. 29 2014, US Patent 8,713,240.
[149] ARM Holdings, "AMBA AXI and ACE Protocol Specification," 2019, last accessed June 9, 2020. [Online]. Available: https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf
[150] N. Meng, M. Strassmaier, and J. Erwin, "LS-DYNA® Performance on Intel® Scalable Solutions," in *Proc. Int. LS-DYNA® Users Conf.*, Jun. 2018, pp. 1–8.
[151] S. M. Tam, 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)*, 2018, pp. 34–36.
[152] N. Beck, S. White, M. Paraschou, and S. Naffziger, "'Zeppelin': An SoC for multichip architectures," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 40–42.
[153] "Infinity Fabric (IF) – AMD," last accessed June 2, 2020. [Online]. Available: https://en.wikichip.org/wiki/amd/infinity_fabric
[154] P. Teich, "The heart of AMD's EPYC comeback is Infinity Fabric," 2017, last accessed June 2, 2020. [Online]. Available: https://www.nextplatform.com/2017/07/12/heart-amds-epyc-comeback-infinity-fabric/
[155] C. McGinnis, "PCI-SIG® Fast Tracks Evolution to 32 GT/s with PCI Express 5.0 Architecture," *News Release, June*, vol. 7, 2017.
[156] CXL Consortium, "Compute Express Link™," 2019, last accessed June 9, 2020. [Online]. Available: https://www.computeexpresslink.org/download-the-specification
[157] Intel Corp., "Logical PHY interface (LPIF) specification," 2019, last accessed June 9, 2020. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/logical-phy-interface-specification.pdf
[158] CCIX® Consortium Incorp., "An Introduction to CCIX® White Paper," 2020, https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf, Last accessed June 9, 2020.
[159] "The Gen-Z Consortium," 2020, https://genzconsortium.org, Last accessed June 9, 2020.
[160] J. Stuecheli, 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*, vol. 62, no. 4/5, pp. 8–1, 2018.
[161] L. Zhang, 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.*, 2009, pp. 385–390.
[162] D. Nagaraj and C. Gianos, "Intel® Xeon® processor D: The first Xeon processor optimized for dense solutions," in *Proc. IEEE Hot Chips Symp. (HCS)*, 2015, pp. 1–22.
[163] R. Rooney and N. Koyle, "Introducing Micron® DDR5 SDRAM: More Than a Generational Update," 2019, last accessed June 9, 2020. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/white-paper/ddr5_new_features_white_paper.pdf
[164] M. Weiland, A. Jackson, N. Johnson, and M. Parsons, "Exploiting the performance benefits of storage class memory for HPC and HPDA workflows," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, pp. 79–94, 2018.
[165] G. W. Burr, 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*, vol. 32, no. 4, p. 040802, Jul. 2014.
[166] J. Ray, V. George, I. M. Sodhi, and J. R. Wilcox, "Common platform for one-level memory architecture and two-level memory architecture," Mar. 21 2017, US Patent 9,600,413.
[167] Intel Corp., "Intel® Optane DC Persistent Memory - Content Delivery Networks Use Case," 2019, https://builders.intel.com/docs/networkbuilders/intel-optane-dc-

persistent-memory-content-delivery-networks-use-case.pdf, Last accessed June 9, 2020.

[168] A. Sainio, "NVDIMM: Changes are here so what's next," 2016, last accessed June 9, 2020. [Online]. Available: https://www.snia.org/sites/default/files/SSSI/NVDIMM%20-%20Changes%20are%20Here%20So%20What's%20Next%20-%20final.pdf

[169] Q. Xu, 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.*, 2015, pp. 6.1–6.11.

[170] D. Han, R.-H. Shiao, and L. Xu, "Graceful shutdown with asynchronous DRAM refresh of non-volatile dual in-line memory module," Jan. 11 2018, US Patent App. 15/261,397.

[171] NVidia Fermi, "Nvidia's next generation CUDA compute architecture," *NVidia, Santa Clara, CA*, 2009.

[172] U. Farooq, Z. Marrakchi, and H. Mehrez, "FPGA architectures: An overview," in *Tree-based Heterogeneous FPGA Architectures*. Springer, New York, NY, 2012, pp. 7–48.

[173] Q. Scheitle, 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*, 2018, pp. 1–2.

[174] Marvell, "NITROX® III Security Processor Family," 2020, https://www.marvell.com/products/security-solutions/nitrox-security-processors/nitrox-iii.html, Last accessed June 9, 2020.

[175] Intel Corp., "Intel® scalable I/O virtualization technical specification, reference number: 337679-001, revision: 1.0," Jun. 2018, last accessed June 9, 2020. [Online]. Available: https://software.intel.com/sites/default/files/managed/cc/0e/intel-scalable-io-virtualization-technical-specification.pdf

[176] ——, "Intel Quickassist Technology Accelerator Abstraction Layer (AAL), White Paper, Platform-level Services for Accelerators," last accessed June 9, 2020. [Online]. Available: https://blog-assets.oss-cn-shanghai.aliyuncs.com/18951/6103fadf4dd3a0dfbd0d637308a94b8e99e799d2.pdf

[177] ——, "Intel® QuickAssist Adapter 8960 and 8970 Product Brief," 2019, last accessed June 9, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/10-25-40-gigabit-adapters/quickassist-adapter-8960-8970-brief.html

[178] ——, "Intel® Data Streaming Accelerator Preliminary Architecture Specification," 2019, last accessed June 9, 2020. [Online]. Available: https://software.intel.com/sites/default/files/341204-intel-data-streaming-accelerator-spec.pdf

[179] J. Macri, "AMD's next generation GPU and high bandwidth memory architecture: FURY," in *Proc. IEEE Hot Chips Symp. (HCS)*, 2015, pp. 1–26.

[180] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. IEEE Symp. on VLSI Techn. (VLSIT)*, 2012, pp. 87–88.

[181] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[182] A. Chen, S. Datta, X. S. Hu, M. T. Niemier, T. Š. Rosing, and J. J. Yang, "A survey on architecture advances enabled by emerging beyond-CMOS technologies," *IEEE Design & Test*, vol. 36, no. 3, pp. 46–68, 2019.

[183] Q. Zhu, 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.*, 2013, pp. 1–7.

[184] O. Castañeda, M. Bobbett, A. Gallyas-Sanhueza, and C. Studer, "PPAC: A versatile in-memory accelerator for matrix-vector-product-like operations," in *Proc. IEEE Int. Conf. on Application-specific Sys., Arch. and Proc. (ASAP)*, vol. 2160, 2019, pp. 149–156.

[185] N. Power, S. Harte, N. D. McDonnell, and A. Cunningham, "System, apparatus and method for real-time activated scheduling in a queue management device," Apr. 4 2019, US Patent App. 15/719,769.

[186] R. Wang, 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," Oct. 5 2017, US Patent App. 15/087,154.

[187] N. D. McDonnell, Z. Zhu, and J. Mangan, "Power aware load balancing using a hardware queue manager," Feb. 7 2019, US Patent App. 16/131,728.

[188] Y. Le, 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.*, 2017, pp. 506–519.

[189] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An infrastructure for inline acceleration of network applications," in *Proc. USENIX Annual Techn. Conf.*, 2019, pp. 345–362.

[190] S. Choi, M. Shahbaz, B. Prabhakar, and M. Rosenblum, "$\lambda$-NIC: Interactive serverless compute on programmable smartnics," *arXiv preprint arXiv:1909.11958*, 2019.

[191] Y. Chen, 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.*, 2016, pp. 26.1–26.17.

[192] J. Regula, "Using non-transparent bridging in PCI Express systems, White Paper," pp. 1–31, Jun. 2004, last accessed Apr. 4, 2020. [Online]. Available: https://docs.broadcom.com/doc/12353428

[193] V. Nguyen, 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*, vol. 8, pp. 78 293–78 314, 2020.

[194] J. Sugerman, 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*, 2001, pp. 1–14.

[195] J. Plouffe, 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," Jul. 8 2014, US Patent 8,776,050.

[196] S. Sengupta, 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*, vol. 194, pp. 105 596.1–105 596.33, 2020.

[197] D. Datta, 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*, 2020, pp. 1–8.

[198] AMD, "ZEN Zeppelin block diagram," 2019, last accessed Apr. 4, 2020. [Online]. Available: https://pc.watch.impress.co.jp/video/pcw/docs/1108/270/p6.pdf

[199] "Intel® Xeon® Processor Scalable Family Technical Overview," 2019, last accessed June 2, 2020. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview

[200] I. Gerszberg, 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," Mar. 28 2000, US Patent 6,044,403.

[201] B. Benton, "CCIX, Gen-Z, OpenCAPI: Overview & Comparison," in *Proc. Ann. Workshop Open Fabric Alliance*, 2017.

[202] Intel Corp., "White Paper: Accelerating High-Speed Networking with Intel® I/OAT," 2019, last accessed June 2, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/io/i-o-acceleration-technology-paper.html

[203] A. Li, 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*, 2019.

[204] V. W. Lee, 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*, vol. 38, no. 3, pp. 451–460, 2010.

[205] S. Koehler, J. Curreri, and A. D. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Computing*, vol. 34, no. 4-5, pp. 217–230, 2008.

[206] Intel Corporation, "Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration, UG-20136, 2020.05.11," 2020, https://www.intel.com/content/www/us/en/programmable/documentation/tnc1513987819990.html, last accessed June 2, 2020.

[207] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: a survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 72.1–72.39, 2018.

[208] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll, “A survey on agent-based simulation using hardware accelerators,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 131.1–131.35, 2019.

[209] C.-Y. Gui, 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*, vol. 34, no. 2, pp. 339–371, 2019.

[210] M. Clark, “A new ×86 core architecture for the next generation of computing.” in *Proc. IEEEE Hot Chips Symp.*, 2016, pp. 1–19.

[211] AMD, “Zen Microarchitectures AMD,” 2020, https://en.wikichip.org/wiki/amd/microarchitectures/zen, last accessed June 2, 2020.

[212] S. Pontarelli, M. Bonola, and G. Bianchi, “Smashing OpenFlow’s "atomic" actions: Programmable data plane packet manipulation in hardware,” *Int. Journal of Netw. Management*, vol. 29, no. 1, pp. e2043.1–e2043.20, 2019.

[213] S. Choi, 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*, 2019, pp. 78–84.

[214] V. Herdt, 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.*, 2019, pp. 1–6.

[215] L. Morais, 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.*, 2019, pp. 861–872.

[216] J. Lu, 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.*, vol. 30, no. 4, pp. 953–964, 2018.

[217] Y. Zhou, F. He, N. Hou, and Y. Qiu, “Parallel ant colony optimization on multi-core SIMD CPUs,” *Future Generation Computer Systems*, vol. 79, pp. 473–487, 2018.

[218] P.-H. Wang, C.-H. Li, and C.-L. Yang, “Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture,” in *Proc. ACM Ann. Design Autom. Conf.*, 2016, pp. 1–6.

[219] G. Papadimitriou, 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.*, 2017, pp. 503–516.

[220] W. Bao, 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)*, vol. 13, no. 4, pp. 51:1–51:26, Dec. 2016.

[221] R. Begum, 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)*, 2016, pp. 161–168.

[222] J. Krzywda, 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*, vol. 81, pp. 114–128, 2018.

[223] E. Matthews 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)*, 2017, pp. 1–4.

[224] Y.-K. Choi, 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.*, 2016, pp. 1–6.

[225] F. Abdallah, 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*, vol. 137, pp. 106 006.1–106 006.13, Nov. 2019.

[226] M. Owaida, 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*, vol. 13, no. 1, pp. 71–85, 2019.

[227] M. Kekely, L. Kekely, and J. Kořenek, “General memory efficient packet matching FPGA architecture for future high-speed networks,” *Microproc. and Microsys.*, vol. 73, pp. 102 950.1–102 950.12, 2020.

[228] S. Karandikar, 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.*, 2020, pp. 715–731.

[229] J. Nie, 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.*, 2019, pp. 6–10.

[230] X. Yi, J. Duan, and C. Wu, “GPUNFV: A GPU-accelerated NFV system,” in *Proc. ACM Asia-Pacific Workshop on Networking*, 2017, pp. 85–91.

[231] M. Mardani, 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*, vol. 59, no. 8, pp. 5186–5205, 2013.

[232] M. Karunaratne, 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)*, 2017, pp. 1–6.

[233] G. Ansaloni, P. Bonzini, and L. Pozzi, “EGRA: A coarse grained reconfigurable architectural template,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 1062–1074, 2010.

[234] A. K. Jain, 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)*, 2016, pp. 1–8.

[235] S. Yazdanshenas and V. Betz, “Interconnect solutions for virtualized field-programmable gate arrays,” *IEEE Access*, vol. 6, pp. 10 497–10 507, 2018.

[236] K. Kang, 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*, 2016, pp. 1465–1468.

[237] M. Coppola, 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.

[238] W.-C. Tsai, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen, “Networks on chips: structure and design methodologies,” *Journal of Electrical and Computer Engineering*, vol. 2012, no. 509465, pp. 1–15, 2012.

[239] D. Goehringer, L. Meder, M. Hubner, and J. Becker, “Adaptive multi-client network-on-chip memory,” in *Proc. IEEE Int. Conf. on Reconfig. Comp. and FPGAs*, 2011, pp. 7–12.

[240] V. Y. Raparti, 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*, vol. 3, no. 2, pp. 72–85, 2017.

[241] A. B. Ahmed and A. B. Abdallah, “Adaptive fault-tolerant architecture and routing algorithm for reliable many-core 3D-NoC systems,” *Journal of Parallel and Distributed Computing*, vol. 93, pp. 30–43, 2016.

[242] S. H. Gade and S. Deb, “HyWin: Hybrid wireless NoC with sandboxed sub-networks for CPU/GPU architectures,” *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1145–1158, 2016.

[243] B. Bahrami, M. A. J. Jamali, and S. Saeidi, “A hierarchical architecture based on traveling salesman problem for hybrid wireless network-on-chip,” *Wireless Networks*, vol. 25, pp. 2187–2200, 2019.

[244] S. Mnejja, 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*, vol. 9, no. 6, pp. 913.1–913.19, 2020.

[245] Y. Ouyang, 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*, vol. 8, pp. 22 650–22 663, 2020.

[246] V. Catania, 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)*, vol. 14, no. 1, pp. 9.1–9.24, Nov. 2017.

[247] B. Yu, 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*, vol. 26, no. 7, pp. 478–480, 2016.

[248] S. H. Gade, S. S. Ram, and S. Deb, “Millimeter wave wireless interconnects in deep submicron chips: Challenges and opportunities,” *Integration*, vol. 64, pp. 127–136, 2019.

[249] A. Scionti, S. Mazumdar, and A. Portero, “Software defined network-on-chip for scalable CMPs,” in *Proc. IEEE Int. Conf. on High Perf. Comp. & Simul. (HPCS)*, 2016, pp. 112–115.

[250] ——, “Towards a scalable software defined network-on-chip for next generation cloud,” *Sensors*, vol. 18, no. 7, pp. 2330.1–2330.24, 2018.

[251] F. D. Nunes 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)*, 2019, pp. 224–225.

[252] R. Sandoval-Arechiga, 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.*, 2016, pp. 129–130.

[253] M. Ruaro, 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*, 2019, pp. 1–8.

[254] J. Bashir, E. Peter, and S. R. Sarangi, "A survey of on-chip optical interconnects," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 115.1–115.34, 2019.

[255] A. Reza, "Online multi-application mapping in photonic Network-on-Chip with mesh topology," *Optical Switching and Networking*, vol. 25, pp. 100–108, 2017.

[256] M. R. Yahya, 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.

[257] Y. Hsu, 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*, vol. 30, no. 11, pp. 1052–1055, 2018.

[258] H. Gu, K. Chen, Y. Yang, Z. Chen, and B. Zhang, "MRONoC: A low latency and energy efficient on chip optical interconnect architecture," *IEEE Photonics Journal*, vol. 9, no. 1, pp. 1–12, 2017.

[259] R. Yao 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.

[260] K. Wang, S. Qi, Z. Chen, Y. Yang, and H. Gu, "SMONoC: Optical network-on-chip using a statistical multiplexing strategy," *Optical Switching and Networking*, vol. 34, pp. 1–9, 2019.

[261] L. Huang, 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.

[262] Y. Ye, 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.

[263] J. S. Dehkordi and V. Tralli, "Interference analysis for optical wireless communications in Network-on-Chip (NoC) scenarios," *IEEE Transactions on Communications*, vol. 68, no. 3, pp. 1662–1674, 2019.

[264] S. Chandna, N. Naas, and H. Mouftah, "Software defined survivable optical interconnect for data centers," *Optical Switching and Networking*, vol. 31, pp. 86–99, 2019.

[265] J. Wang, 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*, vol. 3, no. 4, pp. 407–413, 2016.

[266] K. K. Chang, 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*, vol. 44, no. 1, pp. 323–336, 2016.

[267] H. Hassan, 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)*, 2016, pp. 581–593.

[268] K. K. Chang, 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)*, 2016, pp. 568–580.

[269] S. Khan, 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*, 2017, pp. 27–40.

[270] S. Khan, 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)*, 2016, pp. 239–250.

[271] A. Chen, "A review of emerging non-volatile memory (NVM) technologies and applications," *Solid-State Electronics*, vol. 125, pp. 25–38, 2016.

[272] Z. Wang, 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*, vol. 39, no. 3, pp. 343–346, 2018.

[273] D. Kang, 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*, vol. 52, no. 1, pp. 210–217, 2016.

[274] M. M. Ozdal, 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)*, 2016, pp. 166–177.

[275] A. Caulfield, E. Chung, A. Putnam, H. Angepat, D. Firestone, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "Configurable clouds," *IEEE Micro*, vol. 37, no. 3, pp. 52–61, 2017.

[276] A. Caulfield, P. Costa, and M. Ghobadi, "Beyond SmartNICs: Towards a fully programmable cloud," in *Proc. IEEE Int. Conf. on High Perf. Switching and Routing (HPSR)*, 2018, pp. 1–6.

[277] J. Gray, "GRVI Phalanx: A massively parallel RISC-V FPGA accelerator accelerator," in *Proc. IEEE Int. Symp. on Field-Progr. Custom Comp. Mach. (FCCM)*, 2016, pp. 17–20.

[278] N. Kapre and J. Gray, "Hoplite: Building austere overlay NoCs for FPGAs," in *Proc. IEEE Int. Conf. on Field Progr. Logic and Appl.*, 2015, pp. 1–8.

[279] Y. Lee, W. Kang, and H. Son, "An Internet traffic analysis method with mapreduce," in *Proc. IEEE/IFIP Network Operations and Management Symposium Workshops*, 2010, pp. 357–361.

[280] H. Song, J. Gong, and H. Chen, "Network map reduce," *arXiv preprint arXiv:1609.02982*, 2016.

[281] K. Neshatpour, M. Malik, A. Sasan, S. Rafatirad, and H. Homayoun, "Hardware accelerated mappers for Hadoop MapReduce streaming," *IEEE Trans. on Multi-Scale Computing Sys.*, vol. 4, no. 4, pp. 734–748, 2018.

[282] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: A novel approach for encrypted traffic classification using deep learning," *Soft Computing*, vol. 24, no. 3, pp. 1999–2012, 2020.

[283] S. Zhang, 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.*, 2016, pp. 1–12.

[284] T. Chen, 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*, vol. 42, no. 1, pp. 269–284, 2014.

[285] M. N. Bojnordi 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)*, 2016, pp. 1–13.

[286] Intel Corporation, "Configurable Spatial Accelerator (CSA)," 2020, https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator, last accessed June 2, 2020.

[287] H. Zhang, "The end of the x86 dominance in databases?" in *Proc. Biennial Conf. on Innovative Data Sys. Res. (CIDR)*, 2019, p. 1.

[288] M. G. Dosanjh, 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)*, 2019, pp. 210–220.

[289] F. Daoud, 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.*, 2016, pp. 6.1–6.8.

[290] L. Baldanzi, 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*. Springer, Cham, Switzerland, 2019, pp. 11–18.

[291] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 99–107, 2005.

[292] Y. Zhong, 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.*, 2019, pp. 388–397.

[293] V. Seshadri, 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.*, 2017, pp. 273–287.

[294] B. Li, 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*, 2019, pp. 381–386.

[295] S. Li, 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.*, 2016, pp. 1–6.

[296] Z. Ni, 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)*, 2019, pp. 1–6.

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

[298] M. Tork, 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.*, 2020, pp. 117–131.

[299] A. Venkat, 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)*, 2019, pp. 42–55.

[300] A. Venkat and D. M. Tullsen, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor," in *Proc. ACM/IEEE Int. Symp. on Comp. Arch.*, 2014, pp. 121–132.

[301] C. Pan 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*, vol. 5, no. 1, pp. 75–87, 2015.

[302] D. U. Lee, 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)*, 2014, pp. 432–433.

[303] R. Mahajan, 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*. Wiley – IEEE Press, Hoboken, NJ, 2019, pp. 487–499.

[304] R. Mahajan, 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.*, vol. 9, no. 10, pp. 1952–1962, 2019.

[305] C. Gonzalez, E. Fluhr, D. Dreps, D. Hogenmiller, R. Rao, J. Paredes, M. Floyd, M. Sperling, R. Kruse, V. Ramadurai *et al.*, "3.1 POWER9$^{\text{TM}}$: 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)*, 2017, pp. 50–51.

[306] P. Andreades, 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*, vol. 32, pp. 51–60, 2019.

[307] W. Li, 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*, vol. 31, pp. 183–192, 2019.

[308] S. Fernandes, B. C. Oliveira, and I. S. Silva, "Using NoC routers as processing elements," in *Proc. ACM Symposium on Integrated Circuits and System Design*, 2009, pp. 1–6.

[309] S. R. Fernandes, B. Oliveira, M. Costa, and I. S. Silva, "Processing while routing: A network-on-chip-based parallel system," *IET Computers & Digital Techniques*, vol. 3, no. 5, pp. 525–538, 2009.

[310] K. A. Zhezlov, 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.*, 2020, pp. 1911–1914.

[311] Y. Zhang, 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*, vol. 25, no. 5, pp. 1–9, 2019.

[312] H. Choi, 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*, vol. 72, pp. 102 942.1–102 942.1–11, 2020.

[313] F. Broquedis, 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*. Springer, Berlin, Heidelberg, Germany, 2009, pp. 79–92.

[314] J. Zhang, 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)*, 2020, pp. 287–302.

[315] S. Durai, 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.

[316] H. Akinaga and H. Shima, "Resistive random access memory (ReRAM) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[317] Y. Zhang, A. Rucker, M. Vilim, R. Prabhakar, W. Hwang, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *Proc. Int. Symp. on Computer Arch.*, 2019, pp. 615–628.

[318] C. Stunkel, 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*, vol. 64, no. 3/4, pp. 3:1–3:10, May–July 2020.

[319] K. Hamidouche 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*, 2020, pp. 336–347.

[320] S. S. Sharkawi and G. Chochia, "Communication protocol optimization for enhanced GPU performance," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 9:1–9:9, May–July 2020.

[321] D. Reis, 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.

[322] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, pp. 1–16, Mar. 2020.

[323] J. Li, Z. Sun, J. Yan, X. Yang, Y. Jiang, and W. Quan, "DrawerPipe: A reconfigurable pipeline for network processing on FPGA-Based SmartNIC," *Electronics*, vol. 9, no. 1, pp. 59.1–59.24, 2020.

[324] G. Belocchi, 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*, 2020, pp. 1–6.

[325] L. Guo, 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*, vol. 27, pp. 50–57, 2018.

[326] T. Zhang, 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.*, 2019, pp. 68–81.

[327] Intel Corporation, "Intel® Architecture Instruction Set Extensions and Future Features Programming Reference," 2020, https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, Last accessed June 9, 2020.

[328] C. Sieber, 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.*, 2017, pp. 7–12.

[329] P. Jain, 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)*, 2016.

[330] S. Gobriel, *Energy Efficiency in Communications and Networks*. IntechOpen, London, UK, 2012.

[331] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-latency networking with the OS stack and dedicated NICs," in *Proc. USENIX Ann. Techn. Conf.*, 2016, pp. 43–56.

[332] Intel Corp., "Intel® Resource Director Technology (Intel® RDT), Unlock System Performance in Dynamic Environ-

ments," 2019, last accessed June 9, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html

[333] ——, "Intel oneAPI Product Brief," 2020, last accessed June 9, 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/oneapi-product-brief.html

[334] X. Ge, 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*, vol. 44, no. 4, pp. 353–354, 2014.

[335] D. Nehama, R. Shiveley, J. Gasparakis, and R. Love, "Developing High-Performance, Flexible SDN & NFV Solutions with Intel® Open Network Platform Server Reference Architecture," pp. 1–8, 2014, last accessed June 9, 2020. [Online]. Available: http://docplayer.net/4394470-Developing-high-performance-flexible-sdn-nfv-solutions-with-intel-open-network-platform-server-reference-architecture.html

[336] A. Fiessler, 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*, vol. 25, no. 6, pp. 3655–3669, 2017.

[337] A. Langley, 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*, 2017, pp. 183–196.

[338] Y. Moon, 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)*, 2020, pp. 77–92.

[339] M. Ruiz, 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.*, 2019, pp. 286–292.

PRATEEK SHANTHARAMA is a Ph.D. student at Arizona State University, Tempe. He received his B.E. degree in Electronics and Communication Engineering from the Siddaganga Institute of Technology, Tumakuru, India, in 2014 and his M.Tech. degree in Computer Network Engineering from The National Institute of Engineering, Mysore, India, in 2016. His current research interests lie in wireless communication, networking, 5G, and SDN.

AKHILESH S. THYAGATURU is a Sr. Software Engineer with the Programmable Solutions Group (PSG), Intel Corporation, Chandler, AZ, USA, and an Adjunct Faculty with the School of Electrical, Computer, and Energy Engineering at Arizona State University (ASU), Tempe. He received the Ph.D. in electrical engineering from Arizona State University, Tempe, in 2017. He serves as reviewer for various journals including the *IEEE Communications Surveys & Tutorials*, *IEEE Transactions of Network and Service Management*, and *Optical Fiber Technology*. He was with Qualcomm Technologies Inc., San Diego, CA, USA, as an Engineer from 2013 to 2015.

MARTIN REISSLEIN (S'96-M'98-SM'03-F'14) is a Professor in the School of Electrical, Computer, and Energy Engineering at Arizona State University (ASU), Tempe. He received the Ph.D. in systems engineering from the University of Pennsylvania in 1998. He currently serves as Associate Editor for the *IEEE Transactions on Mobile Computing*, the *IEEE Transactions on Education*, and *IEEE Access* as well as *Computer Networks*. He is currently Associate Editor-in-Chief for the *IEEE Communications Surveys & Tutorials* and Co-Editor-in-Chief of *Optical Switching and Networking*. He chaired the steering committee of the *IEEE Transactions on Multimedia* from 2017–2019 and was an Associate Editor of the *IEEE/ACM Transaction on Networking* from 2009–2013. He received the IEEE Communications Society Best Tutorial Paper Award in 2008, a Friedrich Wilhelm Bessel Research Award from the Alexander von Humboldt Foundation in 2015, as well as a DRESDEN Senior Fellowship in 2016 and in 2019.